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.
  - Contributors: G. Morrisett, K. Crary, N. Glew, D. Grossman, T. Jim, C. Hawblitzel, M. Hicks, L. Hornof, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic
  - See http://www.cs.cornell.edu/talc

- Suggested Reading
  - G. Morrisett, D. Walker, K. Crary, N. Glew. From System-F to Typed Assembly Language. [13]
  - G. Morrisett, K. Crary, N. Glew, D. Walker. Stack-Based Typed Assembly Language. [12]
- 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:

```
fact(n,a) =
      if (n ≤ 0) then
            a
      else
            fact(n−1,a×n)
```

Assembly language code:

```
% r1 holds n, r2 holds a, r31 holds return address
% which expects the result in r1

fact:   ble r1, L2          % if n ≤ 0 goto L2
        mul r2, r2, r1      % a := a × n
        sub r1, r1, 1       % n := n−1
        jmp fact            % goto fact

L2:     mov r1, r2          % result := a
        jmp r31             % jump to return address
```

TAL-0

Syntax of a simple RISC-like assembly language.

- Registers: $r \in \{r1, r2, r3, \ldots\}$
- Labels: $L \in$ Identifier
- Integers: $n \in [-2^{k-1}..2^{k-1})$
- Blocks: $B ::= \texttt{jmp}\, v \mid i; B$
- Instrs: $i ::= aop\, r_d, r_s, v \mid bop\, r, v \mid \texttt{mov}\, r, v$
- Operands: $v ::= r \mid n \mid L$
- Arithmetic Ops: $aop ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \cdots$
- Branch Ops: $bop ::= \texttt{beq} \mid \texttt{bgt} \mid \cdots$

TAL-0 Abstract Machine

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

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

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

Operational Semantics

$$(H, R, \texttt{mov}\, r_d, v; B) \longmapsto (H, R[r_d := R(v)], B)$$

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

$$(H, R, \texttt{jmp}\, v) \longmapsto (H, R, B)$$
where $R(v) = L$ and $H(L) = B$

$$(H, R, \texttt{beq}\, r, v; B) \longmapsto (H, R, B)$$
where $R(r) \neq 0$

$$(H, R, \texttt{beq}\, r, v; B) \longmapsto (H, R, B')$$
where $R(r) = 0, R(v) = L,$ and $H(L) = B'$

The other instructions (`sub`, `bgt`, etc.) follow a similar pattern.

Error Conditions

- The abstract machine is *stuck* if there is no transition from the current state to some next state.
- The *stuck states* define the "bad" things that may happen.
- Our type system will ensure that the machine never gets stuck.
- Example stuck states:
  - $(H, R, \texttt{add}\, r_d, r_s, v; B)$ and r_s or v aren't ints
  - $(H, R, \texttt{jmp}\, v)$ and v isn't a label, or
  - $(H, R, \texttt{beq}\, r, v{:}B)$ 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:

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

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 τ_1, register r_2 must contain ...
- 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

```
% r1 holds n, r2 holds a, r31 holds return address
% which expects the result in r1

fact:   {r1:int, r2:int, r31:{r1:int} → { }} → { }
        ble r1, L2          % if n ≤ 0 goto L2
        mul r2, r2, r1      % a := a × n
        sub r1, r1, 1       % n := n−1
        jmp fact            % goto fact

L2:     {r2:int, r31:{r1:int} → { }} → { }
        mov r1, r2          % result := a
        jmp r31             % jump to return address
```

Mis-typed Program

$fact$: $\{r_1{:}int, r_{31}{:}\{r_1{:}int\} \rightarrow \{\ \}\} \rightarrow \{\ \}$

```
ble r1, L2
mul r2, r2, r1      % ERROR! r2 doesn't have a type
mov r1, r3
jmp L1              % ERROR! no such label
```

$L2$: $\{r_2{:}int, r_{31}{:}\{r_1{:}int\} \rightarrow \{\ \}\} \rightarrow \{\ \}$

```
mov r31, r2
jmp r31             % ERROR! r31 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: Ψ will map labels to label types.
- Register Types: Γ will map registers to types.

Typing Operands

- integer literals are ints:

$$\Psi; \Gamma \vdash n : int$$

- lookup register types in Γ:

$$\Psi; \Gamma \vdash r : \Gamma(r)$$

- lookup label types in Ψ:

$$\Psi; \Gamma \vdash L : \Psi(L)$$

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.

$$\{r_1{:}\tau_1, \ldots, r_{i-1}{:}\tau_{i-1}, r_i{:}\tau_i\} \leq \{r_1{:}\tau_1, \ldots, r_{i-1}{:}\tau_{i-1}\}$$

- 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:

$$\frac{\Gamma' \leq \Gamma}{\Gamma \rightarrow \{\,\} \leq \Gamma' \rightarrow \{\,\}}$$

- Subtyping is also reflexive and transitive

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

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

Typing Instructions

- The judgment for instructions looks like:

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

- Γ_1 describes the registers on input to the instruction (a *typing precondition*)
- Γ_2 describes the registers on output (a *typing postcondition*)
- Ψ is invariant. The types of heap objects will not change as the program executes (at least for now,...).

Typing Instructions

- Arithmetic operations:

$$\frac{\Psi; \Gamma \vdash r_s : int \quad \Psi; \Gamma \vdash v : int}{\Psi \vdash aop\, r_d, r_s, v : \Gamma \to \Gamma[r_d := int]}$$

- Conditional branches:

$$\frac{\Psi; \Gamma \vdash r : int \quad \Psi; \Gamma \vdash v : \Gamma \to \{\ \}}{\Psi \vdash bop\, r, v : \Gamma \to \Gamma}$$

- Data movement:

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash \texttt{mov}\, r, v : \Gamma \to \Gamma[r_d := \tau]}$$

Basic Block Typing

- All basic blocks end in the jump instruction:

$$\frac{\Psi; \Gamma \vdash v : \Gamma \rightarrow \{\,\}}{\Psi \vdash \texttt{jmp}\, v : \Gamma \rightarrow \{\,\}}$$

  Since a `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:

$$\frac{\Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2 \quad \Psi \vdash B : \Gamma_2 \rightarrow \{\,\}}{\Psi \vdash i; B : \Gamma_1 \rightarrow \{\,\}}$$

- 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:

$$\frac{\texttt{Dom}(H) = \texttt{Dom}(\Psi) \quad \forall L \in \texttt{Dom}(H).\Psi \vdash H(L) : \Psi(L)}{\vdash H : \Psi}$$

- Register file typing:

$$\frac{\forall r \in \texttt{Dom}(\Gamma).\Psi; \{\,\} \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$$

- Machine typing:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash B : \Gamma \rightarrow \{\,\}}{\vdash (H, R, B)}$$

Type Safety

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

- **Theorem: Type Safety.** If $\vdash \Sigma$ and $\Sigma \longmapsto^{*} \Sigma'$ then Σ 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

- $\tau = int$ 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 Σ_2 such that $\Sigma_1 \longmapsto \Sigma_2$.

Proof: By cases on the form of the code block in Σ_1.

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

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \texttt{jmp}\, v : \Gamma \to \{\,\}}{\vdash (H, R, \texttt{jmp}\, v)}$$

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

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \texttt{jmp}\, v : \Gamma \to \{\,\}}$$

By Canonical Forms, $R(v) = L$ and $L \in \texttt{Dom}(H)$. Therefore, the operational rule for jumps applies and Σ_1 is not stuck:
$(H, R, \texttt{jmp}\, v) \longmapsto (H, R, H(L))$

Proof: Preservation

Lemma: Preservation. If $\vdash \Sigma_1$ and $\Sigma_1 \longmapsto \Sigma_2$ then $\vdash \Sigma_2$.
Proof: By cases on the form of Σ_1.

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

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \texttt{jmp}\, v : \Gamma \to \{\,\}}{\vdash (H, R, \texttt{jmp}\, v)}$$

and the operational rule must be:

$$\begin{array}{l}(H, R, \texttt{jmp}\, v) \longmapsto (H, R, B) \\ \text{where } R(v) = L \text{ and } H(L) = B\end{array}$$

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:

$$\frac{\Psi; \Gamma \vdash v : \Gamma \to \{\,\}}{\Psi \vdash \texttt{jmp}\, v : \Gamma \to \{\,\}}$$

Therefore, by Canonical Forms, we know

$$\Psi \vdash B : \Gamma \to \{\,\}$$

and hence

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash B : \Gamma \to \{\,\}}{\vdash (H, R, B)}$$

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: α
    - 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

$swap$: $\forall\alpha,\beta.\{r_1:\alpha, r_2:\beta, r_{31}:\{r_1:\beta, r_2:\alpha\}\rightarrow\{\ \}\}\rightarrow\{\ \}$

$\quad\quad \texttt{mov}\, r_3, r_1 \quad\quad \texttt{\%}\ \{r_1:\alpha, r_2:\beta, r_{31}:\{r_1:\beta, r_2:\alpha\}\rightarrow\{\ \}, r_3:\alpha\}$

$\quad\quad \texttt{mov}\, r_1, r_2$

$\quad\quad \texttt{mov}\, r_2, r_3$

$\quad\quad \texttt{jmp}\, r_{31}$

$swap\_ints$: $\{r_1:int, r_2:int, r_{31}:\{r_1:int, r_2:int\}\rightarrow\{\ \}\}\rightarrow\{\ \}$

$\quad\quad \texttt{jmp}\, swap[int, int]$

$swap\_int\_and\_label$: $\{r_1:int, r_2:\{r_2:int\}\rightarrow\{\ \}\}\rightarrow\{\ \}$

$\quad\quad \texttt{mov}\, r_{31}, L$

$\quad\quad \texttt{jmp}\, swap[int, \{r_2:int\}\rightarrow\{\ \}]$

L: $\quad\{r_1:\{r_2:int\}\rightarrow\{\ \}, r_2:int\}\rightarrow\{\ \}$

$\quad\quad \texttt{jmp}\, r_1$

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

callee: $\forall\alpha.\{r_1 : int, r_5 : \alpha, r_{31} : \{r_1 : int, r_5 : \alpha\} \rightarrow \{\ \}\} \rightarrow \{\ \}$

```
        mov r4, r5        % save register r5
        mov r5, 7         % use register r5 for other work
        add r1, r1, r5
        mov r5, r4        % restore register r5
        jmp r31

caller: mov r5, 255       % will need r5  callee returns
        mov r1, 5
        mov r31, L
        jmp callee[int]   % callee[int] :
                          %   {r1 : int, r5 : int, r31 : {r1 : int, r5 : int} → { }}

L:      {r1 : int, r5 : int} → { }
        mul r3, r1, r5
        ...
```

Callee-saves Registers Bug

callee: $\forall\alpha.\{r_1 : int, r_5 : \alpha, r_{31} : \{r_1 : int, r_5 : \alpha\} \rightarrow \{\ \}\} \rightarrow \{\ \}$
    `mov` r_4, r_5
    `mov` $r_5, 7$
    `add` r_1, r_1, r_5
    `jmp` r_{31} `% ERROR!` $r_5 : int$

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

L: $\{r_1 : int, r_5 : int\} \rightarrow \{\ \}$
    `mul` r_3, r_1, r_5
    ...

- We can actually prove formally that *callee* preserves the values of its callee-saves registers. This fact is a property of *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 Γ and B
  - Control-flow operations substitute arguments types for type variables:

$(H, R, \texttt{jmp}\, v[\tau_1, \ldots, \tau_n]) \longmapsto (H, R, B[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n])$
where $R(v) = L$ and $H(L) = \forall \alpha_1, \ldots, \alpha_n . \Gamma \rightarrow \{\ \}.B$

$(H, R, \texttt{beq}\, r, v[\tau_1, \ldots, \tau_n]; B) \longmapsto (H, R, B'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n])$
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 (ie: it makes sense):

$$\frac{\mathit{FreeVars}(\tau) \subseteq \Delta}{\Delta \vdash \tau}$$

  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:

$$\frac{\begin{array}{ll} \forall L \in \mathtt{Dom}(H).\Psi; \alpha_1, \ldots, \alpha_n \vdash B : \Gamma \rightarrow \{\,\} & \\ H(L) = \forall \alpha_1, \ldots, \alpha_n.\Gamma.B & \text{(for all } L\text{)} \\ \Psi(L) = \forall \alpha_1, \ldots, \alpha_n. \rightarrow \{\,\} & \end{array}}{\vdash H : \Psi}$$

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:
  - $S ::= \texttt{nil} \mid v :: S$
- New instructions:
  - $i ::= \texttt{salloc}\, n \mid \texttt{sfree}\, n \mid \texttt{sld}\, r_d, n \mid \texttt{sst}\, r_s, n$
- 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.
  - `salloc` corresponds to subtracting n from a stack-pointer register (e.g. `sub` sp, sp, n)
  - `sfree` corresponds to adding n to the stack pointer (e.g. `add` sp, sp, n)
  - `sst` corresponds to writing a value into offset n from the stack pointer (e.g. `st` $sp(n), r$)
  - `sld` corresponds to reading a value from offset n relative to the stack pointer (e.g. `ld` $r, sp(n)$)
- CISC-like instructions (e.g. push/pop)can be synthesized.
  - `push` v = `salloc` 1; `sst` $v, 1$
  - `pop` r = `sld` $r, 1$; `sfree` 1

Simple Stack-Based Program

- A recursive version of the factorial function:

```
factrec(n) =
    if n ≤ 0 then
        1
    else
        n*factrec (n − 1)
```

```
factrec: % r1 holds argument n, r31 holds return address
         % which expects the result in r1

         bgt r1, L1         % n > 0, goto L1
         mov r1, 1
         jmp r31            % n ≤ 0, return 1

L1:      salloc 2           % allocate space for frame
         sst r31, 1         % save return address
         sst r1, 2          % save n
         sub r1, r1, 1      % n := n − 1
         mov r31, RA        % return address := RA
         jmp factrec        % do recursive call, result in r1

RA:      % result in r1
         sld r2, 2          % restore n into r2
         sld r31, 1         % restore return address
         mul r1, r1, r2     % result := n * fact(n − 1)
         jmp r31            % return
```

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.

$$(H, R, S, \texttt{salloc}\, n; B) \longmapsto (H, R, \overbrace{? :: \cdots :: ?}^{n} :: S, B)$$

$$(H, R, v_1 :: \cdots :: v_n :: S, \texttt{sfree}\, n; B) \longmapsto (H, R, S, B)$$

$$(H, R, S, \texttt{sld}\, r, n; B) \longmapsto (H, R[r := v_n], S, B)$$
where $S = v_1 :: \cdots :: v_n :: S'$

$$(H, R, S_1, \texttt{sst}\, r, n; B) \longmapsto (H, R, S_2, B)$$
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:
  - $\sigma ::= \texttt{nil} \mid \tau :: \sigma \mid \rho$
- The `nil` type represents the empty stack.
- The type $\tau :: \sigma$ represents a stack $v :: S$ where τ is the type of v and σ is the type of S.
- The type ρ 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:

$$\{sp : int :: \rho, r_1 : int, \ldots\}$$

- In addition, we'll let label types be polymorphic over stack types:

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

- 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

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

- Stack allocation:

$$\frac{}{\Psi; \Delta \vdash \texttt{salloc}\, n : \Gamma[sp := \sigma] \to \Gamma[sp := \underbrace{? :: \cdots :: ?}_{n} :: \sigma]}$$

- Stack free:

$$\frac{}{\Psi; \Delta \vdash \texttt{sfree}\, n : \Gamma[sp := \tau_1 :: \cdots :: \tau_n :: \sigma] \to \Gamma[sp := \sigma]}$$

- Stack load:

$$\frac{\Gamma(sp) = \tau_1 :: \cdots :: \tau_n :: \sigma}{\Psi; \Delta \vdash \texttt{sld}\, r, n : \Gamma \to \Gamma[r := \tau_n]}$$

- Stack store:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau \quad \Gamma(sp) = \tau_1 :: \cdots :: \tau_n :: \sigma}{\Psi; \Delta \vdash \texttt{sst}\, v, n : \Gamma \to \Gamma[sp := \tau_1 :: \cdots :: \tau :: \sigma]}$$

Typing Factrec (Bug)

```
type τ_ρ = {r1 : int, sp : ρ} → { }

factrec: ∀ρ.{sp : ρ, r1 : int, r31 : τ_ρ} → { }
        bgt r1, L1[ρ]
        mov r1, 1
        jmp r31

L1:     ∀ρ.{sp : ρ, r1 : int, r31 : τ_ρ} → { }
        salloc 2          % sp : ? :: ? :: ρ
        sst r31, 1        % sp : τ_ρ :: ? :: ρ
        sst r1, 2         % sp : τ_ρ :: int :: ρ
        sub r1, r1, 1
        mov r31, RA[ρ]    % r31 : {sp : τ_ρ :: int :: ρ, r1 : int} → { }
        jmp factrec[τ_ρ :: int :: ρ]

RA:     ∀ρ.{sp : τ_ρ :: int :: ρ, r1 : int} → { }
        sld r2, 2         % r2 : int
        sld r31, 1        % r31 : τ_ρ
        mul r1, r1, r2
        jmp r31           % ERROR!  sp : τ_ρ :: int :: ρ
```

Typing Factrec Corrected

```
type τ_ρ = {r1 : int, sp : ρ} → { }

factrec: ∀ρ.{sp : ρ, r1 : int, r31 : τ_ρ} → { }
        bgt r1, L1[ρ]
        mov r1, 1
        jmp r31

L1:     ∀ρ.{sp : ρ, r1 : int, r31 : τ_ρ} → { }
        salloc 2
        sst r31, 1
        sst r1, 2
        sub r1, r1, 1
        mov r31, RA[ρ]
        jmp factrec[τ_ρ :: int :: ρ]

RA:     ∀ρ.{sp : τ_ρ :: int :: ρ, r1 : int} → { }
        sld r2, 1           % r2 : int
        sld r31, 2          % r31 : τ_ρ
        mul r1, r1, r2
        sfree 2             % sp : ρ
        jmp r31
```

Another Example

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

caller: $\forall\rho'.\{sp : \tau_{code} :: \rho'\} \rightarrow \{\ \}$

```
salloc 1
mov r1, 17
sst r1, 1
mov r31, RA[ρ']
jmp callee[τcode :: ρ']
```

callee: $\forall\rho.\{sp : int :: \rho, r_{31} : \{sp : \rho, r_1 : int\} \rightarrow \{\ \}\} \rightarrow \{\ \}$

```
sld r1, 1
add r1, r1, r1
sst r1, 2          % ERROR!
sfree 1
jmp r31
```

RA: $\forall\rho'.\{sp : \tau_{code} :: \rho', r_1 : int\} \rightarrow \{\ \}$

. . .

- 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 τ:

$$
\begin{array}{lcl}
\texttt{size}(\mathit{int}) & = & 1 \\
\texttt{size}(\mathit{float}) & = & 2 \\
\texttt{size}(\forall \alpha_1, \ldots, \alpha_n . \Gamma \rightarrow \{\ \}) & = & 1 \\
\texttt{size}(?_{32}) & = & 1 \\
\texttt{size}(?_{64}) & = & 2
\end{array}
$$

  - But how do we compute the size of an abstract type α?

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:

$$\kappa ::= \texttt{Sz}(i) \mid \texttt{T}$$

- Type contexts Δ 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:

$$\frac{}{\Delta \vdash \mathit{int} :: \mathtt{Sz}(1)} \qquad \frac{}{\Delta \vdash \mathit{float} :: \mathtt{Sz}(2)}$$

$$\frac{}{\Delta \vdash \mathtt{nil} :: \mathtt{Sz}(0)} \qquad \frac{\Delta \vdash \tau :: \mathtt{Sz}(i) \qquad \Delta \vdash \sigma :: \mathtt{Sz}(j)}{\Delta \vdash (\tau :: \sigma) :: \mathtt{Sz}(i+j)}$$

$$\frac{}{\Delta, \alpha :: \kappa \vdash \alpha :: \kappa}$$

$$\frac{\Delta \vdash \tau :: \mathtt{Sz}(i)}{\Delta \vdash \tau :: \mathtt{T}} \qquad \frac{\Delta \vdash \tau :: \mathtt{T} \qquad \Delta \vdash \sigma :: \mathtt{T}}{\Delta \vdash \tau :: \sigma :: \mathtt{T}}$$

- Modified stack load:

$$\frac{\begin{array}{c}\Gamma(sp) = \tau_1 :: \cdots :: \tau_m :: \sigma \\ \Delta \vdash (\tau_1 :: \cdots :: \tau_{m-1} :: \mathtt{nil}) :: \mathtt{Sz}(n-1) \qquad \Delta \vdash \tau_m :: \mathtt{Sz}(1)\end{array}}{\Psi; \Delta \vdash \mathtt{sld}\, r, n : \Gamma \to \Gamma[r := \tau_m]}$$

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

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

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:

$$
\begin{array}{l}
\texttt{letrec} \\
\quad \texttt{fun}\ \ \mathit{fact}\ (n{:}\mathit{int}) : \mathit{int} = \\
\qquad \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * \mathit{fact}(n-1) \\
\texttt{in} \\
\quad \mathit{fact}\ 6
\end{array}
$$

Tiny Syntax

- Types:

$$\tau ::= \mathit{int} \mid \tau_1 \rightarrow \tau_2$$

- Expressions:

$$\begin{array}{rcl} e & ::= & x \mid f \mid n \mid e_1 + e_2 \mid e_1\ e_2 \mid \texttt{if}\ e_1 = 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \\ & & \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \end{array}$$

- Function declarations:

$$d ::= \texttt{fun}\ f(x{:}\tau_1) : \tau_2 = e$$

- Programs:

$$P ::= \texttt{letrec}\ d_1\ \cdots\ d_n\ \texttt{in}\, e$$

A Tiny Type System

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

Expressions:

$$\frac{}{\Phi \vdash x : \Phi(x)}$$

$$\frac{}{\Phi \vdash f : \Phi(f)}$$

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

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

$$\frac{\Phi \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Phi \vdash e_2 : \tau_1}{\Phi \vdash e_1\ e_2 : \tau_2}$$

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

$$\frac{\Phi \vdash e_1 : \tau_1 \quad \Phi, x{:}\tau_1 \vdash e_2 : \tau_2}{\Phi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Typing Tiny Programs

Declarations:

$$\frac{\Phi, x{:}\tau_1 \vdash e : \tau_2}{\Phi \vdash \texttt{fun } f(x{:}\tau_1) : \tau_2 = e : (f{:}\tau_1 \to \tau_2)}$$

Programs:

$$\frac{\begin{array}{c}\Phi = f_1{:}\tau_{1,1} \to \tau_{1,2}, \ldots, f_n{:}\tau_{n,1} \to \tau_{n,2} \\ \Phi \vdash d_i : (f_i{:}\tau_{i,1} \to \tau_{i,2}) \qquad \Phi \vdash e : \mathit{int}\end{array}}{\vdash \texttt{letrec}\, d_1 \ \cdots \ d_n \,\texttt{in}\, e}$$

- 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 ($\mathcal{T}[\![\cdot]\!]$) maps Tiny types to TAL types
- Integers:

$$\mathcal{T}[\![int]\!] = int$$

- 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.

$$\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \forall \rho.\{sp : \mathcal{K}[\![\tau_2, \rho]\!] :: \mathcal{T}[\![\tau_1]\!] :: \rho\} \to \{\ \}$$

where

$$\mathcal{K}[\![\tau, \sigma]\!] = \{sp : \sigma, r_a : \mathcal{T}[\![\tau]\!]\} \to \{\ \}$$

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
    - $\texttt{I}(M)$ 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
- $\texttt{T}(e)$ is the source type of expression e
  - Technically, we should thread the Tiny typing context Φ 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:

$$\mathcal{E}[\![x]\!]_{M,\sigma} = \texttt{sld}\, r_a, M(x)$$

- Function variables:

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

- Integer constants:

$$\mathcal{E}[\![n]\!]_{M,\sigma} = \texttt{mov}\, r_a, n$$

- Addition:

$$\begin{aligned} &\mathcal{E}[\![e_1 + e_2]\!]_{M,\sigma} = \\ &\qquad \mathcal{E}[\![e_1]\!]_{M,\sigma} \\ &\qquad \texttt{push}\, r_a \\ &\qquad \mathcal{E}[\![e_2]\!]_{\texttt{I}(M),int::\sigma} \\ &\qquad \texttt{pop}\, r_t \\ &\qquad \texttt{add}\, r_a, r_t, r_a \end{aligned}$$

Expression Translation

- Function Call:

$$
\begin{aligned}
&\mathcal{E}[\![e_1\ e_2]\!]_{M,\sigma} = \\
&\qquad \mathcal{E}[\![e_1]\!]_{M,\sigma} \\
&\qquad \texttt{push}\, r_a \\
&\qquad \mathcal{E}[\![e_2]\!]_{\texttt{I}(M),\mathcal{T}[\![\tau_1 \to \tau_2]\!]::\sigma} \\
&\qquad \texttt{pop}\, r_t \\
&\qquad \texttt{push}\, r_a \\
&\qquad \texttt{push}\, L_r[\rho] \\
&\qquad \texttt{jmp}\, r_t[\sigma] \\
&L_r : \forall\rho.\mathcal{K}[\![\tau_2, \sigma]\!]
\end{aligned}
$$

where $\texttt{T}(e_1) = \tau_1 \to \tau_2$
and L_r is fresh

- Conditional:

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{if}\ e_1 = 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_{M,\sigma} = \\
&\qquad \mathcal{E}[\![e_1]\!]_{M,\sigma} \\
&\qquad \texttt{bneq}\, r_a, L_{else}[\rho] \\
&\qquad \mathcal{E}[\![e_2]\!]_{M,\sigma} \\
&\qquad \texttt{jmp}\, L_{end}[\rho] \\
&L_{else} : \forall\rho.\{sp : \sigma\} \\
&\qquad \mathcal{E}[\![e_3]\!]_{M,\sigma} \\
&\qquad \texttt{jmp}\, L_{end}[\rho] \\
&L_{end} : \forall\rho.\mathcal{K}[\![\tau, \sigma]\!]
\end{aligned}
$$

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

- Exercise: Translate the let-expression

Program Translation

- Function translation:

$$
\begin{aligned}
&\mathcal{F}[\![\texttt{fun}\ f(x{:}\tau_1) : \tau_2 = e]\!] = \\
&\qquad L_f : \mathcal{T}[\![\tau_1 \to \tau_2]\!] \\
&\qquad \mathcal{E}[\![e]\!]_{[x:=2],\mathcal{K}[\![\tau_2,\rho]\!]::\mathcal{T}[\![\tau_1]\!]::\rho} \\
&\qquad \texttt{pop}\, r_t \\
&\qquad \texttt{sfree}\, 1 \\
&\qquad \texttt{jmp}\, r_t
\end{aligned}
$$

- Program translation:

$$
\begin{aligned}
&\mathcal{P}[\![\texttt{letrec}\, d_1\ \cdots\ d_n\, \texttt{in}\, e]\!] = \\
&\qquad \mathcal{F}[\![d_1]\!] \\
&\qquad \ldots \\
&\qquad \mathcal{F}[\![d_n]\!] \\
&L_{main} : \forall\rho.\{sp : \mathcal{K}[\![int,\rho]\!] :: \rho\} \\
&\qquad \mathcal{E}[\![e]\!]_{\cdot,\mathcal{K}[\![int,\rho]\!]::\rho} \\
&\qquad \texttt{pop}\, r_t \\
&\qquad \texttt{jmp}\, r_t
\end{aligned}
$$

  - To run the program, jump to L_{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:

letrec
    fun $\mathit{fact}\ (n{:}\mathit{int}) : \mathit{int} =$
        if $n = 0$ then 1 else $n * \mathit{fact}(n-1)$
in
    $\mathit{fact}\ 6$

Example: Compiling Fact

```
L_fact: ∀ρ.{sp : K[[int]] :: int :: ρ}
        sld r_a, 2             % load argument
        bneq r_a, L_else[ρ]    % n = 0?
        mov r_a, 1             % return 1
        jmp L_end

L_else: ∀ρ.{sp : K[[int]] :: int :: ρ}
        sld r_a, 2             % begin multiplication (load n)
        push r_a
        mov r_a, L_fact        % begin 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[ρ]
        jmp r_t[int :: K[[int, ρ]] :: int :: ρ]
L_r:    ∀ρ.{sp : int :: K[[int, ρ]] :: int :: ρ, r_a : int}
        pop r_t                % load n
        mul r_a, r_t, r_a      % n ∗ fact(n − 1)
        jmp L_end[ρ]

L_end:  ∀ρ.{sp : K[[int, ρ]] :: int :: ρ, r_a : int}
        pop r_t                % pop return address
        sfree 1                % 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 $\texttt{push}\, v = \texttt{salloc}\, 1; \texttt{sst}\, v, 1$
  - First, we write down the typing rules for the sequence, specialized to specific operands:

$$\frac{\overline{\Psi; \Delta \vdash \texttt{salloc}\, 1 : \Gamma[sp := \sigma] \to \Gamma[sp := \,?\, :: \sigma]} \quad \mathcal{D}}{\Psi; \Delta \vdash \texttt{salloc}\, 1; \texttt{sst}\, v, 1 : \Gamma[sp := \sigma] \to \Gamma[sp := \tau :: \sigma]}$$

$$\mathcal{D} = \frac{\Psi; \Delta; \Gamma[sp := \,?\, :: \sigma] \vdash v : \tau}{\Psi; \Delta \vdash \texttt{sst}\, v, 1 : \Gamma[sp := \,?\, :: \sigma] \to \Gamma[sp := \tau :: \sigma]}$$

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

$$\frac{\Psi; \Delta; \Gamma[sp := \,?\, :: \sigma] \vdash v : \tau}{\Psi; \Delta \vdash \texttt{push}\, v : \Gamma[sp := \sigma] \to \Gamma[sp := \tau :: \sigma]}$$

Instruction Selection

- Since `push` v is statically equivalent to `salloc` 1; `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:

```
push 7
push 8
push 9
```

Can be replaced by:

```
salloc 1
sst 7, 1
salloc 1
sst 8, 1
salloc 1
sst 9, 1
```

Which can be further reduced to:

```
salloc 3
sst 7, 1
sst 8, 1
sst 9, 1
```

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:

```
        ...
        ∀ρ.{sp : K[[τ_return, ρ]] :: τ_{f−arg} :: ρ, r_a : τ_{g−arg}} → { }
        salloc 2
        sst L_r             % push return address
        sst r_a, 2          % push argument
        jmp L_g[τ_raddr :: τ_{f−arg} :: ρ]

L_r:    ∀ρ.{sp : τ_raddr :: τ_{f−arg} :: ρ, r_a : τ_ret} → { }
        pop r_t             % pop return address
        sfree 1             % throw away f's argument
        jmp r_t             % return
```

With tail-call optimization:

L_f:

```
        ...
        ∀ρ.{sp : τ_raddr :: τ_{f−arg} :: ρ, r_a : τ_{g−arg}} → { }
        sst r_a, 2
        jmp L_g[ρ]          % g will return to f's 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 Ψ.

- 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:

$$int \to bool$$

- TAL Type:

```
All a:T,b:T,c:T,r1:S,r2:S,e1:C,e2:C.
  {ESP: {EAX:bool, M:e1+e2, EBX:a, ESI:b, EDI:c,
   ESP:int::r1@{EAX:exn,ESP:r2,M:e1+e2}::r2}::int::r1@
   {EAX:exn,ESP:r2,M:e1+e2}::r2,
   EBP: sptr{EAX:exn,ESP:r2,M:e1+e2}::r2,
   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
- See Grossman and Morrisett [6] for a survey of techniques used in our implementation.

Summary of Type-Directed Compilation

- Type-directed and type-preserving compilation provides an *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 pathes 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: $\texttt{malloc}\, r_d, n$
  - Load from k^{th} component of the tuple: $\texttt{ld}\, r_d, r_s(k)$
  - Store into k^{th} component of the tuple: $\texttt{st}\, r_d(k), r_s$
- Tuple types: $\langle \tau_1, \ldots, \tau_n \rangle$

Tuple Operational Semantics

- Allocation:

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

- Load:

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

- Store:

$$(H[L = \langle v_1, \ldots, v_n \rangle], R, S, \texttt{st}\, r_d(k), r_s; B) \longmapsto$$
$$(H[L = \langle v_1, \ldots, v_{k-1}, R(r_s), v_{k+1}, \ldots, v_n \rangle], R, S, B)$$
$$\text{where } R(r_d) = L$$

Tuple Typing

- Allocation:

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

- Load:

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

- Store:

$$\frac{\Psi; \Delta; \Gamma \vdash r_d : \langle \tau_1, \ldots, \tau_n \rangle \quad \Psi; \Delta; \Gamma \vdash r_s : \tau_k \quad 1 \leq k \leq n}{\Psi; \Delta \vdash \texttt{st}\, r_d(k), r_s : \Gamma \rightarrow \Gamma}$$

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
- $C ::= \{\ell \mapsto \langle\tau_1, \ldots, \tau_n\rangle\} \mid \mathbf{1} \mid C_1 \otimes C_2 \mid \epsilon$
- ϵ is a logic variable
- ℓ is a label: either a label variable ϕ 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.

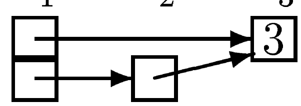
$$\frac{}{\Psi \vdash \{\,\} : \mathbf{1}}$$

$$\frac{\Psi \vdash H_1 : C_1 \quad \Psi \vdash H_2 : C_2}{\Psi \vdash H_1 \bowtie H_2 : C_1 \otimes C_2}$$

$$\frac{\Psi; \cdot \vdash v_i : \tau_i \quad \text{for } 1 \leq i \leq n}{\Psi \vdash \{L \mapsto \langle v_1, \ldots, v_n \rangle\} : \{L \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}}$$

- Example:

L_1: L_2: L_3:



$$\{L_1 \mapsto \langle S(L_3), S(L_2) \rangle\} \otimes$$
$$\{L_2 \mapsto \langle S(L_3) \rangle\} \otimes$$
$$\{L_3 \mapsto \langle int \rangle\}$$

Using Store Types

- New instructions:
  - `mmalloc` 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:
  - $\{hp : C, sp : \sigma, r_1 : \tau_1, \ldots, r_n : \tau_n\} \rightarrow \{\ \}$

Examples

```
foo:    ∀ε, ρ.{hp : ε, sp : ρ, r1 : int,
            r31 : {hp : ε, sp : ρ, r1 : int} → { }} → { }
        mmalloc φ, r2, n   % hp : ε ⊗ {φ ↦ ⟨?, ?⟩}, r2 : S(φ)
        mov r7, r2         % r7 : S(φ)
        st r7[1], r1       % hp : ε ⊗ {φ ↦ ⟨int, ?⟩}
        st r2[2], r1       % hp : ε ⊗ {φ ↦ ⟨int, int⟩}
        free r2            % hp : ε
        jmp r31
```

An error:

```
foo:    ∀ε, ρ.{hp : ε, sp : ρ, r1 : int,
            r31 : {hp : ε, sp : ρ, r1 : int} → { }} → { }
        mmalloc φ, r2, n      % hp : ε ⊗ {φ ↦ ⟨?, ?⟩}, r2 : S(φ)
        mov r7, r2            % r7 : S(φ)
        st r7[1], r1          % hp : ε ⊗ {φ ↦ ⟨int, ?⟩}
        st r2[2], r1          % hp : ε ⊗ {φ ↦ ⟨int, int⟩}
        jmp r31               % 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.

$$\frac{}{\cdot \vdash \mathbf{1}}$$

$$\frac{L, L' \vdash C}{L, \mathbf{1}, L' \vdash C}$$

$$\frac{L, C, C', L' \vdash C''}{L, C \otimes C', L' \vdash C''}$$

$$\frac{L \vdash C \quad L' \vdash C'}{L \bowtie L' \vdash C \otimes C'}$$

$$\frac{}{\{\phi \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \vdash \{\phi \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}}$$

$$\frac{}{\epsilon \vdash \epsilon}$$

- 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 Ψ and the type variables Δ, instruction i has register file precondition Γ and there exist types Δ' such that the postcondition Γ' 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:

$$\frac{\Psi; \Delta \vdash i : \Gamma \rightarrow [\Delta']\Gamma' \quad \Psi; \Delta, \Delta' \vdash B : \Gamma' \rightarrow \{\,\}}{\Psi; \Delta \vdash i; B : \Gamma \rightarrow \{\,\}}$$

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

$$\frac{\Psi; \Gamma \vdash v : \Gamma \rightarrow \{\,\}}{\Psi \vdash \texttt{jmp}\, v : \Gamma \rightarrow \{\,\}}$$

Instruction Typing Rules

$$\frac{\Gamma(hp) = C \quad \Gamma' = \Gamma[hp := C \otimes \{\phi \mapsto \overbrace{\langle ?, \ldots, ? \rangle}^{n}\}][r := S(\phi)]}{\Psi; \Delta \vdash \texttt{mmalloc}\ \phi, \texttt{r}, \texttt{n} : \Gamma \to [\phi]\Gamma'}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash r : S(\ell) \\ \Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \qquad \Gamma' = \Gamma[hp := C]\end{array}}{\Psi; \Delta \vdash \texttt{free}\ r : \Gamma \to [\,]\Gamma'}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash r_d : S(\ell) \qquad \Psi; \Delta; \Gamma \vdash r_s : \tau \\ \Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_k, \ldots, \tau_n \rangle\} \\ \Gamma' = \Gamma[hp := C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\}]\end{array}}{\Psi; \Delta \vdash \texttt{st}\ r_d(k), r_s : \Gamma \to [\,]\Gamma'}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash r_d : \tau_k \qquad \Psi; \Delta; \Gamma \vdash r_s : S(\ell) \\ \Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_k, \ldots, \tau_n \rangle\}\end{array}}{\Psi; \Delta \vdash \texttt{ld}\ r_d, r_s(k) : \Gamma \to [\,]\Gamma}$$

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.

$$\frac{\Gamma \leq \Gamma'}{\Psi; \Delta; \Gamma \vdash i : \Gamma \to [\,]\Gamma'}$$

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:

$$a ::= i \mid n \mid a_1 +_{32} a_2 \mid a_1 -_{32} a_2 \mid a_1 \times_{32} a_2 \mid a_1 \;\texttt{xor}\; a_2 \mid \cdots$$

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

- Logical connectives:

$$P ::= p \mid \texttt{true} \mid \texttt{false} \mid a_1 \leq_u a_2 \mid P_1 \supset P_2 \mid P_1 \wedge P_2 \mid \neg P \mid \cdots$$

- New types:
  - Singleton integers: $S(a)$
  - Array types: $\tau\ array(a)$

Refined Operand Typing

- New type contexts:

$$\Delta ::= \cdot \mid \Delta, \alpha :: \kappa \mid \Delta, P$$

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

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[P, \Delta'].\Gamma' \rightarrow \{\,\} \qquad \Delta \vdash \mathit{proof} : P \ \texttt{true}}{\Psi; \Delta; \Gamma \vdash v[\mathit{proof}] : \forall[\Delta'].\Gamma' \rightarrow \{\,\}}$$

  - 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:

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

- Array access:

$$\frac{\Psi; \Delta; \Gamma \vdash r_2 : \tau\ \mathit{array}(a) \qquad \Psi; \Delta; \Gamma \vdash r_3 : S(a_3) \qquad \Delta \vdash a_3 \leq_u a\ \texttt{true}}{\Psi; \Delta \vdash \texttt{ld}\, r_1, r_2(r_3) : \Gamma \rightarrow \Gamma[r_1 := \tau]}$$

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

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

- Conditional branches

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[P].\Gamma \rightarrow \{\ \} \qquad \Psi; \Delta; \Gamma \vdash r : S(a) \qquad \Delta, a \leq 0 \vdash P\ \texttt{true}}{\Psi; \Delta \vdash \texttt{ble}\, r, v : \Gamma \rightarrow [a > 0]\Gamma}$$

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*: $\forall\rho.\{sp:\rho, r_1:int,$
$r_{31}:\{r_1:int, sp:\rho\}\rightarrow\{\ \}\}\rightarrow\{\ \}$

`fact.tal`:

EXPORT `fact_e.tali`

factrec: $\forall\rho.\{sp:\rho, r_1:int,$
$r_{31}:\{r_1:int, sp:\rho\}\rightarrow\{\ \}\}\rightarrow\{\ \}$
`sub` $r_3, r_1, 1$
`ble` $r_3, L1[\rho]$
`jmp` r_{31}

*L*1: $\forall\rho.\{sp:\rho, r_1:int, r_3:int,$
$r_{31}:\{r_1:int, sp:\rho\}\rightarrow\{\ \}\}\rightarrow\{\ \}$
`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

[1] Karl Crary. A simple proof technique for certain parametricity results. In *International Conference on Functional Programming*, pages 82–89, Paris, France, September 1999.

[2] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[3] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998.

[4] Neal Glew. *Low-level Type Systems For Modularity and Object-oriented Constructs*. PhD thesis, Cornell University, January 2000.

[5] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, January 1999.

[6] Dan Grossman and Greg Morrisett. Scalable certification of native code: Experience from compiling to TALx86. Technical Report TR2000-1783, Cornell University, February 2000.

[7] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994.

[8] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In Robert Harper, editor, *Third International Workshop on Types in Compilation*, number 2071 in LNCS, pages 147–176, Montreal, Canada, March 2001.

[9] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, January 2001.

[10] Greg Morrisett. Lecture notes on language-based security, July 2001. Available at www.funtechs.org/lbs.

[11] 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 *ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[12] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[14] Frank Pfenning and Carsten Schürmann. system description: Twelf — a metalogical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in LNAI, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[15] John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information processing*, pages 513–523, 1983.

[16] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

[17] Carsten Schürmann. *Automating the Metatheory of Deductive Systems*. PhD thesis, Carnegie Mellon University, August 2000. Published as CMU Technical Report CMU-CS-00-146.

[18] Frederick Smith. *Certified Run-time Code Generation*. PhD thesis, Cornell University, 2001.

[19] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[20] TALx86. See http://www.cs.cornell.edu/talc for an implementation of Typed Assembly Language based on Intel's IA32 architecture.

[21] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[22] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

[23] David Walker. *Typed Memory Management*. PhD thesis, Cornell University, January 2001.

[24] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000.

[25] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, September 2000.

[26] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[27] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International conference on functional programming*, Florence, September 2001.

[28] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.