# CS 4110

# Programming Languages & Logics

## Lecture 37
## Typed Assembly Language

28 November 2012

# Schedule

## Last Friday

- Typed Assembly Language

## Today

- Polymorphism
- Stack Types

## Next Monday

- Compilation

# TAL-0 Review

- Syntax
- Semantics
- Type System
  - $\Psi; \Gamma \vdash v : \tau$
  - $\Psi \vdash i : \Gamma \to \Gamma'$
  - $\tau \leq \tau'$
  - $\vdash H : \Psi$
  - $\vdash R : \Gamma$
  - $\vdash (H, R, B)$

# TAL-0 Review

- Syntax
- Semantics
- Type System
  - $\Psi; \Gamma \vdash v : \tau$
  - $\Psi \vdash i : \Gamma \to \Gamma'$
  - $\tau \leq \tau'$
  - $\vdash H : \Psi$
  - $\vdash R : \Gamma$
  - $\vdash (H, R, B)$

**Theorem (Type Safety)**

*If* $\vdash \Sigma$ *and* $\Sigma \mapsto^{\star} \Sigma'$, *then* $\Sigma'$ *is not stuck.*

**Lemma (Progress and Preservation)**

- *If* $\vdash \Sigma_1$ *then there exists a* $\Sigma_2$ *such that* $\Sigma_1 \mapsto \Sigma_2$
- *If* $\vdash \Sigma_1$ *and* $\Sigma_1 \mapsto \Sigma_2$ *then* $\vdash \Sigma_2$

# TAL-1: Polymorphism

# Syntax

- Add type variables $\alpha$ and universal types $\forall\alpha.\tau$
- Allow code label types to be polymorphic
  $\forall\alpha,\beta.\,\{r_1 : \alpha, r_2 : \beta, r_3 : \{r_1 : \beta, r_2 : \alpha\} \rightarrow \{\}\} \rightarrow \{\}$
- Add type application $v\,[\tau]$
- Write $v\,[\tau_1, \ldots, \tau_k]$ for $v\,[\tau_1] \cdots [\tau_k]$

# Polymorphism Example

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

mov $r_3$,$r_1$
mov $r_1$,$r_2$
mov $r_2$,$r_3$
jmp $r_{31}$

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

jmp *swap* [int, int]

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

mov $r_{31}$,$L$
jmp *swap* [int, $\{r_2\text{:int}\} \to \{\}$]

$L : \{r_1 : \{r_2 : \text{int}\} \to \{\}, r_2 : \text{int}\} \to \{\}$

jmp $r_1$

# Callee-Saves Registers

## Common Strategy

- When calling a function...
- Save the contents of some registers on the stack...
- Allow the callee to save (and restore) other designated registers...
- If the callee does not use all registers, the cost of saving and restoring is not incurred...

## Correctness Critereon

Callee must return the callee-saves registers to the caller with the same values as when the function was invoked.

# Callee-Saves Example

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

mov $r_4$,$r_5$ % Save $r_5$

mov $r_5$,7 % Use $r_5$ for other work

add $r_1$,$r_1$,$r_5$

mov $r_5$,$r_4$ % Restore $r_5$

jmp $r_{31}$

$caller : \{\} \to \{\}$

mov $r_5$,255

mov $r_1$,5

mov $r_{31}$,$L$

jmp *callee*[int]

$L : \{r_1 : \text{int}, r_5 : \text{int}\} \to \{\}$

mul $r_3$,$r_1$,$r_5$

...

# Callee-Saves Bug

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

```
    mov r4,r5          % Save r5
    mov r5,7           % Use r5 for other work
    add r1,r1,r5
    mov r5,r4          % Restore r5
    jmp r31            % Error! r5:int
```

$caller : \{\} \to \{\}$

```
    mov r5,255
    mov r1,5
    mov r31,L
    jmp callee[int]
```

$L : \{r_1 : \text{int}, r_5 : \text{int}\} \to \{\}$

```
    mul r3,r1,r5
        ...
```

# Parametricity

- Can prove that the correct version preserves callee-saves registers

# Parametricity

- Can prove that the correct version preserves callee-saves registers
- This follows directly from *callee*'s polymorphic type!

$$\forall\alpha.\left\{\begin{array}{l} r_1 : \text{int}, \\ r_5 : \alpha, \\ r_{31} : \{r_1 : \text{int}, r_5 : \alpha\} \to \{\} \end{array}\right\} \to \{\}$$

# Parametricity

- Can prove that the correct version preserves callee-saves registers
- This follows directly from *callee*'s polymorphic type!

$$\forall\alpha.\left\{\begin{array}{l} r_1 : \mathrm{int}, \\ r_5 : \alpha, \\ r_{31} : \{r_1 : \mathrm{int}, r_5 : \alpha\} \rightarrow \{\} \end{array}\right\} \rightarrow \{\}$$

- Moral: polymorphism is useful for more than just code reuse

# Parametricity

- Can prove that the correct version preserves callee-saves registers
- This follows directly from *callee*'s polymorphic type!

$$\forall \alpha. \left\{ \begin{array}{l} r_1 : \text{int}, \\ r_5 : \alpha, \\ r_{31} : \{r_1 : \text{int}, r_5 : \alpha\} \rightarrow \{\} \end{array} \right\} \rightarrow \{\}$$

- Moral: polymorphism is useful for more than just code reuse
- Types can also be used to constrain the behavior of functions

# Parametricity

- Can prove that the correct version preserves callee-saves registers
- This follows directly from *callee*'s polymorphic type!

$$\forall \alpha.\ \left\{ \begin{array}{l} r_1 : \text{int}, \\ r_5 : \alpha, \\ r_{31} : \{r_1 : \text{int}, r_5 : \alpha\} \to \{\} \end{array} \right\} \to \{\}$$

- Moral: polymorphism is useful for more than just code reuse
- Types can also be used to constrain the behavior of functions

Theorems for free!

Philip Wadler
University of Glasgow*

June 1989

**Abstract**

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

**1 Introduction**

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

Say that $r$ is a function of type

$$r : \forall X.\ X^* \to X^*.$$

Here $X$ is a type variable, and $X^*$ is the type "list of $X$". From this, as we shall see, it is possible to conclude that $r$ satisfies the following theorem: for all types $A$ and $A'$ and every total function $a : A \to A'$ we have

$$a^* \circ r_A = r_{A'} \circ a^*.$$

Here $\circ$ is function composition, and $a^* : A^* \to A'^*$ is the function "map $a$" that applies $a$ elementwise to a list of $A$ yielding a list of $A'$, and $r_A : A^* \to A^*$ is the instance of $r$ at type $A$.

The intuitive explanation of this result is that $r$ must work on lists of $X$ for *any* type $X$. Since $r$ is provided with no operations on values of type $X$, all it can do is rearrange such lists, independent of the values contained in them. Thus applying $a$ to each element of a list and then rearranging yields the same result as rearranging and then applying $a$ to each element.

For instance, $r$ may be the function $reverse : \forall X.\ X^* \to X^*$ that reverses a list, and $a$ may be the function $code : Char \to Int$ that converts a character to its ASCII code. Then we have

$$\begin{array}{ll} & code^*\ (reverse_{Char}\ [\text{`a'}, \text{`b'}, \text{`c'}]) \\ = & [99, 98, 97] \\ = & reverse_{Int}\ (code^*\ [\text{`a'}, \text{`b'}, \text{`c'}]) \end{array}$$

which satisfies the theorem. Or $r$ may be the function $tail : \forall X.\ X^* \to X^*$ that returns all but the first element of a list, and $a$ may be the function $inc : Int \to Int$ that adds one to an integer. Then we have

$$\begin{array}{ll} & inc^*\ (tail_{Int}\ [1,2,3]) \\ = & [3,4] \\ = & tail_{Int}\ (inc^*\ [1,2,3]) \end{array}$$

which also satisfies the theorem.

On the other hand, say $r$ is the function $odds : Int^* \to Int^*$ that removes all odd elements from a list of integers, and say $a$ is $inc$ as before. Now we have

$$\begin{array}{ll} & inc^*\ (odds_{Int}\ [1,2,3]) \\ = & [2,4] \\ \neq & [4] \\ = & odds_{Int}(inc^*\ [1,2,3]) \end{array}$$

and the theorem is *not* satisfied. But this is not a counterexample, because *odds* has the wrong type: it is too specific, $Int^* \to Int^*$ rather than $\forall X.\ X^* \to X^*$.

This theorem about functions of type $\forall X.\ X^* \to X^*$ is pleasant but not earth-shaking. What is more exciting is that a similar theorem can be derived for *every* type.

*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

This is a slightly revised version of a paper appearing in: *4'th International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.



1

Paper: P. Wadler. "Theorems for Free!" In *FPCA*, pp. 347–359. September 1989.

# Operational Semantics

We need to make a few small changes to the operational semantics

# Operational Semantics

We need to make a few small changes to the operational semantics

- Heaps $H$ now map labels to type-labeled blocks:

  $H(L) = \forall\alpha_1, \ldots, \alpha_k.\, \Gamma \to \{\}.\, B$

# Operational Semantics

We need to make a few small changes to the operational semantics

- Heaps $H$ now map labels to type-labeled blocks:

  $H(L) = \forall \alpha_1, \ldots, \alpha_k.\, \Gamma \to \{\}.\, B$

- Type variables may appear free in both $\Gamma$ and $B$

# Operational Semantics

We need to make a few small changes to the operational semantics

- Heaps $H$ now map labels to type-labeled blocks:

  $H(L) = \forall \alpha_1, \ldots, \alpha_k.\, \Gamma \to \{\}.\, B$

- Type variables may appear free in both $\Gamma$ and $B$

- Control-flow operations substitute types

# Operational Semantics

We need to make a few small changes to the operational semantics

- Heaps $H$ now map labels to type-labeled blocks:

  $H(L) = \forall\alpha_1, \ldots, \alpha_k.\, \Gamma \to \{\}.\, B$

- Type variables may appear free in both $\Gamma$ and $B$

- Control-flow operations substitute types

  $(H, R, \text{jmp } v[\tau_1, \ldots, \tau_k]) \mapsto (H, R, B[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k])$
  where $R(v) = L$ and $H(L) = \forall\alpha_1, \ldots, \alpha_k.\, \Gamma \to \{\}.\, B$

# Operational Semantics

We need to make a few small changes to the operational semantics

- Heaps $H$ now map labels to type-labeled blocks:

  $$H(L) = \forall \alpha_1, \ldots, \alpha_k.\, \Gamma \rightarrow \{\}.\, B$$

- Type variables may appear free in both $\Gamma$ and $B$

- Control-flow operations substitute types

  $$(H, R, \text{jmp } v[\tau_1, \ldots, \tau_k]) \mapsto (H, R, B[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k])$$
  where $R(v) = L$ and $H(L) = \forall \alpha_1, \ldots, \alpha_k.\, \Gamma \rightarrow \{\}.\, B$

  $$(H, R, \text{beq } r, v[\tau_1, \ldots, \tau_k]; B) \mapsto (H, R, B'[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k])$$
  where $R(r) = 0$, $R(v) = L$, and $H(L) = \forall \alpha_1, \ldots, \alpha_k.\, \Gamma \rightarrow \{\}.\, B'$

# Typing Polymormphism

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

# Typing Polymormphism

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

## Type application

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall \alpha_1, \ldots, \alpha_k.\, \Gamma' \rightarrow \{\} \qquad \Delta \vdash \tau}{\Psi; \Delta; \Gamma \vdash v\,[\tau] : (\forall \alpha_2, \ldots, \alpha_k.\, \Gamma' \rightarrow \{\})[\tau/\alpha]}$$

# TAL-2: Stack Types

# Run-Time Stack

Almost every compiler uses a *run-time stack*

- What is a stack?
- A consecutive sequence of memory addresses with one end designated as the *top* of the stack
- Values are stored on the top of the stack and retrieved later
- The compiler may grow or shrink the stack as needed

Stack uses

- Local variables
- Spilled registers
- Return addresses

# Stack Syntax

- Machine states:
  $M ::= (H, R, S, B)$
- Stacks:
  $S ::= [] \mid v :: S$
- Instructions:
  $i ::= \cdots \mid \text{salloc } n \mid \text{sfree } n \mid \text{sld } r_d, n \mid \text{sst } v, n$
- Errors:
  - Free too many values
  - Read too deep in the stack
  - Write too deep in the stack

# Stack Instructions

The new stack instructions can be easily encoded:

- A designated register *sp* points to the top of the stack
- salloc *n* subtracts *n* from *sp* (i.e., sub *sp*, *sp*, *n*)
- sfree *n* adds *n* to *sp* (i.e., add *sp*, *sp*, *n*)
- sld $r_d$, *n* reads a value at offset *n* relative to *sp* (i.e., ld $r_d$, $sp(n)$)
- sst *v*, *n* writes a value at offset *n* relative to *sp* (i.e., st $sp(n)$, *v*)

CISC-like stack instructions can also be encoded:

- push *v* is salloc 1; sst *v*, 1
- pop $r_d$ is sld $r_d$, 1; sfree 1

# Example: Factorial

$$
\begin{aligned}
&\mathit{fact}(n) = \\
&\quad \text{if } n \leq 0 \text{ then } 1 \\
&\quad \text{else } n \times \mathit{fact}(n-1)
\end{aligned}
$$

# Example: Factorial

| | | |
|---|---|---|
| *fact*: | bgt $r_1$,$L1$ | % if $n > 0$, goto $L1$ |
| | mov $r_1$,1 | |
| | jmp $r_{31}$ | % if $n \leq 0$, return |
| $L1$ : | salloc 2 | % allocate space for frame |
| | sst $r_{31}$,1 | % save return address |
| | sst $r_1$,2 | % save $n$ |
| | sub $r_1$,$r_1$,1 | % $n := n - 1$ |
| | mov $r_{31}$,$L$ | % set return address |
| | jmp *fact* | % recursive call |
| $L$ : | sld $r_2$,2 | % restore $n$ |
| | sld $r_{31}$,1 | % restore return address |
| | sfree 2 | % free space for frame |
| | mul $r_1$,$r_1$,$r_2$ | % $result := n \times fact(n - 1)$ |
| | jmp $r_{31}$ | % return |

# Operational Semantics

$$\overline{(H, R, S, \text{salloc } n; B) \mapsto (H, R, ? :: \cdots :: ? :: S, B)}$$

# Operational Semantics

$$\frac{}{(H, R, S, \mathrm{salloc}\ n; B) \mapsto (H, R, \mathbf{?} :: \cdots :: \mathbf{?} :: S, B)}$$

$$\frac{}{(H, R, v_1 :: \cdots :: v_n :: S, \mathrm{sfree}\ n; B) \mapsto (H, R, S, B)}$$

# Operational Semantics

$$\frac{}{(H, R, S, \text{salloc } n; B) \mapsto (H, R, ? :: \cdots :: ? :: S, B)}$$

$$\frac{}{(H, R, v_1 :: \cdots :: v_n :: S, \text{sfree } n; B) \mapsto (H, R, S, B)}$$

$$\frac{S = v_1 :: \cdots :: v_n :: S'}{(H, R, S, \text{sld } r_d, n; B) \mapsto (H, R[r_d := v_n], S, B)}$$

# Operational Semantics

$$\frac{}{(H, R, S, \text{salloc } n; B) \mapsto (H, R, ? :: \cdots :: ? :: S, B)}$$

$$\frac{}{(H, R, v_1 :: \cdots :: v_n :: S, \text{sfree } n; B) \mapsto (H, R, S, B)}$$

$$\frac{S = v_1 :: \cdots :: v_n :: S'}{(H, R, S, \text{sld } r_d, n; B) \mapsto (H, R[r_d := v_n], S, B)}$$

$$\frac{S = v_1 :: \cdots :: v_n :: S'}{(H, R, S, \text{sst } v, n; B) \mapsto (H, R, v_1 :: \cdots :: R(v) :: S', B)}$$

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- [] represents empty stacks

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- $[]$ represents empty stacks
- $\tau :: \sigma$ represents stacks with top of type $\tau$ and rest of type $\sigma$

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- $[]$ represents empty stacks
- $\tau :: \sigma$ represents stacks with top of type $\tau$ and rest of type $\sigma$
- $\rho$ is a stack type variable

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- $[]$ represents empty stacks
- $\tau :: \sigma$ represents stacks with top of type $\tau$ and rest of type $\sigma$
- $\rho$ is a stack type variable
- Register file types contain a special variable *sp*

$$\{sp : \text{int} :: \text{int} :: [], r_1 : \text{int}, \ldots\}$$

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- $[]$ represents empty stacks
- $\tau :: \sigma$ represents stacks with top of type $\tau$ and rest of type $\sigma$
- $\rho$ is a stack type variable
- Register file types contain a special variable *sp*

$$\{sp : \text{int} :: \text{int} :: [], r_1 : \text{int}, \ldots\}$$

- Code label types can be polymorphic over stack types

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

# Type System

## Syntax

$$\sigma ::= [] \mid \tau :: \sigma \mid \rho$$

- $[]$ represents empty stacks
- $\tau :: \sigma$ represents stacks with top of type $\tau$ and rest of type $\sigma$
- $\rho$ is a stack type variable
- Register file types contain a special variable *sp*

$$\{sp : \text{int} :: \text{int} :: [], r_1 : \text{int}, \ldots\}$$

- Code label types can be polymorphic over stack types

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

- Junk values "?" have junk types "?"

# Typing Stacks

$$\boxed{\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2}$$

# Typing Stacks

$\boxed{\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2}$

## Stack allocation

$$\frac{\Gamma(sp) = \sigma}{\Psi; \Delta \vdash \mathsf{salloc}\ n : \Gamma \to \Gamma[sp := ? :: \cdots :: ? :: \sigma]}$$

# Typing Stacks

$\boxed{\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2}$

## Stack allocation

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

## Stack free

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

# Typing Stacks

$$\boxed{\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2}$$

Stack load

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

# Typing Stacks

$$\boxed{\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2}$$

## Stack load

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

## Stack store

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

# Example: Factorial Bug

```
fact: ∀ρ. {sp : ρ, r1 : int, r31 : {r1 : int, sp : ρ} → {}} → {}
      bgt r1,L1[ρ]
      mov r1,1
      jmp r31
L1 : ∀ρ. {sp : ρ, r1 : int, r31 : {r1 : int, sp : ρ} → {}} → {}
      salloc 2
      sst r31,1
      sst r1,2
      sub r1,r1,1
      mov r31,L[ρ]
      jmp fact
L : ∀ρ. {sp : {r1 : int, sp : ρ} → {} :: int :: ρ, r1 : int} → {}
      sld r2,2
      sld r31,1
      sfree 2
      mul r1,r1,r2
      jmp r31          % Error! sp : {r1 : int, sp : ρ} → {} :: int :: ρ
```

# Example: Callee Bug

$caller\colon \forall\rho.\ \{sp : \tau_{code} :: \rho\} \to \{\}$

salloc 1
mov $r_1$,17
sst $r_1$,1
mov $r_{31}$,$L[\rho]$
jmp $callee[\tau_{code} :: \rho]$

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

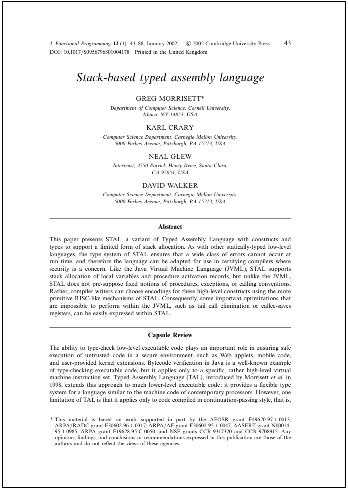
sld $r_1$,1
add $r_1$,$r_1$,$r_1$
sst $r_1$,2 % Error!
sfree 1
jmp $r_{31}$

$L : \forall\rho.\ \{sp : \tau_{code} :: \rho, r_1 : \text{int}\} \to \{\}$

$\ldots$

# Type Safety

- Type safety ensures we don't get stuck
- With a few additional features, can handle exceptions
- Paper: G. Morrisett, K. Crary, N. Glew, and D. Walker. "Stack-based Typed Assembly Language." In *JFP*. 12(1):43–88. January 2002.


*J. Functional Programming* **12** (1): 43–88, January 2002. © 2002 Cambridge University Press 43
DOI: 10.1017/S0956796801004178 Printed in the United Kingdom

*Stack-based typed assembly language*

GREG MORRISETT*
*Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*

KARL CRARY
*Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA*

NEAL GLEW
*Intertrust, 4750 Patrick Henry Drive, Santa Clara, CA 95054, USA*

DAVID WALKER
*Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA*

**Abstract**

This paper presents STAL, a variant of Typed Assembly Language with constructs and types to support a limited form of stack allocation. As with other statically-typed low-level languages, the type system of STAL ensures that a wide class of errors cannot occur at run time, and therefore the language can be adapted for use in certifying compilers where security is a concern. Like the Java Virtual Machine Language (JVML), STAL supports stack allocation of local variables and procedure activation records, but unlike the JVML, STAL does not pre-suppose fixed notions of procedures, exceptions, or calling conventions. Rather, compiler writers can choose encodings for these high-level constructs using the more primitive RISC-like mechanisms of STAL. Consequently, some important optimizations that are impossible to perform within the JVML, such as tail call elimination or callee-saves registers, can be easily expressed within STAL.

**Capsule Review**

The ability to type-check low-level executable code plays an important role in ensuring safe execution of untrusted code in a secure environment, such as Web applets, mobile code, and user-provided kernel extensions. Bytecode verification in Java is a well-known example of type-checking executable code, but it applies only to a specific, rather high-level virtual machine instruction set. Typed Assembly Language (TAL), introduced by Morrisett *et al.* in 1998, extends this approach to much lower-level executable code: it provides a flexible type system for a language similar to the machine code of contemporary processors. However, one limitation of TAL is that it applies only to code compiled in continuation-passing style, that is,

* This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-96-1-0317, ARPA/AF grant F30602-95-1-0047, AASERT grant N00014-95-1-0985, ARPA grant F19628-95-C-0050, and NSF grants CCR-9317320 and CCR-9708915. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

# Discussion

- Note that we didn't bake in a specific calling convention

# Discussion

- Note that we didn't bake in a specific calling convention
- Stacks plus jmp were sufficient

# Discussion

- Note that we didn't bake in a specific calling convention
- Stacks plus jmp were sufficient
- Easy to handle tail calls

# Discussion

- Note that we didn't bake in a specific calling convention
- Stacks plus jmp were sufficient
- Easy to handle tail calls
- Polymorphism is critical

# Discussion

- Note that we didn’t bake in a specific calling convention
- Stacks plus jmp were sufficient
- Easy to handle tail calls
- Polymorphism is critical
- Can encode many calling conventions:
  - Arguments on stack or in registers?
  - Results on stack or in registers?
  - Return address: caller pops? callee pops?
  - Registers: caller saves? callee saves?

# Discussion

- Note that we didn't bake in a specific calling convention
- Stacks plus jmp were sufficient
- Easy to handle tail calls
- Polymorphism is critical
- Can encode many calling conventions:
  - Arguments on stack or in registers?
  - Results on stack or in registers?
  - Return address: caller pops? callee pops?
  - Registers: caller saves? callee saves?

Moral: orthogonal combination of type system constructs makes it easy to scale language features