Stack-based Typed Assembly Language

International Workshop on TC '98

Greg Morrisett, Karl Crary, Neal Glew, David Walker

Presented by Sunae Seo
Department of Ropas, CS, KAIST
November 25, 2000

Introduction

TAL

- Type systems enforces many important invariants(catch internal errors)
- No compiler propagates type information through final stage $\Rightarrow$ motivation of low-level typed language s.t. *TAL(Typed Assembly Language)*
- Their compiler is based on
  - CPS (Continuation Passing Style)
  - heap-allocated closures

Why Stack?!

On the heap

- easy to implement control primitives (exceptions, coroutine threads...)
- better space properties because of shared environments
- unified memory management mechanism (eg. GC)

Instead, stack

- For imperative languages, (eg. Java, Pascal)
  - need for shared env. is greatly reduced
  - GC is not needed in some language.
- many architectures expect programs to behave in a stack-like fashion(eg. Pentium Pro).
- stack needs smaller memory foot print than heap

So, they do

adding a stack to TAL in order to give compiler writers the flexibility they need $\rightarrow$ STAL

Something in TAL

Program is (**H,RF,IS**)

- **H** : heap $\{l_1 \to h_1, ..., l_n \to h_n\}$
- **RF** : Register file $\{r_1 \to w_1, ..., r_n \to w_n\}$
- **IS** : Instruction Sequence (terminated by a jmp or halt)

Context $\vec{\alpha}$ (sometimes Δ)

- For $\underline{\forall[\vec{\alpha}].\Gamma}$ in types, freely occurred type variables in Γ
- For $\underline{\texttt{code}[\vec{\alpha}]\Gamma.I}$ in code blocks, freely occurred type variables in Γ, I

Syntax of TAL

types	$\tau ::= \alpha \mid int \mid \forall[\Delta].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	$\varphi ::= 0 \mid 1$
label assignments	$\Psi ::= \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
type assignments	$\Delta ::= \cdot \mid \alpha, \Delta$
register assignments	$\Gamma ::= \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
registers	$r ::= \mathtt{r1} \mid \cdots \mid \mathtt{r}k$
word values	$w ::= \ell \mid i \mid ?\tau \mid w[\tau] \mid pack\ [\tau, w]\ as\ \tau'$
small values	$v ::= r \mid w \mid v[\tau] \mid pack\ [\tau, v]\ as\ \tau'$
heap values	$h ::= \langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\Delta]\Gamma.I$
heaps	$H ::= \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	$R ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	$\iota ::= aop\ r_d, r_s, v \mid bop\ r, v \mid \mathtt{ld}\ r_d, r_s(i) \mid \mathtt{malloc}\ r[\vec{\tau}] \mid$ $\mathtt{mov}\ r_d, v \mid \mathtt{st}\ r_d(i), r_s \mid \mathtt{unpack}\ [\alpha, r_d], v \mid$
arithmetic ops	$aop ::= \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul}$
branch ops	$bop ::= \mathtt{beq} \mid \mathtt{bneq} \mid \mathtt{bgt} \mid \mathtt{blt} \mid \mathtt{bgte} \mid \mathtt{blte}$
instruction sequences	$I ::= \iota; I \mid \mathtt{jmp}\ v \mid \mathtt{halt}\ [\tau]$
programs	$P ::= (H, R, I)$

Fig. 2. Syntax of TAL

Type repr. example

before CPS

$$int \to bool$$

after CPS

$$(int, (bool \to void)) \to void$$

after Closure
introduced

$$\exists\alpha.\langle(\alpha, int, \exists\beta.\langle(\beta, bool) \to void, \beta\rangle) \to void, \alpha\rangle$$

At TAL level

$$\exists\alpha.\langle\forall[].\{\texttt{r1} : \alpha, \texttt{r2} : int, \texttt{r3} : \exists\beta.\langle\forall[].\{\texttt{r1} : \beta, \texttt{r2} : bool\}^1, \beta^1\rangle\}^1, \alpha^1\rangle$$

Added Syntax of STAL

$$
\begin{array}{lll}
\textit{types} & \tau ::= & ... \mid ns \\
\textit{stack types} & \sigma ::= & \rho \mid nil \mid \tau :: \sigma \\
\textit{type assignments} & \Delta ::= & ... \mid \rho, \Delta \\
\textit{register assignments} & \Gamma ::= & \{r_1 : \tau_1, ..., r_n : \tau_n, \texttt{sp} : \sigma\} \\
\textit{word values} & w ::= & ... \mid w[\sigma] \mid ns \\
\textit{small values} & v ::= & ... \mid v[\sigma] \\
\textit{register files} & R ::= & \{r_1 \mapsto w_1, ..., r_n \mapsto w_n, \texttt{sp} \mapsto S\} \\
\textit{stacks} & S ::= & nil \mid w :: S \\
\textit{instructions} & \iota ::= & ... \mid \texttt{salloc}\, n \mid \texttt{sfree}\, n \mid \\
 & & \texttt{sld}\, r_d,\ \texttt{sp}(i) \mid \texttt{sst}\, \texttt{sp}(i),\ r_s
\end{array}
$$

		$\%\sigma$
salloc	n	$\%ns :: ns :: ... :: ns :: \sigma$
sst	sp(0), r_1	$\%\tau_1 :: ns :: ... :: ns :: \sigma$
:		
sst	sp($n-1$), r_n	$\%\tau_1 :: \tau_2 :: ... :: \tau_n :: \sigma$
code for e		
sld	r_1, sp(0)	$\%\tau_1 :: \tau_2 :: ... :: \tau_n :: \sigma$
:		
sld	r_n, sp($n-1$)	$\%\tau_1 :: \tau_2 :: ... :: \tau_n :: \sigma$
sfree	n	$\%\sigma$

Why stack type variable?

$$\forall[].\{\texttt{r1} : int, \texttt{sp} : \sigma, \texttt{ra} : \forall[].\{\texttt{r1} : bool, \texttt{sp} : \sigma\}\}$$

The problems are

- possibility to touch the values inside the stack.
- no way to use function polymorphically in that specific type.

$$\forall[\rho].\{\texttt{r1} : int, \texttt{sp} : \rho, \texttt{ra} : \forall[].\{\texttt{r1} : bool, \texttt{sp} : \rho\}\}$$

```
(H, {sp ↦ nil}, I)  where

H = l_fact:
        code[ρ]{r1 : ⟨⟩, r2 : int, sp : ρ, ra : τ_ρ}.
          bneq r2,l_nonzero[ρ]           % if n = 0 continue
          mov r1,1                       % result is 1
          jmp ra                         % return
    l_nonzero:
        code[ρ]{r1 : ⟨⟩, r2 : int, sp : ρ, ra : τ_ρ}.
          sub r3,r2,1                    % n − 1
          salloc 2                       % save n and return address to stack
          sst sp(0),r2
          sst sp(1),ra
          mov r2,r3                      % recursive call fact(n − 1)
          mov ra,l_cont[ρ]
          jmp l_fact[int::τ_ρ::ρ]
    l_cont:
        code[ρ]{r1 : int, sp : int::τ_ρ::ρ}.
          sld r2,sp(0)                   % restore n and return address
          sld ra,sp(1)
          sfree 2
          mul r1,r2,r1                   % result is n × fact(n − 1)
          jmp ra                         % return
    l_halt:
        code[]{r1 : int, sp : nil}.
          halt [int]

and I =   malloc r1[]                    % environment
          mov r2,6                       % argument
          mov ra,l_halt                  % return address for initial call
          jmp l_fact[nil]

and τ_ρ = ∀[].{r1 : int, sp : ρ}
```

Fig. 4. STAL Factorial Example