FROM SYSTEM F TO TYPED ASSEMBLY LANGUAGE

Greg Morrisett, David Walker, Karl Crary & Neal Glew
TOPLAS 1999

Presentation by:
Drew Zagieboylo/Matthew Milano

TYPED ASSEMBLY LANGUAGE

types	τ, σ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
heap types	Ψ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
register file types	Γ	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$

TYPED ASSEMBLY LANGUAGE

types	τ, σ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
heap types	Ψ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
register file types	Γ	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$

registers	r	$::=$	$\mathtt{r1} \mid \mathtt{r2} \mid \mathtt{r3} \mid \cdots$
word values	w	$::=$	$\ell \mid i \mid ?\tau \mid w[\tau] \mid \mathsf{pack}\ [\tau, w]\ \mathsf{as}\ \tau'$
small values	v	$::=$	$r \mid w \mid v[\tau] \mid \mathsf{pack}\ [\tau, v]\ \mathsf{as}\ \tau'$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle \mid \mathsf{code}[\vec{\alpha}]\Gamma.I$
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	ι	$::=$	$\mathtt{add}\, r_d, r_s, v \mid \mathtt{bnz}\, r, v \mid \mathtt{ld}\, r_d, r_s[i] \mid \mathtt{malloc}\, r_d[\vec{\tau}] \mid \mathtt{mov}\, r_d, v \mid$ $\mathtt{mul}\, r_d, r_s, v \mid \mathtt{st}\, r_d[i], r_s \mid \mathtt{sub}\, r_d, r_s, v \mid \mathtt{unpack}[\alpha, r_d], v$
instruction sequences	I	$::=$	$\iota; I \mid \mathtt{jmp}\, v \mid \mathtt{halt}[\tau]$
programs	P	$::=$	(H, R, I)

TYPED ASSEMBLY LANGUAGE

$$
\begin{array}{lrcl}
\textit{types} & \tau, \sigma & ::= & \alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau \\
\textit{initialization flags} & \varphi & ::= & 0 \mid 1 \\
\textit{heap types} & \Psi & ::= & \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\} \\
\textit{register file types} & \Gamma & ::= & \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \\
\textit{type contexts} & \Delta & ::= & \alpha_1, \ldots, \alpha_n
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{registers} & r & ::= & \mathtt{r1} \mid \mathtt{r2} \mid \mathtt{r3} \mid \cdots \\
\textit{word values} & w & ::= & \ell \mid i \mid ?\tau \mid w[\tau] \mid \mathsf{pack}\ [\tau, w]\ \mathsf{as}\ \tau' \\
\textit{small values} & v & ::= & r \mid w \mid v[\tau] \mid \mathsf{pack}\ [\tau, v]\ \mathsf{as}\ \tau' \\
\textit{heap values} & h & ::= & \langle w_1, \ldots, w_n \rangle \mid \mathsf{code}[\vec{\alpha}]\Gamma.I \\
\textit{heaps} & H & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\textit{register files} & R & ::= & \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{instructions} & \iota & ::= & \mathtt{add}\, r_d, r_s, v \mid \mathtt{bnz}\, r, v \mid \mathtt{ld}\, r_d, r_s[i] \mid \mathtt{malloc}\, r_d[\vec{\tau}] \mid \mathtt{mov}\, r_d, v \mid \\
 & & & \mathtt{mul}\, r_d, r_s, v \mid \mathtt{st}\, r_d[i], r_s \mid \mathtt{sub}\, r_d, r_s, v \mid \mathtt{unpack}[\alpha, r_d], v \\
\textit{instruction sequences} & I & ::= & \iota; I \mid \mathtt{jmp}\, v \mid \mathtt{halt}[\tau] \\
\textit{programs} & P & ::= & (H, R, I)
\end{array}
$$

TYPED ASSEMBLY LANGUAGE

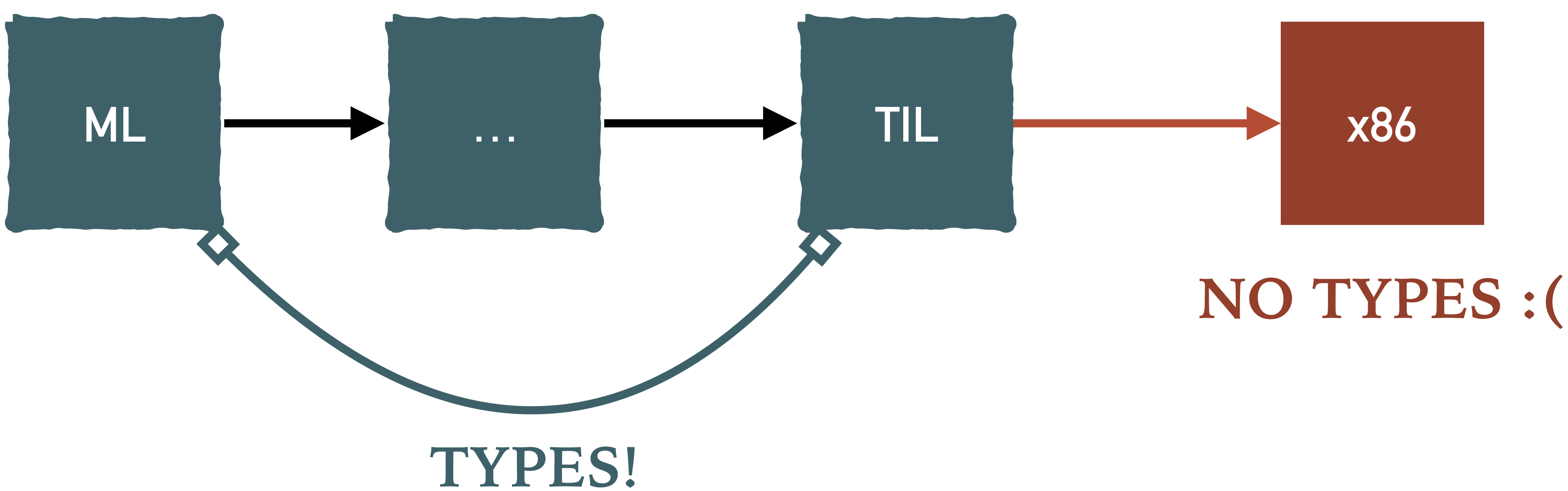
```
l_main:
  code[]{}.                          % entry point
    mov r1,6
    jmp l_fact
l_fact:
  code[]{r1:int}.                    % compute factorial of r1
    mov r2,r1                        % set up for loop
    mov r1,1
    jmp l_loop
l_loop:
  code[]{r1:int,r2:int}.             % r1: the product so far,
                                     % r2: the next number to be multiplied
    bnz r2,l_nonzero                 % branch if not zero
    halt[int]                        % halt with result in r1
l_nonzero:
  code[]{r1:int,r2:int}.
    mul r1,r1,r2                     % multiply next number
    sub r2,r2,1                      % decrement the counter
    jmp l_loop
```

WHY DO WE WANT TAL?

TYPE SYSTEMS ALL THE WAY!!

TYPED INTERMEDIATE LANGUAGES

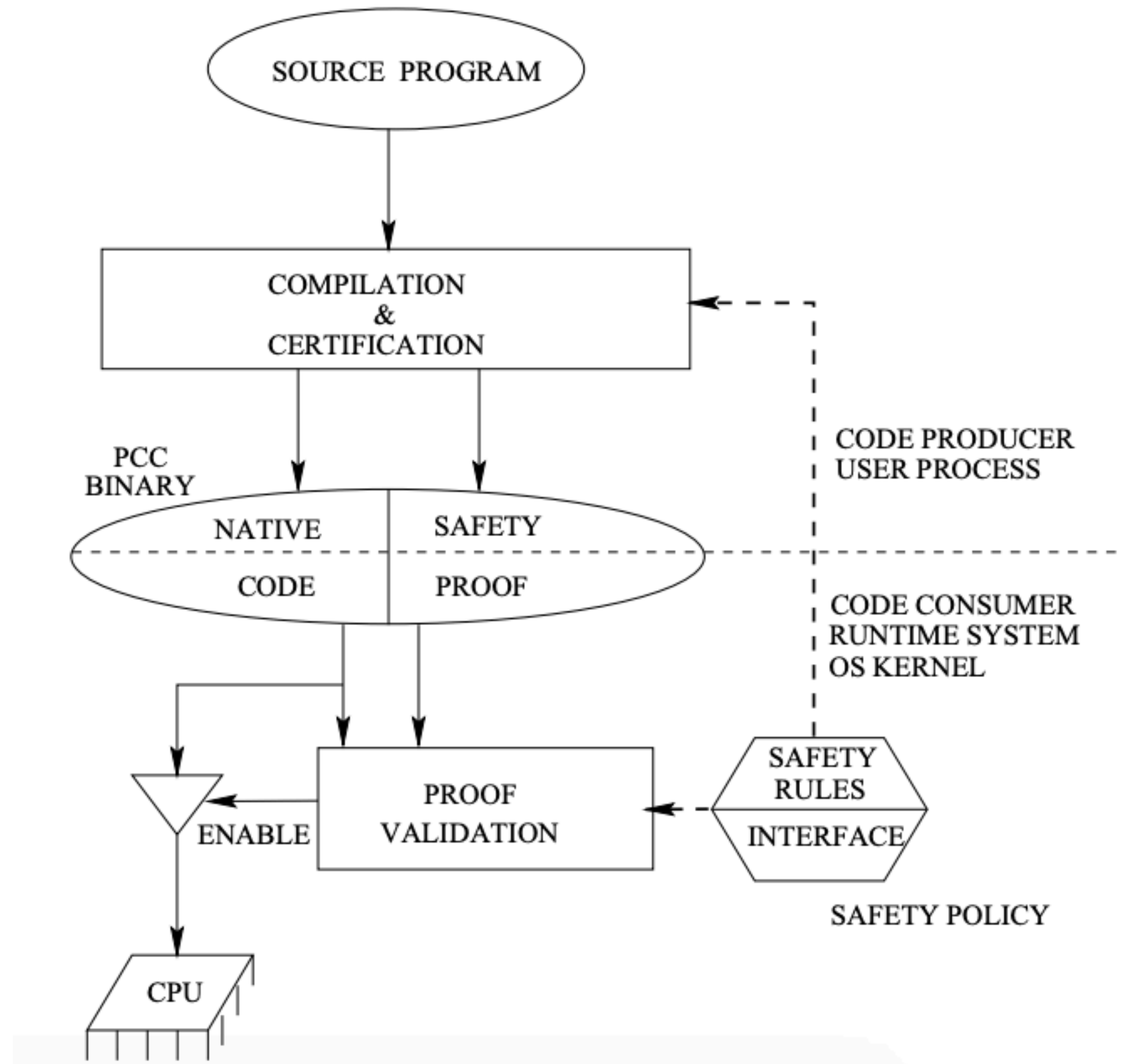
- TIL
  - Throughout the 90's (and today!)
  - Benefits of Types (efficiency + soundness)
  - Target Language is *Untyped*



HOW TO GUARANTEE SAFETY W/ UNTYPED AND UNTRUSTED CODE?

PROOF-CARRYING CODE

- George Necula (POPL '97)
- Compiler Produces:
    1. Program
    2. Proof
- First-Order Predicate Logic Based
- Difficult to Build Compilers



TYPED ASSEMBLY LANGUAGE

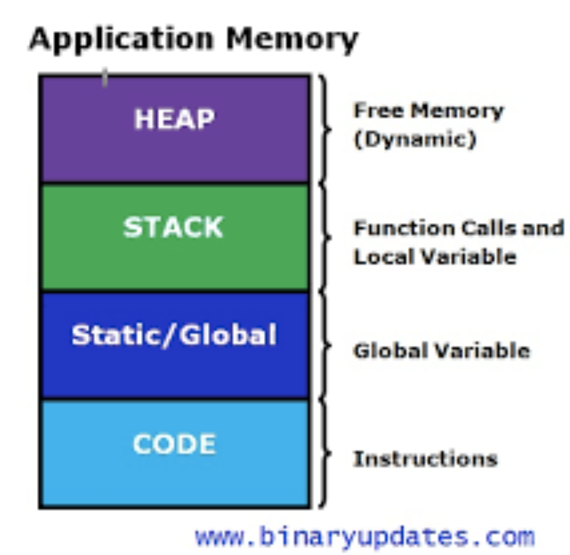
- Extend benefits of *types* **all the way** to the target
- Types as implementation of *Proof-Carrying Code*

```
l_main:
  code[]{}.                          % entry point
    mov r1,6
    jmp l_fact
l_fact:
  code[]{r1:int}.                    % compute factorial of r1
    mov r2,r1                        % set up for loop
    mov r1,1
    jmp l_loop
l_loop:
  code[]{r1:int,r2:int}.             % r1: the product so far,
                                     % r2: the next number to be multiplied
    bnz r2,l_nonzero                 % branch if not zero
    halt[int]                        % halt with result in r1
l_nonzero:
  code[]{r1:int,r2:int}.
    mul r1,r1,r2                     % multiply next number
    sub r2,r2,1                      % decrement the counter
    jmp l_loop
```

TYPED ASSEMBLY LANGUAGE – FEATURES

- **RISC**-style language
- Types:
  - Code types
  - Pointer Types
  - Existential Type Constructor
- **Security:**
  - No pointer forging!
  - Control Flow Integrity
- **Other:**
  - Memory Allocation

```
l_fact:
  code[]{r1:int}.
```



SYSTEM F TO TAL

- Show that TAL is *expressive*

$$\lambda^{\mathrm{F}} \xrightarrow{\substack{\text{CPS}\\ \text{conversion}}} \lambda^{\mathrm{K}} \xrightarrow{\substack{\text{Closure}\\ \text{conversion}}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\substack{\text{Code}\\ \text{generation}}} \mathrm{TAL}$$

SYSTEM F TO TAL

- CPS Conversion

$$\lambda^{\mathrm{F}} \xrightarrow{\text{CPS conversion}} \lambda^{\mathrm{K}} \xrightarrow{\text{Closure conversion}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\text{Code generation}} \mathrm{TAL}$$

CPS TRANSLATION

- Continuation Passing Style
  - Translate to near-linear series of let bindings & calls
  - Removes function call stack

Abstraction Translation

$$\mathcal{K}_{\text{exp}}[\![(\mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e)^{\tau}]\!]k \stackrel{\text{def}}{=} k((\mathsf{fix}\, x(x_1{:}\mathcal{K}[\![\tau_1]\!], c{:}\mathcal{K}_{\text{cont}}[\![\tau_2]\!]).\mathcal{K}_{\text{exp}}[\![e]\!]c^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}[\![\tau]\!]})$$

Application Translation

$$\begin{aligned}\mathcal{K}_{\text{exp}}[\![(u_1^{\tau_1}\ u_2^{\tau_2})^{\tau}]\!]k \stackrel{\text{def}}{=}\ & \mathcal{K}_{\text{exp}}[\![u_1^{\tau_1}]\!](\lambda x_1{:}\mathcal{K}[\![\tau_1]\!].\\ & \quad \mathcal{K}_{\text{exp}}[\![u_2^{\tau_2}]\!](\lambda x_2{:}\mathcal{K}[\![\tau_2]\!].\\ & \qquad x_1^{\mathcal{K}[\![\tau_1]\!]}(x_2^{\mathcal{K}[\![\tau_2]\!]}, k))^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau_1]\!]}\end{aligned}$$

SYSTEM F TO λ_K

- Continuation Passing Style

$$\lambda_F \quad (\mathit{fix}\, f(n : int) : int\, .\, \mathit{if}0\,(n, 1, n \times f(n-1)))\, 6$$

$$\downarrow$$

$$\begin{aligned}
\lambda_K \quad & (\mathit{fix}\, f(n : int, k : (int) \to void)\, . \\
& \quad \mathit{if}0(n, k(1), \\
& \qquad let\, x = n - 1 in \\
& \qquad f(x, \lambda(y : int)\, .\, let\, z = n \times y\, in\, k(z)))) \\
& (6, \lambda(n : int)\, .\, halt[int]n)
\end{aligned}$$

SYSTEM F TO TAL

- Closure Conversion

$$\lambda^{\mathrm{F}} \xrightarrow{\text{CPS conversion}} \lambda^{\mathrm{K}} \xrightarrow{\text{Closure conversion}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\text{Code generation}} \mathrm{TAL}$$

POLYMORPHIC CLOSURE CONVERSION

- Generate Explicit Closures
- Implements Encapsulation
- New Syntax
  - Existential Types

$$\tau, \sigma ::= \ldots \mid \exists \alpha \,.\, \tau$$

  - Packing/Unpacking

$$u ::= \ldots \mid v[\tau] \mid pack[\tau_1, v] \; as \; \tau_2$$
$$d ::= \ldots \mid [\alpha, x] = \; unpack \; v$$

- Uses *Type Erasure**
  - Function bodies type-check w/o environment type info
  - **Pack** is a no-op at runtime

λ_K TO λ_C

- Polymorphic Closure Conversion

Function Type Translation

$$\mathcal{C}\llbracket\forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n) \to void\rrbracket = \exists\beta.\langle\forall[\vec{\alpha}].(\beta, \mathcal{C}\llbracket\tau_1\rrbracket,\ldots,\mathcal{C}\llbracket\tau_n\rrbracket) \to void, \beta\rangle$$

Application Translation

$$\begin{array}{lcl}
\mathcal{C}_{\text{exp}}\llbracket u^\tau[\sigma_1,\ldots,\sigma_m](v_1,\ldots,v_n)\rrbracket & \stackrel{\text{def}}{=} & \mathsf{let}\ [\gamma, z] = \mathsf{unpack}\ \mathcal{C}_{\text{val}}\llbracket u^\tau\rrbracket\ \mathsf{in} \\
& & \mathsf{let}\ z_{code} = \pi_1(z^{\langle\tau_{code},\gamma\rangle})\ \mathsf{in} \\
& & \mathsf{let}\ z_{env} = \pi_2(z^{\langle\tau_{code},\gamma\rangle})\ \mathsf{in} \\
& & (z_{code}{}^{\tau_{code}}[\mathcal{C}\llbracket\sigma_1\rrbracket,\ldots,\mathcal{C}\llbracket\sigma_m\rrbracket]) \\
& & \qquad (z_{env}{}^{\gamma}, \mathcal{C}_{\text{val}}\llbracket v_1\rrbracket,\ldots,\mathcal{C}_{\text{val}}\llbracket v_n\rrbracket) \\
& & \text{where} \\
& & \mathcal{C}\llbracket\tau\rrbracket = \exists\gamma.\langle\tau_{code},\gamma\rangle
\end{array}$$

SYSTEM F TO TAL

- Hoisting

$$\lambda^{\mathrm{F}} \xrightarrow{\substack{\text{CPS}\\ \text{conversion}}} \lambda^{\mathrm{K}} \xrightarrow{\substack{\text{Closure}\\ \text{conversion}}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\substack{\text{Code}\\ \text{generation}}} \mathrm{TAL}$$

HOISTING

- Separating Code Definition & Program
- Much like real memory layout
  - Closures make this easy!
  - Bind **fix** statements to variables, pointing to code

Syntax changes:

$$\begin{array}{llcl} \textit{values} & u & ::= & \text{delete } \mathsf{fix}\, x(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ \textit{heap values} & h & ::= & \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ \textit{programs} & P & ::= & \mathsf{letrec}\, x_1 \mapsto h_1, \ldots, x_n \mapsto h_n \;\mathsf{in}\; e \end{array}$$

The typing rule for fix is replaced by a heap value rule for **code**:

$$\frac{\vec{\alpha} \vdash_{\mathrm{H}} \tau_i \quad \vec{\alpha}; (\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \vdash_{\mathrm{H}} e}{\Gamma \vdash_{\mathrm{H}} \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \mathit{void} \;\; \mathrm{hval}} \; (x_1, \ldots, x_n \notin \Gamma)$$

New typing rule:

$$\frac{\emptyset \vdash_{\mathrm{H}} \tau_i \quad x_1{:}\tau_1, \ldots, x_n : \tau_n \vdash_{\mathrm{H}} h_i : \tau_i \;\; \mathrm{hval} \quad \emptyset; x_1{:}\tau_1, \ldots, x_n : \tau_n \vdash_{\mathrm{H}} e}{\vdash_{\mathrm{H}} \mathsf{letrec}\, x_1 \mapsto h_1, \ldots, x_n \mapsto h_n \;\mathsf{in}\; e} \; (x_i \neq x_j \text{ for } i \neq j)$$

λ_K

➤ P

➤ F

```
letrec f_code    ↦ (* main factorial code block *)
                   code[](env:⟨⟩, n:int, k:τ_k).
                       if0(n, (* true branch: continue with 1 *)
                              let [β, k_unpack] = unpack k in
                              let k_code = π1(k_unpack) in
                              let k_env  = π2(k_unpack) in
                              k_code(k_env, 1),
                              (* false branch: recurse with n − 1 *)
                              let x = n − 1 in
                              f_code(env, x, pack [⟨int, τ_k⟩, ⟨cont_code, ⟨n, k⟩⟩] as τ_k))
       cont_code ↦ (* code block for continuation after factorial computation *)
                   code[](env:⟨int, τ_k⟩, y:int).
                       (* open the environment *)
                       let n = π1(env) in
                       let k = π2(env) in
                       (* compute n! into z *)
                       let z = n × y in
                       (* continue with z *)
                       let [β, k_unpack] = unpack k in
                       let k_code = π1(k_unpack) in
                       let k_env  = π2(k_unpack) in
                       k_code(k_env, z)
       halt_code ↦ (* code block for top-level continuation *)
                   code[](env:⟨⟩, n:int). halt[int]n
in
       f_code(⟨⟩, 6, pack [⟨⟩, ⟨halt_code, ⟨⟩⟩] as τ_k)
```

where τ_k is $\exists\alpha.\langle(\alpha, int) \to void, \alpha\rangle$

SYSTEM F TO TAL

- Memory Allocation

$$\lambda^{\mathrm{F}} \xrightarrow{\text{CPS conversion}} \lambda^{\mathrm{K}} \xrightarrow{\text{Closure conversion}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\text{Code generation}} \mathrm{TAL}$$

ALLOCATION

- Assembly language doesn't have Tuples!
- Need to allocate memory for tuples (and initialize!)

$$A[[\langle \tau_1, \ldots, \tau_n \rangle]] \triangleq \langle A[[\tau_1]]^1, \ldots, A[[\tau_n]]^1 \rangle$$

  - x = (v_1, v_2)

$$\begin{aligned}
\mathsf{let}\ \ & x_1 : \langle int^0, int^0 \rangle = \mathsf{malloc}[int, int] \\
& x_2 : \langle int^1, int^0 \rangle = x_1[1] \leftarrow v_1 \\
& x\ \ : \langle int^1, int^1 \rangle = x_2[2] \leftarrow v_2 \\
& \vdots
\end{aligned}$$

ALLOCATION

λ_H

(* false branch: recurse with $n - 1$ *)
let $x = n - 1$ in
$f_{code}(env, x, \textsf{pack}\ [\langle int, \tau_k\rangle,\ \langle cont_{code}, \langle n, k\rangle\rangle]\ \textsf{as}\ \tau_k))$

λ_A

let $x = n - 1$ in
let $y_1 = \textsf{malloc}[int, \tau_k]$ in
let $y_2 = y_1[1] \leftarrow n$ in
let $y_3 = y_2[2] \leftarrow k$ in (* $\langle n, k\rangle$ *)
let $y_4 = \textsf{malloc}[(\langle int, \tau_k\rangle, int) \rightarrow void, \langle int, \tau_k\rangle]$ in
let $y_5 = y_4[1] \leftarrow cont_{code}$ in
let $y_6 = y_5[2] \leftarrow y_3$ in (* $\langle cont_{code}, \langle n, k\rangle\rangle$ *)
$f_{code}(env, x, \textsf{pack}\ [\langle int, \tau_k\rangle, y_6]\ \textsf{as}\ \tau_k))$

SYSTEM F TO TAL

- Code Generation

$$\lambda^{\mathrm{F}} \xrightarrow[\text{conversion}]{\text{CPS}} \lambda^{\mathrm{K}} \xrightarrow[\text{conversion}]{\text{Closure}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow[\text{generation}]{\text{Code}} \mathrm{TAL}$$

SYSTEM F TO TAL

- Code Generation
  - Mostly direct translation to assembly
  - Function types annotate registers

$$\mathcal{T}[\![\forall[\vec{\alpha}](\tau_1, \cdots, \tau_n) \to void]\!] \stackrel{\text{def}}{=} \forall[\vec{\alpha}]\{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!], \ldots, \mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\}$$

  - **unpack** is just a **mov** instruction w/ type erasure
  - **malloc** is abstract

TAL IMPLEMENTATION

- TALx86 : IA32 ISA
- Variation from Paper:
  - Other data types (arrays, floats, etc.)
  - Not CPS -> Uses Explicit Stack
  - Implements **malloc** and **unpack** instructions
  - Modules with Type Interfaces
- Some optimizations
  - Register-sized objects vs. “large objects”
  - Cross-module optimization

CONCLUSIONS

- System F -> TAL
  - We *can* have security and expressivity
  - Utilizes many PL techniques
    - Type-directed Compilation
  - Formalism omits many optimizations (other work)
- Future Work & Impact
  - Cyclone (low level, typed language)
    - (and then Rust)

THANK YOU!

POLYMORPHIC CC – TWICE EXAMPLE

λ^{F} source:

$$twice = \Lambda\alpha.\, \lambda f{:}\alpha \to \alpha.\, \lambda x{:}\alpha.\, f(fx)$$

λ^{K} source:

$$
\begin{array}{l}
twice = \\
\quad \lambda[\alpha](f{:}\tau_f,\, c{:}(\tau_f) \to void). \\
\quad\quad \mathsf{let}\ twicef = \\
\quad\quad\quad \lambda(x{:}\alpha,\, c'{:}(\alpha) \to void). \\
\quad\quad\quad\quad \mathsf{let}\ oncef = \lambda(z{:}\alpha).\, f(z, c')\ \mathsf{in} \\
\quad\quad\quad\quad f(x, oncef) \\
\quad\quad \mathsf{in} \\
\quad\quad\quad c[\,](twicef)
\end{array}
$$

where $\tau_f = (\alpha, (\alpha) \to void) \to void$

POLYMORPHIC CC

λ^{F} source:

$$twice = \Lambda\alpha.\, \lambda f{:}\alpha \to \alpha.\, \lambda x{:}\alpha.\, f(f x)$$

λ^{K} source:

$$\begin{array}{l}
twice = \\
\quad \lambda[\alpha](f{:}\tau_f,\, c{:}(\tau_f) \to void). \\
\qquad \textsf{let } twicef = \\
\qquad\quad \lambda(x{:}\alpha,\, c'{:}(\alpha) \to void). \\
\qquad\qquad \textsf{let } oncef = \lambda(z{:}\alpha).\, f(z, c') \textsf{ in} \\
\qquad\qquad f(x, oncef) \\
\qquad \textsf{in} \\
\qquad\quad c[\,](twicef)
\end{array}$$

where $\tau_f = (\alpha, (\alpha) \to void) \to void$

λ^{H} translation:

$$\begin{array}{l}
\textsf{letrec}\quad twice_{code}[\alpha](env{:}\langle\rangle, f{:}\tau_f,\, c{:}\exists\rho_3.\langle(\rho_3, \tau_f) \to void, \rho_3\rangle). \\
\qquad\qquad \textsf{let } twicef = \textsf{pack } [\langle\tau_f\rangle, \langle twicef_{code}[\alpha], \langle f\rangle\rangle] \textsf{ as } \tau_f \textsf{ in} \\
\qquad\qquad \textsf{let } [\rho_3, c_{unpack}] = \textsf{unpack } c \textsf{ in} \\
\qquad\qquad \textsf{let } c_{code} = \pi_1(c_{unpack}) \textsf{ in} \\
\qquad\qquad \textsf{let } c_{env} = \pi_2(c_{unpack}) \textsf{ in} \\
\qquad\qquad\quad c_{code}(c_{env}, twicef) \\
\qquad twicef_{code}[\alpha](env{:}\langle\tau_f\rangle, x{:}\alpha,\, c'{:}\tau_{\alpha c}). \\
\qquad\qquad \textsf{let } f = \pi_1(env) \textsf{ in} \\
\qquad\qquad \textsf{let } oncef = \textsf{pack } [\langle\tau_f, \tau_{\alpha c}\rangle, \langle oncef_{code}[\alpha], \langle f, c'\rangle\rangle] \textsf{ as } \tau_{\alpha c} \\
\qquad\qquad \textsf{let } [\rho_1, f_{unpack}] = \textsf{unpack } f \textsf{ in} \\
\qquad\qquad \textsf{let } f_{code} = \pi_1(f_{unpack}) \textsf{ in} \\
\qquad\qquad \textsf{let } f_{env} = \pi_2(f_{unpack}) \textsf{ in} \\
\qquad\qquad\quad f_{code}(f_{env}, x, oncef) \\
\qquad oncef_{code}[\alpha](env : \langle\tau_f, \tau_{\alpha c}\rangle, z : \alpha). \\
\qquad\qquad \textsf{let } f = \pi_1(env) \textsf{ in} \\
\qquad\qquad \textsf{let } c' = \pi_2(env) \textsf{ in} \\
\qquad\qquad \textsf{let } [\rho_1, f_{unpack}] = \textsf{unpack } f \textsf{ in} \\
\qquad\qquad \textsf{let } f_{code} = \pi_1(f_{unpack}) \textsf{ in} \\
\qquad\qquad \textsf{let } f_{env} = \pi_2(f_{unpack}) \textsf{ in} \\
\qquad\qquad\quad f_{code}(f_{env}, z, c') \\
\textsf{in} \cdots
\end{array}$$

$$\begin{array}{lrcl}
\text{where} & \tau_f & = & \exists\rho_1.\langle(\rho_1, \alpha, \tau_{\alpha c}) \to void, \rho_1\rangle \\
& \tau_{\alpha c} & = & \exists\rho_2.\langle(\rho_2, \alpha) \to void, \rho_2\rangle
\end{array}$$