From System F to Typed Assembly Language

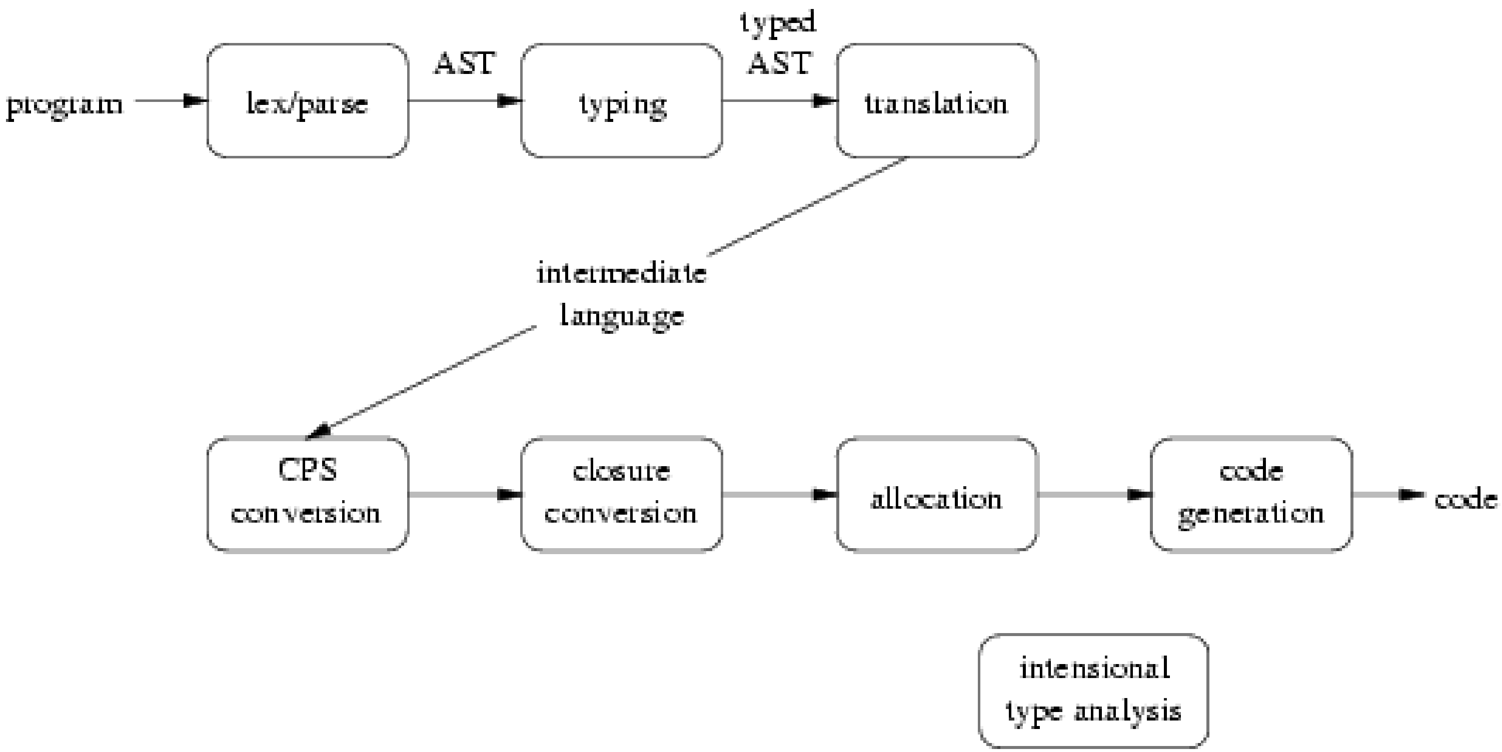
G. Morrisett, D. Walker, K. Crary, and N. Glew
TOPLAS 1999

Presented by Oukseh Lee

Research On Program Analysis System, KAIST

Sep. 18, 2000

Compilation



λ^F

types	τ, σ	$::=$	$\alpha \mid \texttt{int} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \langle\vec{\tau}\rangle$
annotated terms	e	$::=$	u^τ
terms	u	$::=$	$x \mid i \mid \texttt{fix}\, f\, \lambda(x{:}\, \tau).e{:}\, \tau \mid e\; e \mid \Lambda\alpha.e \mid e[\tau]$
		$\mid$	$\langle\vec{e}\rangle \mid \pi_i(e) \mid e\; p\; e \mid \texttt{if0}(e, e, e)$
primitives	p	$::=$	$+ \mid - \mid \times$

$$(\texttt{fix}\, f\, \lambda(n{:}\, \texttt{int}).\texttt{if0}(n, 1, n \times f(n-1)){:}\, \texttt{int})\; 6$$

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

- Based on System F of Girard and Reynolds.

CPS Coversion

- Purpose:
  - Eliminate the need for a control stack.
  - Names all intermediate computations.
- How: Pass continuation (remained work) to a callee.

$$
\begin{array}{l}
(\texttt{fix}\ f\ \lambda(n{:}\ \texttt{int}, k{:}\ \texttt{int} \to \texttt{unit}). \\
\quad \texttt{if0}(n, k\ 1, \\
\qquad \texttt{let}\ x \texttt{=}\ n - 1\ \texttt{in} \\
\qquad\quad f(x, \lambda y{:}\ \texttt{int}.\texttt{let}\ z\ =\ n \times y\ \texttt{in}\ k\ z))) \\
(6, \lambda n{:}\ \texttt{int}.\texttt{halt}[\texttt{int}]\ n)
\end{array}
$$

- Based on Harper and Lillibridge's.

Closure Conversion

- Purpose: Remove free variables.
- How: Pass free variables' values.

$$\begin{array}{ll} \texttt{let} & y = 1 \\ & f = \lambda x.(\cdots y \cdots) \\ \texttt{in} & f\,2 \end{array} \quad \Rightarrow \quad \begin{array}{ll} \texttt{let} & y = 1 \\ & f = \lambda(x,y).(\cdots y \cdots) \\ \texttt{in} & f\,(2,y) \end{array}$$

- Hoisting: Lift up all functions to top-level.

$$\begin{array}{l} \texttt{letrec}\, f \mapsto \lambda(x,y).(\cdots y \cdots) \\ \texttt{in let}\, y = 1\, \texttt{in}\, f\,(2,y) \end{array}$$

- Based on Minamide et al.

```
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 $\exists\alpha.\langle(\alpha, int) \rightarrow void, \alpha\rangle$

Fig. 8. Factorial in λ^{H}.

Allocation

- Purpose: Explicitly allocate memory to non-primitive values.
- How: Use explicit memory allocator.

$$\texttt{let}\, x = \langle v_1, v_2 \rangle\, \texttt{in} \cdots$$

$$\Downarrow$$

$$\begin{array}{ll} \texttt{let} & x_1 = \texttt{malloc}[\texttt{int}, \texttt{int}] \\ & x_2 = x_1[1] \leftarrow v_1 \\ & x \;\;= x_2[2] \leftarrow v_2 \\ \texttt{in} & \cdots \end{array}$$

$$
\begin{array}{lll}
\textsf{letrec}\ f_{code} & \mapsto & \texttt{(*}\ \text{main factorial code block}\ \texttt{*)} \\
 & & \textsf{code}[\,](env{:}\langle\rangle, n{:}int, k{:}\tau_k). \\
 & & \quad \textsf{if0}(n,\ \texttt{(*}\ \text{true branch: continue with 1}\ \texttt{*)} \\
 & & \qquad \textsf{let}\ [\beta, k_{unpack}] = \textsf{unpack}\ k\ \textsf{in} \\
 & & \qquad \textsf{let}\ k_{code} = \pi_1(k_{unpack})\ \textsf{in} \\
 & & \qquad \textsf{let}\ k_{env} = \pi_2(k_{unpack})\ \textsf{in} \\
 & & \qquad k_{code}(k_{env}, 1), \\
 & & \qquad \texttt{(*}\ \text{false branch: recurse with}\ n-1\ \texttt{*)} \\
 & & \qquad \textsf{let}\ x = n - 1\ \textsf{in} \\
 & & \qquad \textsf{let}\ y_1 = \textsf{malloc}[int, \tau_k]\ \textsf{in} \\
 & & \qquad \textsf{let}\ y_2 = y_1[1] \leftarrow n\ \textsf{in} \\
 & & \qquad \textsf{let}\ y_3 = y_2[2] \leftarrow k\ \textsf{in} \qquad \texttt{(*}\ \langle n, k\rangle\ \texttt{*)} \\
 & & \qquad \textsf{let}\ y_4 = \textsf{malloc}[(\langle int, \tau_k\rangle, int) \to void, \langle int, \tau_k\rangle]\ \textsf{in} \\
 & & \qquad \textsf{let}\ y_5 = y_4[1] \leftarrow cont_{code}\ \textsf{in} \\
 & & \qquad \textsf{let}\ y_6 = y_5[2] \leftarrow y_3\ \textsf{in} \qquad \texttt{(*}\ \langle cont_{code}, \langle n, k\rangle\rangle\ \texttt{*)} \\
 & & \qquad f_{code}(env, x, \textsf{pack}\ [\langle int, \tau_k\rangle, y_6]\ \textsf{as}\ \tau_k)) \\
\quad cont_{code} & \mapsto & \texttt{(*}\ \text{code block for continuation after factorial computation}\ \texttt{*)} \\
 & & \textsf{code}[\,](env{:}\langle int, \tau_k\rangle, y{:}int). \\
 & & \quad \texttt{(*}\ \text{open the environment}\ \texttt{*)} \\
 & & \quad \textsf{let}\ n = \pi_1(env)\ \textsf{in} \\
 & & \quad \textsf{let}\ k = \pi_2(env)\ \textsf{in} \\
 & & \quad \texttt{(*}\ \text{continue with}\ n \times y\ \texttt{*)} \\
 & & \quad \textsf{let}\ z = n \times y\ \textsf{in} \\
 & & \quad \textsf{let}\ [\beta, k_{unpack}] = \textsf{unpack}\ k\ \textsf{in} \\
 & & \quad \textsf{let}\ k_{code} = \pi_1(k_{unpack})\ \textsf{in} \\
 & & \quad \textsf{let}\ k_{env} = \pi_2(k_{unpack})\ \textsf{in} \\
 & & \quad k_{code}(k_{env}, z) \\
\quad halt_{code} & \mapsto & \texttt{(*}\ \text{code block for top-level continuation}\ \texttt{*)} \\
 & & \textsf{code}[\,](env{:}\langle\rangle, n{:}int).\ \textsf{halt}[int]n \\
\textsf{in} & & \\
\quad \textsf{let}\ y_7 = \textsf{malloc}[\,]\ \textsf{in} \qquad \texttt{(*}\ \langle\rangle\ \texttt{*)} & & \\
\quad \textsf{let}\ y_8 = \textsf{malloc}[\,]\ \textsf{in} \qquad \texttt{(*}\ \langle\rangle\ \texttt{*)} & & \\
\quad \textsf{let}\ y_9 = \textsf{malloc}[(\langle\rangle, int) \to void, \langle\rangle]\ \textsf{in} & & \\
\quad \textsf{let}\ y_{10} = y_9[1] \leftarrow halt_{code}\ \textsf{in} & & \\
\quad \textsf{let}\ y_{11} = y_{10}[2] \leftarrow y_8\ \textsf{in} \qquad \texttt{(*}\ \langle halt_{code}, \langle\rangle\rangle\ \texttt{*)} & & \\
\quad f_{code}(y_7, 6, \textsf{pack}\ [\langle\rangle, y_{11}]\ \textsf{as}\ \tau_k) & &
\end{array}
$$

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

Fig. 12. Factorial in λ^{A}.

TAL

- High-level abstract assembly language that has type information.

```
(H, {}, I) where
 H =
   l_fact:
       code[]{r1:⟨⟩,r2:int,r3:τ_k}.
         bnz r2,l_nonzero
         unpack [α,r3],r3                 % zero branch: call k (in r3) with 1
         ld r4,r3[0]                      % project k code
         ld r1,r3[1]                      % project k environment
         mov r2,1
         jmp r4                           % jump with {r1 = env, r2 = 1}
   l_nonzero:
       code[]{r1:⟨⟩,r2:int,r3:τ_k}.
         sub r4,r2,1                      % n − 1
         malloc r5[int, τ_k]              % create environment for cont in r5
         st r5[0],r2                      % store n into environment
         st r5[1],r3                      % store k into environment
         malloc r3[∀[].{r1:⟨int^1, τ_k^1⟩,r2:int}, ⟨int^1, τ_k^1⟩] % create cont closure in r3
         mov r2,l_cont
         st r3[0],r2                      % store cont code
         st r3[1],r5                      % store environment ⟨n, k⟩
         mov r2,r4                        % arg := n − 1
         mov r3,pack [⟨int^1, τ_k^1⟩,r3] as τ_k % abstract the type of the environment
         jmp l_fact                       % call fact(env, n − 1, l_cont)
   l_cont:
       code[]{r1:⟨int^1, τ_k^1⟩,r2:int}.  % r2 contains (n − 1)!
         ld r3,r1[0]                      % retrieve n
         ld r4,r1[1]                      % retrieve k
         mul r2,r3,r2                     % n × (n − 1)!
         unpack [α,r4],r4                 % unpack k
         ld r3,r4[0]                      % project k code
         ld r1,r4[1]                      % project k environment
         jmp r3                           % jump to k with {r1 = env, r2 = n!}
   l_halt:
       code[]{r1:⟨⟩,r2:int}.
         mov r1,r2
         halt[int]                        % halt with result in r1

 and I =
         malloc r1[]                      % create an empty environment (⟨⟩)
         malloc r2[]                      % create another empty environment
         malloc r3[∀[].{r1:⟨⟩,r2:int}, ⟨⟩] % create halt closure in r3
         mov r4,l_halt
         st r3[0],r4                      % store halt code
         st r3[1],r2                      % store halt environment ⟨⟩
         mov r2,6                         % load argument (6)
         mov r3,pack [⟨⟩,r3] as τ_k       % abstract the type of the environment
         jmp l_fact                       % call fact(⟨⟩, 6, l_halt)

 and τ_k = ∃α.⟨∀[].{r1:α,r2:int}^1, α^1⟩
```

Fig. 21. Typed assembly code for factorial.

```
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
```

Fig. 1. A TAL program that computes 6 factorial.

Type Preservation

Corrollary (Compiler Type Correctness). *If* $\vdash_F e : \tau$ *then* $\vdash_{\mathrm{TAL}} (\mathcal{T}_{\mathrm{prog}} \circ \mathcal{A}_{\mathrm{prog}} \circ \mathcal{H}_{\mathrm{prog}} \circ \mathcal{C}_{\mathrm{prog}} \circ \mathcal{K}_{\mathrm{prog}})[\![e]\!]$.