Implementing Typed Intermediate Languages

Zhong Shao, Christopher League, and Stefan Monnier
ICFP 1998

Sukyoung Ryu

October 9, 2000

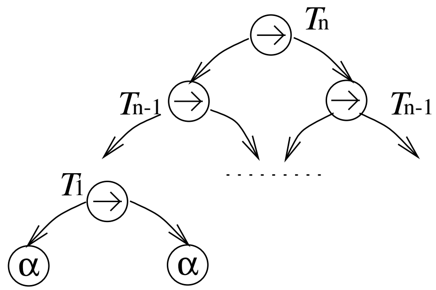
Motivation

- Type information is very hard to represent and manipulate efficiently.

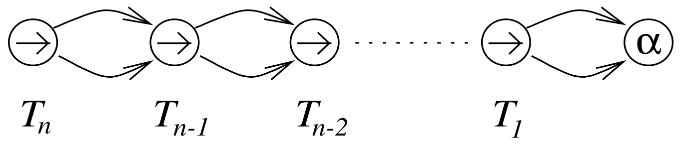
```
fun f x = x
fun g y = ( ... (((f f) f) f) ... f) y
```

$\bar{T}_n$ $\qquad\qquad\qquad$ $\bar{T}_1$

  – a tree-like structure



  – a linear-sized dag



$\boxed{\text{Core-FLINT}}$

$$
\begin{array}{llll}
(\mathit{kind}) & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \\
(\mathit{tycon}) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \to \mu_2 \mid \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\
(\mathit{type}) & \sigma & ::= & T(\mu) \mid \sigma_1 \to \sigma_2 \mid \forall t :: \kappa.\sigma \\
(\mathit{term}) & e & ::= & i \mid x \mid \lambda x : \sigma.e \mid @x_1 x_2 \mid \Lambda t :: \kappa.e \mid x[\mu] \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ f = \Lambda t :: \Omega.\lambda x : T(t).x \\
\ \texttt{in}\ \ldots\ f[\texttt{Int}]\ \ldots
\end{array}
$$

- Based on F_w with the term language written in A-normal form
- Goal: how to handle complex FLINT types efficiently

de Bruijn indices

- A reference to a formal parameter is the number of λ's between the reference and the λ which binds the parameter.

$$\lambda x_1\!:\!\sigma_1.\lambda x_2\!:\!\sigma_2.\lambda x_3\!:\!\sigma_3.@x_2x_3 \quad \leadsto \quad \lambda\sigma_1.\lambda\sigma_2.\lambda\sigma_3.@\#2\ \#1$$

- α-equivalent constructors always have the same representation.

$$\begin{array}{lcl} \lambda t_1\!::\!\Omega.t_1 \rightarrow t_1 & \leadsto & \lambda\Omega.\#1 \rightarrow \#1 \\ \lambda t_2\!::\!\Omega.t_2 \rightarrow t_2 & \leadsto & \lambda\Omega.\#1 \rightarrow \#1 \end{array}$$

Suspension-based λ Encoding

$$
\begin{array}{lll}
(\mathit{constructor}) & \mu & ::= \ \#n \mid \texttt{Int} \mid \mu_1 \to \mu_2 \mid \lambda\kappa.\mu \mid \mu_1[\mu_2] \mid \texttt{Env}(\mu, \rho) \\
(\mathit{substitution}) & \rho & ::= \ (i, j, \mathit{env}) \\
(\mathit{environment}) & \mathit{env} & ::= \ \mathit{nil} \mid j :: \mathit{env} \mid (j, \mu) :: \mathit{env}
\end{array}
$$

- Type reductions are done *lazily*.
- a suspension term $\texttt{Env}(\mu, \rho)$
  the intermediate result of some unevaluated type applications $\rho(\mu)$
- (i, j, env): i - the current embedding level of bound type variables
  j - its new embedding level
  env - the actual bindings of all i bound variables
- $(\lambda\kappa.\mu_1)[\mu_2] \implies \texttt{Env}(\mu_1, (1, 0, (0, \mu_2) :: \mathit{nil}))$

Type reductions under suspension-based λ encoding

$$
\begin{array}{ll}
(r1) & (\lambda\kappa.\mu_1)[\mu_2] \implies \mathtt{Env}(\mu_1, (1, 0, (0, \mu_2) :: nil)) \\
(r2) & \mathtt{Env}(\mu, (0, 0, nil)) \implies \mu \\
(r3) & \mathtt{Env}(\mu, (i, j, env)) \implies \mu \quad \text{if } \mu \text{ is closed, i.e., it has no free type variables} \\
(r4) & \mathtt{Env}(\#n, (i, j, env)) \implies \#(n - i + j) \;\; \text{if } n > i \\
(r5) & \mathtt{Env}(\#n, (i, j, env)) \implies \#(j - j') \\
 & \qquad\qquad\qquad\qquad \text{if } n \leq i \text{ and the } n\text{-th element of } env \text{ is } j' \\
(r6) & \mathtt{Env}(\#n, (i, j, env)) \implies \mathtt{Env}(\mu, (0, j - j', nil)) \\
 & \qquad\qquad\qquad\qquad \text{if } n \leq i \text{ and the } n\text{-th element of } env \text{ is } (j', \mu) \\
(r7) & \mathtt{Env}(\mathtt{Int}, \rho) \implies \mathtt{Int} \\
(r8) & \mathtt{Env}(\mu_1 \to \mu_2, \rho) \implies \mathtt{Env}(\mu_1, \rho) \to \mathtt{Env}(\mu_2, \rho) \\
(r9) & \mathtt{Env}(\mu_1[\mu_2], \rho) \implies (\mathtt{Env}(\mu_1, \rho))[\mathtt{Env}(\mu_2, \rho)] \\
(r10) & \mathtt{Env}(\lambda\kappa.\mu, (i, j, env)) \implies \lambda\kappa.\mathtt{Env}(\mu, (i + 1, j + 1, j :: env)) \\
(r11) & \mathtt{Env}(\mathtt{Env}(\mu, (i, j, env)), (0, j', nil)) \implies \mathtt{Env}(\mu, (i, j + j', env))
\end{array}
$$

Examples

$$
\begin{aligned}
(\lambda\kappa_1.\lambda\kappa_2.\lambda\kappa_3.\#5)[\texttt{Int}] &\Longrightarrow \texttt{Env}(\lambda\kappa_2.\lambda\kappa_3.\#5, (1, 0, (0, \texttt{Int}) :: nil)) \\
&\Longrightarrow \lambda\kappa_2.\texttt{Env}(\lambda\kappa_3.\#5, (2, 1, 0 :: (0, \texttt{Int}) :: nil)) \\
&\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\texttt{Env}(\#5, (3, 2, 1 :: 0 :: (0, \texttt{Int}) :: nil)) \\
&\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\#4
\end{aligned}
$$

$$
\begin{aligned}
(\lambda\kappa_1.\lambda\kappa_2.\lambda\kappa_3.\#2)[\texttt{Int}] &\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\texttt{Env}(\#2, (3, 2, 1 :: 0 :: (0, \texttt{Int}) :: nil)) \\
&\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\#2
\end{aligned}
$$

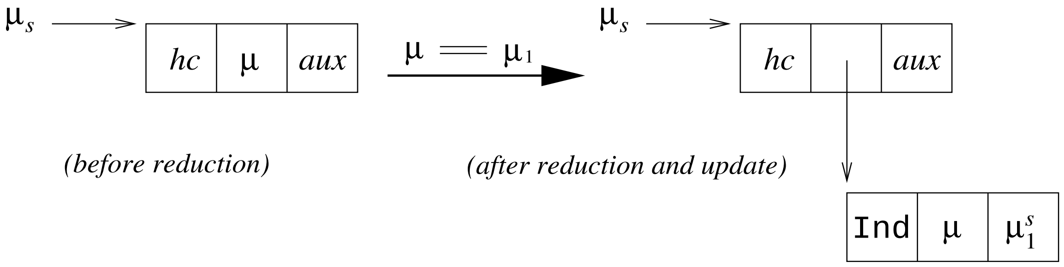
$$
\begin{aligned}
(\lambda\kappa_1.\lambda\kappa_2.\lambda\kappa_3.\#3)[\texttt{Int}] &\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\texttt{Env}(\#3, (3, 2, 1 :: 0 :: (0, \texttt{Int}) :: nil)) \\
&\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\texttt{Env}(\texttt{Int}, (0, 2, nil)) \\
&\Longrightarrow \lambda\kappa_2.\lambda\kappa_3.\texttt{Int}
\end{aligned}
$$

Hash-consing and Memoization

- hash-consing and memoization

$$\begin{array}{llcl} (\textit{hash-cell}) & \mu^s & ::= & \texttt{Ref}(\textit{hashcode},\ \mu,\ \textit{auxinfo}) \\ (\textit{constructor}) & \mu & ::= & \#n \mid \texttt{Int} \mid \mu_1^s \to \mu_2^s \mid \lambda\kappa^s.\mu^s \mid \mu_1^s[\mu_2^s] \\ & & \mid & \texttt{Env}(\mu^s, \rho^s) \mid \texttt{Ind}(\mu_1, \mu_2^s) \end{array}$$

- a memoization node $\texttt{Ind}(\mu, \mu_1^s)$



Discussion

- With hash-consing and memoization,

  common operations such as equality tests, normal form tests, and finding out the set of free variables can all be done in constant time.

- With suspension terms,

  type applications are always done on a *by-need* basis, and once it is done, the result will be memoized for future uses.

- The de Bruijn indices complicate some optimizations (e.g., inlining and specialization).