Linear Lambda Calculus Compiler Documentation

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

February 17, 2017

Overview:

We compile the source language, linear lambda calculus programs e

$$\begin{aligned} b &::= + \mid - \mid / \mid * \\ e &::= n \mid x \mid e_1 \; b \; e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\ &\mid \lambda\, x.\, e \mid e_1 \; e_2 \end{aligned}$$

to the target language, stack machine programs ρ

$$\begin{aligned} \mathit{binop} &::= \text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} \\ \mathit{funop} &::= \text{Save\_Function } (\mathit{fun\_id}, k) \mid \text{Form\_Closure } (\mathit{fun\_id}) \mid \text{Apply} \\ \mathit{rollop} &::= \text{Roll } k \mid \text{Unroll } k \\ \mathit{tupleop} &::= \text{Construct\_Tuple } k \mid \text{Deconstruct\_Tuple } k \\ \mathit{instr} &::= \mathit{binop} \mid \mathit{funop} \mid \mathit{rollop} \mid \mathit{tupleop} \mid \text{Push } n \\ \rho &::= \mathit{instr} \text{ list} \end{aligned}$$

which operate on stacks σ and σ_f

$$\begin{aligned} \sigma_f &::= (\mathit{fun\_id}, \rho) \text{ list} \\ \mathit{tup} &::= \text{Tuple } (v_1, \ldots, v_k) \\ v &::= \text{Int } n \mid \mathit{tup} \mid \text{Closure } (\rho, \mathit{tup}) \\ \sigma &::= v \text{ list} \end{aligned}$$

Defunctionalization:

Given a lambda-calculus program p, defunctionalization produces a first-order language. That is, functions are no longer considered to be values. Instead, $l ::= (\lambda x.\ e)$ is represented as $C_l(v_1, \ldots, v_n)$, where C_l uniquely identifies the function l, and $v_1, \ldots, v_n$ are the values of the variables $x_1, \ldots, x_n$ which are free in l.

We define the translation from lambda-calculus program to first-order program.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.\ e]\!] &= C_{\lambda x.\ e}(x_1, \ldots, x_n),\ \text{where } x_1, \ldots, x_n \text{ are free in } \lambda x.\ e \\
[\![e_1\ b\ e_2]\!] &= [\![e_1]\!]\ b\ [\![e_2]\!] \\
[\![\text{let } x = e_1 \text{ in } e_2]\!] &= [\![(\lambda x.\ e_2)\ e_1]\!] \\
[\![e_1\ e_2]\!] &= \text{apply}([\![e_1]\!], [\![e_2]\!])
\end{aligned}
$$

Thus a lambda calculus program p is translated to a first-order program as such:

```
let rec apply_defunc (f, arg) =
    match f with
        | C_{λx. e}(x_1,...,x_n) -> let x = arg in [[e]]
        | ... in
        [[p]]
```

Analogue to Defunctionalization:

The values Closure (fun_id, Tuple $(v_1, \ldots, v_k)$) on stack σ are analogous to the constructors $C_{fun\_id}(v_1, \ldots, v_k)$.

The function stack σ_f stores the programs $\hat{\rho}$ corresponding to each case of the dispatch function 'apply_defunc'. At the beginning of a program ρ, there will be a series of Save_Function instructions to initialize σ_f.

Executing and Reversing a Program:

The rules we present below show the reversibility for each instruction using the following tuple:

$$(\rho, \sigma, \sigma_f, \sigma_h \in \text{int list}, \rho_h).$$

Binomial arithmetic operations:

$$((\text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div as } b) :: \rho, \text{Int } n_1 :: \text{Int } n_2 :: \sigma, \sigma_f, \sigma_h, b :: \rho_h)$$
$$\Longleftrightarrow (\rho, n_1 \star n_2 :: \sigma, \sigma_f, n_1 :: \sigma_h, b :: \rho_h)$$

Function operations:

$$(\text{Save\_Function } (fun\_id, k) \text{ as } sf :: \hat{\rho} \text{ @ } \rho, \sigma, \sigma_f, \sigma_h, \rho_h), \text{where } |\hat{\rho}| = k$$
$$\Longleftrightarrow (\rho, \sigma, (fun\_id, \hat{\rho}) :: \sigma_f, \sigma_h, sf :: \rho_h)$$

$$(\text{Form\_Closure } (fun\_id) \text{ as } fc :: \rho, \text{Tuple } (v_1, \dots, v_k) \text{ as } tup :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\text{where List.assoc } fun\_id \ \sigma_f = \hat{\rho}$$
$$\Longleftrightarrow (\rho, \text{Closure } (\hat{\rho}, tup) :: \sigma, \sigma_f, \sigma_h, fc :: \rho_h)$$

$$(\text{Apply} :: \rho, \text{Closure } (\hat{\rho}, \text{ Tuple } (v_1, \dots, v_k) \text{ as } tup) :: \text{arg} :: \sigma, \sigma_f, \sigma_h, \rho_h),$$
$$\Longleftrightarrow (\hat{\rho} \text{ @ } \rho, tup :: \text{arg} :: \sigma, \sigma_f, |\hat{\rho}| :: \sigma_h, fc :: \rho_h)$$

Roll, Tuple, and Integer Pushing operations:

$$(\text{Roll } k :: \rho, v_1 :: v_2 :: \cdots :: v_k :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, v_k :: v_1 :: \cdots :: v_{k-1} :: \sigma, \sigma_f, \sigma_h, \text{Roll } k :: \rho_h)$$

$$(\text{Unroll } k :: \rho, v_1 :: v_2 :: \cdots :: v_k :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, v_2 :: \cdots :: v_k :: v_1 :: \sigma, \sigma_f, \sigma_h, \text{Unroll } k :: \rho_h)$$

$$(\text{Compose\_Tuple } k :: \rho, v_1 :: \cdots :: v_k :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, \text{Tuple}(v_1, \dots, v_k) :: \sigma, \sigma_f, \sigma_h, \text{Compose\_Tuple } k :: \rho_h)$$

$$(\text{Decompose\_Tuple } k :: \rho, \text{Tuple}(v_1, \dots, v_k) :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, v_1 :: \cdots :: v_k :: \sigma, \sigma_f, \sigma_h, \text{Decompose\_Tuple } k :: \rho_h)$$

$$(\text{Push } n :: \rho, \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, \text{Int } n :: \sigma, \sigma_f, \sigma_h, \text{Push } n :: \rho_h)$$

Reversability of Apply:

All the operations change the size of the stack σ.

$$
\begin{aligned}
\text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} &\rightarrow -1 \\
\text{Save\_Function}\ \_ &\rightarrow +0 \\
\text{Form\_Closure}\ \_ &\rightarrow +0 \\
\text{Apply} &\rightarrow -1 \\
\text{Roll}\ \_ \mid \text{Unroll}\ \_ &\rightarrow +0 \\
\text{Compose\_Tuple}\ k &\rightarrow -k+1 \\
\text{Decompose\_Tuple}\ k &\rightarrow +k-1 \\
\text{Push}\ \_ &\rightarrow +1
\end{aligned}
$$

Therefore, we can classify the instructions into grow ops and shrink ops.

$$
\begin{aligned}
i_g &::= \text{Push}\ \_ \mid \text{Compose\_Tuple}\ 0 \mid \text{Decompose\_Tuple}\ k > 1 \\
\rho_g &::= i_g\ \text{list} \\
i_s &::= \text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} \\
&\quad \mid \text{Save\_Function}\ \_ \mid \text{Form\_Closure}\ \_ \mid \text{Apply} \\
&\quad \mid \text{Roll}\ \_ \mid \text{Unroll}\ \_ \mid \text{Compose\_Tuple}\ k > 0 \mid \text{Decompose\_Tuple}\ k \leq 1 \\
\rho_s &::= i_s\ \text{list}
\end{aligned}
$$

Let us define the following property.

$$
\textbf{Grow-Shrink Property}: \\
\text{gs}\ (\rho \text{ where } |\rho| > 0) \ = \ \exists \rho_g,\ \rho_s.\ \rho = \rho_g \mathbin{@} \rho_s
$$

The translation we produce has important properties that may allow Apply to be injective. For every program ρ produced by the translation, gs (ρ) holds. Additionally, for every closure Closure $(n, \hat{\rho})$, gs $(\hat{\rho})$ holds.