Linear Lambda Calculus Compiler Documentation

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

February 21, 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

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

which operate on stacks σ and σ_f

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

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:

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

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

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

Roll, Tuple, and Integer Pushing operations:

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

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

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

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

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

Reversability of Apply:

Define $\vec{\rho} = (\hat{\rho}, \rho)$ and $\vec{\sigma} = (\text{Tuple}(v_1, \ldots, v_n), \text{arg}, \sigma)$.

Well-Formed Closure Property :

$$\begin{aligned} \text{wf}(\vec{\rho}, \vec{\sigma}) \;\triangleq\; & (\text{Apply::}\rho,\ \text{Closure } (\hat{\rho},\ \text{Tuple } (v_1, \ldots, v_k) \text{ as } tup)\text{::arg} :: \sigma, \ldots) \\ & \Longrightarrow (\hat{\rho} \mathbin{@} \rho, tup :: \text{arg} :: \sigma, \ldots) \Longrightarrow^* (\rho, result :: \sigma, \ldots) \end{aligned}$$

All the operations change the size of the stack σ. We define the function δ which gives the change in the size of σ after executing an instruction.

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

Let us define the following property.

Grow-Shrink Property :

$$\text{gs } (\rho \text{ where } |\rho| > 0) \;\triangleq\; \exists \rho_g,\ \rho_s.\ \rho = \rho_g \mathbin{@} \rho_s$$

Recall that to reverse Apply, we must split the program such that we restore the instructions in the original closure. Currently, we must store the length of the instructions in the closure body to reverse Apply. We define this split as follows.

$$\text{splits } (\vec{\rho}, \vec{\sigma}) \;\triangleq\; \rho' = \hat{\rho} \mathbin{@} \rho \wedge :\ \texttt{wf } (\vec{\rho}, \vec{\sigma})$$

If the following proposition holds, then Apply would be injective.

$$\text{InjApply} : \quad \forall \rho', \vec{\sigma}.\ \exists\ \vec{\rho^a}.\ \text{splits } (\vec{\rho^a}, \sigma) \wedge \exists\ \vec{\rho^b}.\ \text{splits } (\vec{\rho^b}, \vec{\sigma}) \Longrightarrow \vec{\rho^a} = \vec{\rho^b}$$

More definitions:

Let $\Phi(\rho) = \text{List.fold\_left (fun sum instr. acc} + \delta \text{ (instr)) } 0\ \rho$. Note that for $\hat{\rho}$ the body of a well formed closure,

$$\Phi(\hat{\rho}) = -1.$$

Furthermore, for any any $\hat{\rho} = \hat{\rho}_g \mathbin{@} \hat{\rho}_s$,

$$\Phi(\hat{\rho}_g) = \alpha \geq 0, \qquad \Phi(\hat{\rho}_s) = -\alpha - 1.$$

Proof of InjApply:

Let $\vec{\sigma}$. Suppose, for a contradiction, there exist $\vec{\rho^a} \neq \vec{\rho^b}$ such that splits $(\vec{\rho^a}, \vec{\sigma})$ and splits $(\vec{\rho^b}, \vec{\sigma})$. Assuming the grow shrink property holds for $\hat{\rho}^a$ and $\hat{\rho}^b$,

$$\hat{\rho}^a = \hat{\rho}^a_g \mathbin{@} \hat{\rho}^a_s, \qquad \hat{\rho}^b = \hat{\rho}^b_g \mathbin{@} \hat{\rho}^b_s.$$

TODO $\hat{\rho}^a_g = \hat{\rho}^b_g$.

Assume $\Phi(\hat{\rho}^a_g) = \alpha$. Then $\Phi(\hat{\rho}^a_s) = \Phi(\hat{\rho}^b_s) = -\alpha - 1$.

Then $\hat{\rho}^b_s = \hat{\rho}^a_s \mathbin{@} \rho^*$, and $\Phi(\rho^*) = 0$ (more clearly, the instructions in ρ^* do not change the size of the stack).

TODO guarantee that our compiler will not produce closure body programs with end with instructions that do not change the size of the stack.