Linear Lambda Calculus Compiler Definitions

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

September 16, 2016

Source Language

The following grammar defines an expression e, where x is a variable and n is an integer.

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

Stack Machine Instruction Set:

An instruction can be any of the following:

$$\begin{aligned} i \; &::= \; \text{Push } n \mid \text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} \\ &\mid \text{Roll } n \mid \text{Unroll } n \mid \text{Form\_Closure } (n_v, n_\rho) \mid \text{MultiApply } n \end{aligned}$$

Here we define machine program p, stack s, and stack value v:

$$\begin{aligned} \rho \; &::= \; i \text{ list} \\ v \; &::= \; \text{Int } n \mid \text{Closure } (n, \rho) \\ \sigma \; &::= \; v \text{ list} \end{aligned}$$

Evaluating and Reversing Target Language:

Each tuple is of the form $(\rho, \sigma, \sigma_h \in \text{ int list}, \rho_h)$.

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

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

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

$$(\text{Unroll } n :: \rho, v_1 :: v_2 :: \cdots :: v_n :: \sigma, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, v_2 :: \cdots :: v_n :: v_1 :: \sigma, \sigma_h, \text{Unroll } n :: \rho_h)$$

$$(\text{Form\_Closure } (n_v, n_\rho) :: i_1 :: \cdots :: i_{n_\rho} :: \rho, \sigma, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, \text{Closure } (n_v, i_1 :: \cdots :: i_{n_\rho}) :: \sigma, \sigma_h, \text{Form\_Closure } (n_v, n_\rho) :: \rho_h)$$

$$(\text{MultiApply } n :: \rho, \text{Closure}(n, \hat{\rho}) :: v_1 :: \cdots :: v_n :: \sigma, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\hat{\rho} \mathbin{@} \rho, v_1 :: \cdots :: v_n @ \ \sigma, |\hat{\rho}| :: \sigma_h, \text{MultiApply } n :: \rho_h)$$

Properties of compiled programs:

Every closure has the following property. After executing the body of the closure, consumes its arguments. Let $\vec{\sigma} = (n, \sigma, \hat{\sigma} \text{ where} |\hat{\sigma}| = n)$ and $\vec{\rho} = (\rho, \hat{\rho})$.

Well-Formed Closure Property :

$$\begin{aligned} \text{wf}(\vec{\sigma}, \vec{\rho}) &\stackrel{\text{def}}{=} (\text{MultiApply } n :: \rho, \text{Closure}(n, \hat{\rho}) :: \hat{\sigma} :: \sigma, \ldots) \\ &\Longrightarrow (\hat{\rho} \text{ @ } \rho, \hat{\sigma}\text{@ } \sigma, \ldots) \Longrightarrow^* (\rho, v :: \sigma, \ldots) \end{aligned}$$

Notice that Form_Closure does not take any values from the stack because the body of a λ-expression contains no free variables. Therefore, we can classify the instructions into grow ops and shrink ops.

$$\begin{aligned} i_g &::= \text{Push } n \mid \text{Form\_Closure } (n_v, n_\rho) \\ \rho_g &::= i_g \text{ list} \\ i_s &::= \text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} \\ &\quad \mid \text{Roll } n \mid \text{Unroll } n \mid \text{MultiApply } n \\ \rho_s &::= i_s \text{ list} \end{aligned}$$

Every program produced by our compiler has the following property.

Grow-Shrink Property :

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