Linear Lambda Calculus Compiler Documentation

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

August 1, 2016

Overview:

Given a linear lambda calculus expression, the compiler translates the expression to a lambda-lifted expression and then to stack machine instructions. Once the set of instructions have been produced, the expression can be evaluated to a value v, which can either be an integer or a function closure.

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

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

The compiler assumes that expressions are well-typed. In expressions of the form $e_1\ b\ e_2$, both e_1 and e_2 must evaluate to integers. Expressions of the form $e_1\ e_2$ are assumed to be valid function applications. Since the language is linear, each variable is assumed to be used exactly once.

Lambda Lifting Expressions:

The following grammar defines a lambda-lifted program p.

$$
\begin{aligned}
p &::= \text{let } x\ =\ \lambda\ x_1, \ldots, x_n.\ e \text{ in } p \mid e \\
b &::= + \mid - \mid / \mid * \\
e &::= n \mid x \mid e_1\ b\ e_2 \mid e_1\ e_2
\end{aligned}
$$

The expressions e are λ-free, meaning they do not contain λ-abstractions or let-expressions. The important property that lambda lifting gives is that λ-expressions are closed. Therefore, the closures they produce have empty environments.

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 s_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 a Program:

Suppose we have executed some program $\rho \in$ `program`, which produces some value v on the stack. Given a history of which instructions have been executed $\rho_h \in$ `program`, where the head of the list is the most recent instruction, we can revert the stack to its original state before executing ρ according to the following rules.

Some instructions constitute non-injective functions $\sigma \rightarrow \sigma$. Therefore, we must introduce a history tape to maintain information about about the stack state when these instructions are executed. We call this history tape σ_h. The rules we present below show the reversibility for each instruction using the following tuple:

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

Evaluation and Reversal Rules:

$$\begin{aligned}&(\text{Push } n :: \rho, \sigma, \sigma_h, \rho_h)\\ \iff &(\rho, \text{Int } n :: \sigma, \sigma_h, \text{Push } n :: \rho_h)\end{aligned}$$

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

$$\begin{aligned}&(\text{Roll } n :: \rho, v_1 :: v_2 :: \cdots :: v_n :: \sigma, \sigma_h, \rho_h)\\ \iff &(\rho, v_n :: v_1 :: \cdots :: v_{n-1} :: \sigma, \sigma_h, \text{Roll } n :: \rho_h)\end{aligned}$$

$$\begin{aligned}&(\text{Unroll } n :: \rho, v_1 :: v_2 :: \cdots :: v_n :: \sigma, \sigma_h, \rho_h)\\ \iff &(\rho, v_2 :: \cdots :: v_n :: v_1 :: \sigma, \sigma_h, \text{Unroll } n :: \rho_h)\end{aligned}$$

$$\begin{aligned}&(\text{Form\_Closure } (n_v, n_\rho) :: i_1 :: \cdots :: i_{n_\rho} :: \rho, \sigma, \sigma_h, \rho_h)\\ \iff &(\rho, \text{Closure } (n_v, i_1 :: \cdots :: i_{n_\rho}) :: \sigma, \sigma_h, \text{Form\_Closure } (n_v, n_\rho) :: \rho_h)\end{aligned}$$

$$\begin{aligned}&(\text{MultiApply } n :: \rho, \text{Closure}(n, \hat{\rho}) :: v_1 :: \cdots :: v_n :: \sigma, \sigma_h, \rho_h)\\ \iff &(\hat{\rho} \mathbin{@} \rho, v_1 :: \cdots :: v_n @ \sigma, |\hat{\rho}| :: \sigma_h, \text{MultiApply } n :: \rho_h)\end{aligned}$$

Additionally, every closure has the following property. Essentially, every closure, after executing the body of the closure, consumes its arguments.

$$\begin{aligned}\vec{\sigma} &= (n, \sigma, \hat{\sigma} \text{ where} |\hat{\sigma}| = n)\\ \vec{\rho} &= (\rho, \hat{\rho})\end{aligned}$$

Well-Formed Closure Property :

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

Translation to Stack Machine Instructions:

Recall that we are working with lifted lambda expressions. All the operations change the size of the stack predictably. Here is how each operation affects the size of the stack:

$$\begin{aligned} \text{Push } i &\rightarrow +1 \\ (\text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div}) &\rightarrow -1 \\ \text{Roll } \_ \mid \text{Unroll } \_ &\rightarrow 0 \\ \text{Form\_Closure } \_ &\rightarrow +1 \\ \text{MultiApply } n &\rightarrow -n+1 \end{aligned}$$

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.

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

Let us define the following property.

Grow-Shrink Property :

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

The translation we produce has important properties that may allow MultiApply 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.

The translation is produced using many careful roll and unroll operations. It is defined in compiler.ml.

Limitations of Lambda-Lifting:

Unfortunately, not all sample programs provided work properly (sample_programs/app5_not_working.lam). Though the closures in the lambda-lifted expressions have no free variables (and thus empty environments), this does not guarantee that additional closures formed as the result of partial application have empty environments. Consider the following illustrative example of an

expression in the target lambda-lifted language.

$$\begin{array}{l} \text{let } f \;=\; (\lambda\ x\ y.\ x + y) \text{ in} \\ \text{let } g \;=\; (\lambda\ f.\ f\ 1) \text{ in} \\ g\ f\ 1 \end{array}$$

After evaluating $(g\ f)$, we produce the closure $\{(\lambda\ x\ y.\ x + y), [x = 1]\}$, which we do not have the capacity to represent in our stack-machine. Allowing such closures may cause the grow-shrink properties outlined in the previous section to no longer hold.

Reversability of MultiApply:

The `MultiApply` instruction may be injective because of the linear constraints in our language. Recall that we define the Well-Formed Closure Property and the Grow-Shrink Property. Recall that to reverse MultiApply, we must split the program and the stack such that we restore the original closure. We define this split by the following. The vectors used in these definitions where defined with the Well-Formed Closure Property.

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

If we could prove the following proposition, then `MultiApply` would be injective.

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

In the test cases we have tried, InjApply seems to hold, and here is the pseudocode that finds the appropriate split.

$$\begin{array}{l} \text{let } rev\ n\ (\hat{\rho}, \rho) = \\ \quad \text{if } n = 1 \wedge |\hat{\rho}| > 0 \\ \quad \text{then } (\hat{\rho}, \rho) \text{ else match } \rho \text{ with} \\ \quad\quad \mid \text{Push } \_ \text{ as } p :: t \rightarrow rev\ (n+1)\ (p :: \hat{\rho}, t) \\ \quad\quad \mid (\text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div}) \text{ as } b :: t \rightarrow rev\ (n-1)\ (b :: \hat{\rho}, t) \\ \quad\quad \mid (\text{Unroll } \_ \mid \text{Roll } \_) \text{ as } r :: t \rightarrow rev\ n\ (r :: \hat{\rho}, t) \\ \quad\quad \mid \text{MultiApply } m \text{ as } ma :: t \rightarrow rev\ (n-m)\ (ma :: \hat{\rho}, t) \end{array}$$

There are cases missing from this match, but Form_Closure is not expected in the body of a closure. If the $n = 1$ condition does not hold and $\rho = []$, then reversing MultiApply has failed. Note that we require every closure to have at least one instruction. Otherwise a valid split of ρ' would trivially be $\rho' = [] \mathbin{@} \rho'$.

To reverse MultiApply n given the program ρ', we call $rev\ n\ ([], \rho')$ which provides the split $\rho' = \hat{\rho_g}\ @\ \hat{\rho_s}\ @\ \rho$, where $\hat{\rho_g}\ @\ \hat{\rho_s}$ is the body of the closure we aim to restore (recall that this closure body satisfies the Grow-Shrink Property).

Let $n_0 = n$. Let $\{n_m\}_m$ be the sequence of n through the recursive calls, where n_m is the size of the local stack (of the closure) after executing the first m instructions of ρ'. Note that each instruction of ρ_g increases n by 1. Therefore, $n_{|\rho_g|} = n + |\rho_g|$. Because our closure is well-formed, after executing $\rho_g\ @\ \rho_s$, all arguments to the closure will be consumed to produce a result value. Therefore, $n_{|\rho_g\ @\ \rho_s|} = 1$. This is by no means a proof, but hopefully the explanation provides intuition to the argument that InjApply holds.

Future Aims:

We need to prove that the translation from lifted lambda expression to machine operations is sound and that this translation satisfies the Grow-Shrink Property.

Because of the limitations of lifted-lambda expressions, we need to translate our expressions to a target language without partial application.

The injectiveness of MultiApply must be proved rigorously.