Linear Lambda Calculus Compiler: Fall '16

Anna Yesypenko, supervised by Professor Nate Foster

December 20, 2016

Overview

Before we begin, a few quick reminders on the compiler. 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

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

which operate on a stack σ

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

Instructions execute as defined in the documentation. Here, we highlight a few that we will adjust later. The execution of instructions is defined on the tuple $(\rho, \sigma, \sigma_h \in \text{int list}, \rho_h)$, where ρ_h is a program history tape that records instructions executed. Each instruction is reversible.

$$\begin{aligned} &(\text{Form\_Closure } (n_i, n_v) :: \hat{\rho} :: \rho, \hat{\sigma} :: \sigma, \sigma_h, \rho_h), \text{where } |\hat{\rho}| = n_i \text{ and } |\hat{\sigma}| = n\_v \\ &\Longleftrightarrow (\text{CL } (\hat{\rho}, \hat{\sigma}) :: \rho, \sigma, \sigma_h, \text{Form\_Closure } (n_o, n_v) :: \rho_h) \end{aligned}$$

$$\begin{aligned} &(\text{Apply} :: \rho, v :: \text{CL } (\hat{\rho}, \hat{\sigma}) :: \sigma, \sigma, \sigma_h, \rho_h) \\ &\Longleftrightarrow (\hat{\rho} \ @ \ \rho, \hat{\sigma} \ @ \ \sigma, |\hat{\rho}| :: |\hat{\sigma}| :: \sigma_h, \text{Apply} :: \rho_h) \end{aligned}$$

We have been attempting to reverse apply without adding information to σ_h. Suppose $l \ ::= (\text{lam } x. \ e)$ corresponds to the stack value $c \ ::= \text{CL}(\hat{\sigma}, \hat{\rho})$, where $\hat{\sigma}$ are the stack values corresponding to free variables of l and $\hat{\rho}$ is the compilation of the function body e. We noted that if l is a well-formed closure and $\hat{\rho}$ has the grow-shrink property (see documentation for definitions), then we need not store $|\hat{\rho}|$ to recover the closure c following an Apply instruction.

Objective CL: Produce programs ρ such that all stack closures $\mathrm{CL}(\hat{\sigma}, \hat{\rho})$ produced during the execution of ρ on stack σ have $|\hat{\rho}| = c$ for some constant c, in addition to closures being well-formed and $\hat{\rho}$ obeying the grow-shrink property.

Motivating Example:

```
let add = λ x. λ y.x + y in
let g   = λ f.f 1 in
g (add 2)
```

The example above is lambda-lifted. That is, the top-level functions add and g are closed. Therefore, all the top-level functions can be represented as stack closures $\mathrm{CL}(\hat{\sigma}, \hat{\rho})$ with $|\hat{\sigma}| = 0$. However, the partially applied function $(add 2)$ has free variable 'x'. Representing such a lambda on the stack would require a closure with $|\hat{\sigma}| = 1$.

Therefore, lambda lifting does not produce programs that satisfy Objective CL.

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 $C(v_1, \ldots, v_n)$, where C uniquely identifies the function, 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 (f, arg) =
  match f with
  | C_{λx. e} -> let x = arg in ⟦e⟧
  | ... in
  ⟦p⟧
```

We defunctionalize the example presented earlier

$$[\![(\lambda\ add.\ (\lambda\ g.\ g\ (add\ 2))\ (\lambda\ f.\ f\ 1))(\lambda\ x.\ (\lambda\ y.\ x + y))]\!].$$

```
let rec apply (f, arg) =
    match f with
    | Cletadd      () -> let add = arg in apply (Cletg (add), Cg ())
    | Cletg     (add) -> let g   = arg in apply (g, apply (add, 2))
    | Cg           () -> let f   = arg in apply (f, 1)
    | Cadd         () -> let x   = arg in Caddacc (x)
    | Caddacc     (x) -> let y   = arg in x + y
    apply (Cletadd (), Cadd ())
```

The above program evaluates as follows:

```
apply (Cletadd        (),               Cadd ())
apply (Cletg (Cadd ()),                   Cg ())
apply (Cg             (), apply (Cadd (), 2))
apply (Cg             (),          Caddacc (2))
apply (Caddacc       (2),                    1)
2 + 1
```

Executing Defunctionalized Programs on Stack Machine:

We introduce the idea of a function stack. First let's adjust our definitions for machine programs and stacks.

$$
\begin{aligned}
\rho &::= \textit{instr} \text{ list} \\
\textit{instr} &::= \ \ldots \mid \text{Form\_Closure } (n_\rho) \mid \text{Apply} \\
&\quad \mid \text{Compose\_Tuple } n \mid \text{Decompose\_Tuple } n \\
&\quad \mid \text{Load} \mid \text{Save @fun} \mid \text{Push @fun} \\
\textit{cl} &::= \text{CL } (\text{@fun}, \rho) \\
\sigma_f &::= \textit{cl} \text{ list} \\
v &::= \text{Int } n \mid \text{Tuple } (v_1, \ldots, v_n) \mid \text{@fun } (v_1, \ldots, v_n) \mid CL(\rho) \\
\sigma &::= v \text{ list}
\end{aligned}
$$

These definitions operate on the tuple $(\rho, \sigma, \sigma_f, \sigma_h \in \text{int list}, \rho_h)$. The instructions omitted operate as they are defined previously in the documentation (without modifying σ_f).

$$(\text{Load} :: \rho, @\text{fun}(v_1, \ldots, v_n) :: \sigma, \text{CL}(@\text{fun}, \hat{\rho}) :: \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, \text{CL}(\hat{\rho}) :: \text{Tuple}(v_1, \ldots, v_n) :: \sigma, \sigma_f, \sigma_h, \text{Load} :: \rho_h)$$

$$(\text{Save } @\text{fun} :: \rho, \text{CL}(\hat{\rho}) :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, \sigma, \text{CL}(@\text{fun}, \hat{\rho}) :: \sigma_f, \sigma_h, \text{Save } @\text{fun} :: \rho_h)$$

$$(\text{Push } @\text{fun} :: \rho, \text{Tuple}(v_1, \ldots, v_n) :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\rho, @\text{fun}(v_1, \ldots, v_n) :: \sigma, \sigma_f, \sigma_h, \text{Push } @\text{fun} :: \rho_h)$$

$$(\text{Form\_Closure } (n_{\hat{\rho}}) :: \hat{\rho} :: \rho, \sigma, \sigma_f, \sigma_h, \rho_h), \text{where } |\hat{\rho}| = n_{\hat{\rho}}$$
$$\Longleftrightarrow (\text{CL } (\hat{\rho}) :: \rho, \sigma, \sigma_f, \sigma_h, \text{Form\_Closure } (n_{\hat{\rho}}) :: \rho_h)$$

$$(\text{Apply} :: \rho, \text{CL } (\hat{\rho}) :: \text{Tuple}(v_1, \ldots, v_n) :: v :: \sigma, \sigma_f, \sigma_h, \rho_h)$$
$$\Longleftrightarrow (\hat{\rho} \ @ \ \rho, \text{Tuple}(v_1, \ldots, v_n) :: v :: \sigma, \sigma_f, |\hat{\rho}| :: \sigma_h, \text{Apply} :: \rho_h)$$
$$\Longrightarrow^* (\rho, \mathit{result} :: \sigma, \ldots)$$

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

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

For the tuple operations, $n \geq 0$, and for $n = 0$, the tuple formed is simply (). The functions annotations @fun appear for clarity. The tags can be removed from the stack machine language. We have omitted an instruction RollF from above that operates on σ_f in the same manner that Roll operates on σ.

Machine instructions corresponding to a defunctionalized program e_{df} begins with a series of Form_Closure and Save instructions that populate σ_f.

Consider the defunctionalization of a simple example

$$[\![\text{let } a = 10 \text{ in } (\lambda x.x + a)\ 5]\!] = [\![(\lambda a.(\lambda x.x + a)\ 5)\ 10]\!].$$

```
let rec apply (f, arg) =
    match f with
    | C1  () -> let a = arg in apply (C2 (a), 5)
    | C2 (a) -> let x = arg in x + a
apply (C1 (), 10)
```

Here are the machine instructions corresponding to this program:
Form_Closure (7); *closure for C1*
Decompose_Tuple 0; Push 5; Unroll 2;
Compose_Tuple 1; Push @2;
Load; Apply;
Save @1;
Form_Closure (2); *closure for C2*
Decompose_Tuple 1; Add;
Save @2;
Push 10; Compose_Tuple 0; Push @1
Load; Apply

Now we execute the above instructions and provide snapshots of the stack $(top)[\,](bottom)$. Assume we have already saved @1 and @2 to σ_f.
Push 10; Compose_Tuple 0; Push @1;
Stack looks like [@1(), 10]
Load; Apply;
Stack looks like [(), 10]
Decompose_Tuple 0; Push 5; Unroll 2;
Compose_Tuple 1; Push @2;
Stack looks like [@2(10), 5]
Load; Apply;
Stack looks like [(10), 5]
Decompose_Tuple 1; Add
Stack looks like [15]

Reversibility of Apply

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

Well-Formed Closure Property :

$$\begin{aligned} \text{wf}(\vec{\rho}, \vec{\sigma}) &\stackrel{\text{def}}{=} (\text{Apply} :: \rho, \text{CL}(\hat{\rho}) :: \text{Tuple}(v_1, \ldots, v_n) :: \text{arg} :: \sigma, \ldots) \\ &\Longrightarrow (\hat{\rho} \mathbin{@} \rho, \text{Tuple}(v_1, \ldots, v_n) :: \text{arg} :: \sigma, \ldots) \Longrightarrow^* (\rho, \mathit{result} :: \sigma, \ldots) \end{aligned}$$

Take the subset of machine instructions that can appear in the body of a closure (thus, exclude Form_Closure and Save). We classify these instructions into those which grow the stack (i_g) and those which shrink or do not change the size of the stack (i_s).

$$\begin{aligned} i_g &::= \text{Push } n \mid \text{Decompose\_Tuple } (n > 1) \mid \text{Compose\_Tuple } 0 \\ \rho_g &::= i_g \text{ list} \\ i_s &::= \text{Add} \mid \text{Subt} \mid \text{Mult} \mid \text{Div} \\ &\quad \mid \text{Roll } n \mid \text{Unroll } n \mid \text{Apply} \\ &\quad \mid \text{Push @fun} \mid \text{Load } n \mid \text{Compose\_Tuple } (n > 0) \mid \text{Decompose\_Tuple}(n \leq 1) \\ \rho_s &::= i_s \text{ list} \end{aligned}$$

Let us define the following property.

Grow-Shrink Property :

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

The Apply instruction may be injective because of the linear constraints in our language. Recall that to reverse Apply, we must split the program such that we restore the original closure. We define this split as follows.

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

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

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

A further aim is to prove that InjApply holds so long as the closure we are trying to recover is of the form $cl\ ::=\ \text{CL}(\hat{\rho})$ where gs $(\hat{\rho})$ and cl is well-formed.