Linear Lambda Calculus Compiler Documentation

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

December 19, 2015

Overview:

Given a linear lambda calculus expression, the compiler translates the expression to stack machine instructions. Once the set of instructions have been produced, the expression can be evaluated to a value `<val>`, which can either be an integer `<i>` or a function closure `<closure>`.

The following grammar defines an expression `exp`, where `<var>` is a string that represents a value.

```
exp ::= <i>
      | <var>
      | <exp> + <exp>
      | let <var> = <exp> in <exp>
      | λ<var>.  <exp>
      | <exp> <exp>
```

The compiler assumes that expressions are well-typed. In expressions of the form `<exp> + <exp>`, the expressions added must evaluate to integers. Expressions of the form `<exp> <exp>` are assumed to be valid function applications. Since the language is linear, each variable is assumed to be used exactly once.

Stack Machine Instruction Set:

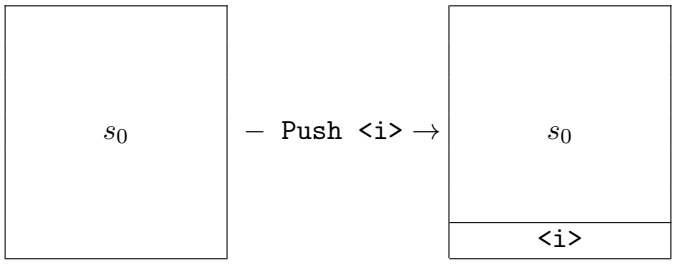
An instruction can be any of the following:

```
instr ::= Push <i> | Add | Roll <i> | Form_Closure (<i>, <i>) | Apply
```

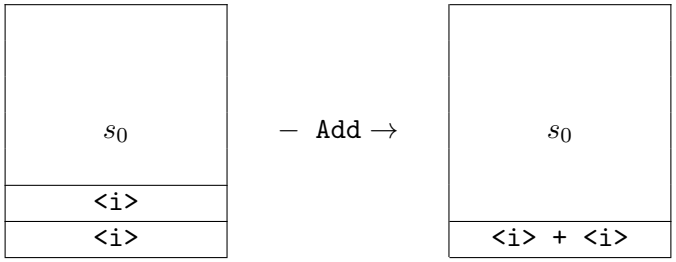
The execution of each instruction modifies the stack, a list of `stack_value`, defined as such:

```
stack_value ::= Int <i> | Closure(<instr list>, <stack_value list>)
```

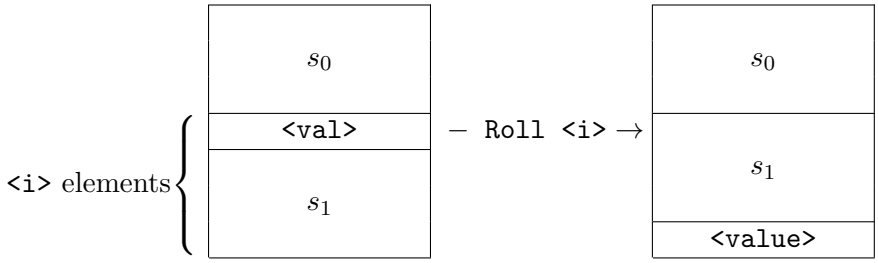
In this documentation and the code, it is also useful to define `program ::= instr list` and `stack ::= stack_value list`.



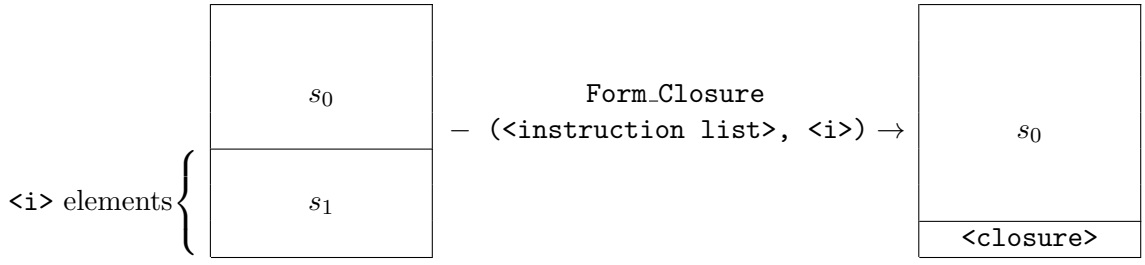
`Push <i>` simply pushes an integer to the bottom of the stack.



`Add` pops the two integers at the bottom of the stack, adds them, and pushes the result to the stack.

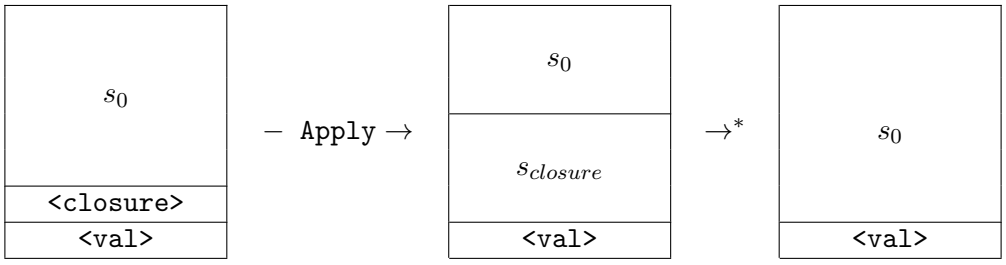


`Roll <i>` removes the `<i>`th element, where the elements are 1-indexed, from the stack and pushes it to the stack. `Roll 1` makes no changes to the stack.



`Form_Closure (num_ops : i, num_vals : i)` removes the bottom `<num-vals>` values from the stack. Additionally, if we represent the program we are executing as a series of instructions `Form_Closure (<num_ops, num_vals), instr-1, instr-2, ..., instr-n`,

then the instructions `intr-1, instr-2, ..., instr-num_ops` are removed from the program and placed into the closure, and the program we are executing becomes `instr-(num_ops + 1), ..., instr-n`. The values removed from the stack and the instructions removed from the program constitute the closure pushed to the stack.



`Apply` assumes that the bottom values of the stack are a closure and the argument to the function. Recall that a closure contains some small stack $s_{closure}$ of values and a list of instructions $p_{closure}$. Say p is the program we are executing, then after executing `Apply`, we proceed to execute $p_{closure}$ @ p. Furthermore, `Apply` pushes $s_{closure}$ onto the stack, then executes a `Roll <i>` instruction, where `<i>` is the number of values in $s_{closure}$, placing the argument to the function at the top of the stack.

Translation to Stack Machine Instructions:

Translation from expression `exp` to stack machine instructions occurs through a series of recursive calls to the function `to_stack`. Through each call, the list of instructions is updated. The compiler also maintains a representation of the stack (to keep track of the location of variables `<var>`, which have string identifiers, and constants `<const>` but not their values). As an instruction is added, the compiler's representation of the stack updates. What follows explains the behavior of the recursive function `to_stack` depending on the expression.

`exp ::= <i>` adds a `Push <i>` operation to the list of instructions. The compiler pushes `<const>` to its stack representation.

`| <var>` determines the location of the variable in the stack representation then adds a `Roll <i>` operation to the instructions such that the variable is placed at the bottom of the stack. The stack representation is appropriately updated after removing the variable from its old location.

`| <exp> + <exp>` recursively calls `to_stack` for each expression, then adds `Add` to the instruction list. Then, the compiler pops the bottom two elements of the stack representation and pushes `<const>`.

`| let <var> = <exp> in <exp>` recursively calls `to_stack` for the first expression, then pops the bottom value in the stack representation and pushes `<var>`. Then, the compiler calls `to_stack` for the second expression.

| (λ <var>. <exp>) first statically determines which of the free variables in the stack are used in <exp>. A helper function recurses through the expression, maintaining the original stack s_0, a stack of free variables $s_{closure}$ used from the original stack, and a stack of local variables s_{local} introduced in the expression (let-expressions and arguments to lambdas introduce local variables).

Each time a variable is referenced, the compiler searches s_{local} and $s_{closure}$ for it. In the case that is not found, a Roll <i> is added to the instruction list and the variable is removed from the original stack s_0 and placed on $s_{closure}$.

The motivation behind shifting these referenced free variables to the bottom of the stack is to prepare the stack for a Form_Closure (num_ops, num_vars) instruction. The bottom num_vars values are free variables referenced in the body of the lambda. Because of linear constraints, free variables referenced in the body must be removed from the stack s_0 to enforce the constraint that each variable is used exactly once.

After statically determining the free variables the function body uses and maneuvering them into place at the bottom of the stack, to_stack is called on the body of the lambda with $s_{closure}$ as the stack and the argument to the function at the bottom of $s_{closure}$. The list of instructions produced assumes that the free variables will be particular positions at the bottom of the stack, which is an assumption that holds when the function is applied.

Finally, a Form_Closure (num_ops, num_vars) instruction with the number of instructions that constitute a program of the function body and the number of elements in $s_{closure}$ is produced. In addition, the free variables present in $s_{closure}$, which are at the bottom of the stack, are removed from the stack representation.

| <exp> <exp> recursively calls to_stack for both expressions and adds Apply to the list of instructions. Then the compiler pops two values in the stack representation and pushes <const>, which represents the result of function application.

Bidirectionality:

Suppose we have executed some program $\rho \in \texttt{program}$, which produces some value v on the stack. Given a history of which instructions have been executed $\rho_h \in \texttt{program}$, 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 $\sigma \to \sigma$, where $\sigma \in \texttt{stack}$. 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 $\sigma_h \in \texttt{int list}$. The rules we present below show the bidirectionality for each instruction using the following tuple:

$$(\rho \in \texttt{program}, \sigma \in \texttt{stack}, \sigma_h \in \texttt{int list}, \rho_h \in \texttt{program})$$

Bidirectionality Rules:

$$\begin{gathered}(\texttt{Push}\ i :: \rho, \sigma, \sigma_h, \rho_h) \\ \iff (\rho, \texttt{Int}\ i :: \sigma, \sigma_h, \texttt{Push}\ i :: \rho_h)\end{gathered}$$

$$\begin{gathered}(\texttt{Add} :: \rho, \texttt{Int}\ i_1 :: \texttt{Int}\ i_2 :: \sigma, \sigma_h, \rho_h) \\ \iff (\rho, \texttt{Int}\ (i_1 + i_2) :: \sigma, i_1 :: \sigma_h, \texttt{Add} :: \rho_h)\end{gathered}$$

$$\begin{gathered}(\texttt{Roll}\ i :: \rho, v_1 :: v_2 :: \cdots :: v_i :: \sigma, \sigma_h, \rho_h) \\ \iff (\rho, v_i :: v_1 :: \cdots :: v_{i-1} :: \sigma, \sigma_h, \texttt{Roll}\ i :: \rho_h)\end{gathered}$$

$$\begin{gathered}(\texttt{Form\_Closure}\ (num\_ops, num\_vals) :: i_1 :: \cdots :: i_{num\_ops} :: \rho, v_1 :: \cdots :: v_{num\_vals} :: \sigma, \sigma_h, \rho_h) \\ \iff (\rho, \sigma, \sigma_h, \texttt{Form\_Closure}\ (num\_ops, num\_vals) :: \rho_h)\end{gathered}$$

$$\begin{gathered}(\texttt{Apply} :: \rho, v :: \texttt{Closure}\ (\hat{\rho}, \hat{\sigma}) :: \sigma, \sigma, \sigma_h, \rho_h) \\ \iff (\hat{\rho}\ @\ \rho, \hat{\sigma}\ @\ \sigma, |\hat{\rho}| :: |\hat{\sigma}| :: \sigma_h, \texttt{Apply} :: \rho_h)\end{gathered}$$

Now we will discuss the non-injective functions and the approach to maintaining the history tape. When reversing an `Add` instruction, one cannot uniquely determine two integers whose sum is is the integer at the top of the stack, therefore, when adding `Int` i_1 and `Int`i_2, we place i_1 onto the history since the function $(+)i_1$ is injective.

Likewise, when reversing an `Apply` operation, one cannnot uniquely determine which instructions belong in the closure and which stack values are used in the closure, therefore to restore `Closure`$(\hat{\rho}, \hat{\sigma})$, we must add the lengths of $\hat{\rho}$ and $\hat{\sigma}$ to the history tape upon executing an `Apply` instruction.

Further Aims:

The `Apply` instruction may be injective because of the linear constraints in our language. Recall that in our stack-machine, we can define a well-formed closure, where all the local variables of the closure are consumed when executing the body of the closure, using the property `wf`.

$$\vec{x} = (\hat{\rho}_x \in \texttt{program}, \hat{\sigma}_x \in \texttt{stack}, \rho_x \in \texttt{program}, \sigma_x \in \texttt{stack})$$

$$\texttt{wf}\ (\vec{x}, v \in \texttt{stack\_value}) \quad \stackrel{\text{def}}{=} \quad (\texttt{Apply} :: \rho, v :: \texttt{Closure}\ (\hat{\rho}, \hat{\sigma}) :: \sigma, \dots) \Longrightarrow (\hat{\rho}\ @\ \rho, v :: \hat{\sigma}\ @\ \sigma, \dots) \Longrightarrow^* (\rho, v' :: \sigma, \dots)$$

Recall that to reverse `Apply`, we must split the program and the stack such that we restore the original closure. We define this split by the following:

$$\texttt{splits}\ (\rho', \sigma', v, \vec{x}) \quad \stackrel{\text{def}}{=} \quad \rho' = \hat{\rho}_x\ @\ \rho_x \wedge \sigma' = \hat{\sigma}_x\ @\ \sigma_x :\ \texttt{wf}\ (\vec{x}, v)$$

Using the linear constraints of the language, if we could prove the following proposition, then `Apply` would be injective.

$$\texttt{Prop}: \quad \forall \rho', \sigma'.\ \exists\ \vec{x}\ \texttt{splits}\ (\rho', \sigma', \vec{x}) \wedge \exists\ \vec{y}\ \texttt{splits}\ (\rho', \sigma', \vec{y}) \Longrightarrow \vec{x} = \vec{y}$$

The constraints placed on the language currently are not enough to prove `Prop`. We must require that every well-formed closure contains at least one instruction, because otherwise, a valid split of ρ' would trivially always be $\rho' = \texttt{[]}\ @\ \rho'$.

Additionally, consider the following counterexample as to why `Prop` may not hold. The body of the function ($\lambda x.$ `let` $sum = x{+}1$ `in` sum) translates to the following stack-machine instructions:

$$\rho_c := \texttt{[Roll 1; Push 1; Add; Roll 1]}$$

However, one could choose to emit the last instruction. Therefore,
`Closure` (`[Roll 1; Push 1; Add]`, `[]`) and `Closure` (ρ_c, []) are both valid well-formed closures. Furthermore, `Closure` (`[Roll 1]`, `[]`) is also a well-formed closure. We could decide to remove `Roll 1` altogether from all programs, but body of the identity function ($\lambda x.\ x$) would only be representable as an empty program, which would violate our constraint that every closure must contain at least one instruction.

Therefore, `Prop` is currently not true under the current constaints of our stack-machine. A further aim of mine is adding constraints such that `Prop` holds true.