Linear Lambda Calculus Compiler Documentation

CS 4999: Anna Yesypenko Supervised by Professor Nate Foster

May 18, 2015

Overview:

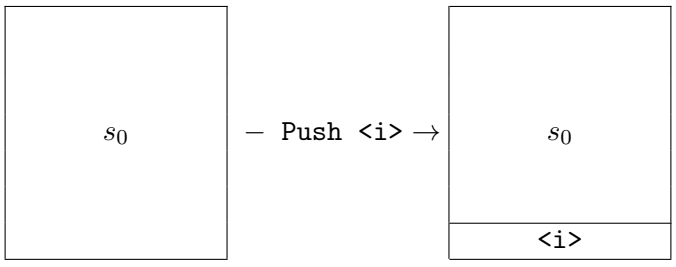
Given a linear lambda calculus expression, the compiler translates the expression to stack machine instructions. Once the set of instructions have been produced, the expression can be evaluated to a value `<val>`, which can either be an integer `<i>` or a function closure `<closure>`.

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

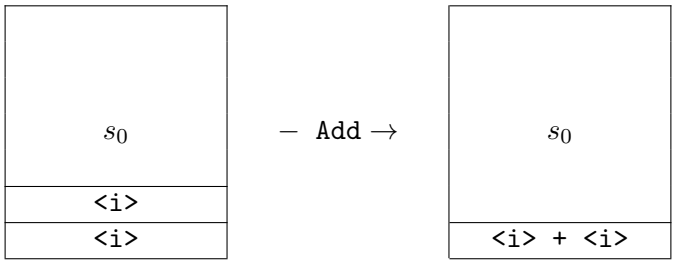
```
exp ::= <i>
      | <var>
      | <exp> + <exp>
      | let <var> = <exp> in <exp>
      | ( λ <var>.  <exp>)
      | <exp> <exp>
```

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.

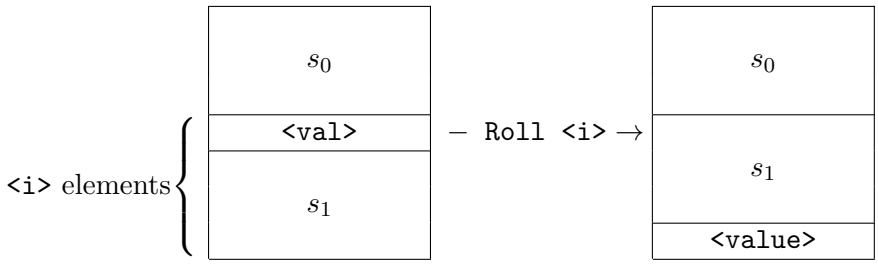
Stack Machine Instruction Set:



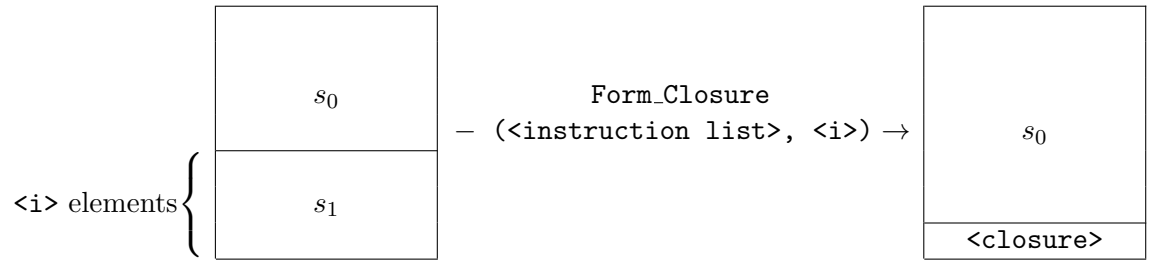
`Push <i>` simply pushes an integer to the bottom of the stack.



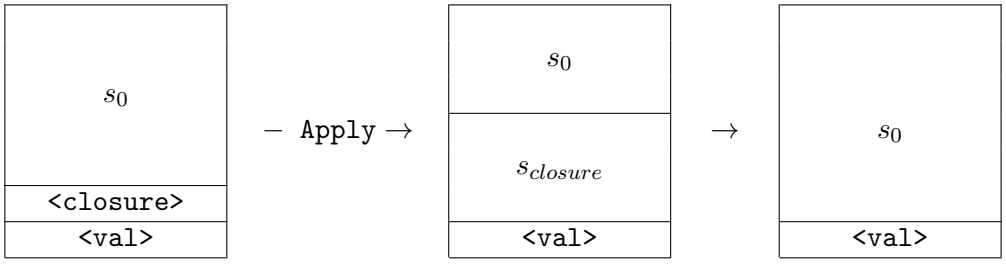
`Add` pops the two integers at the bottom of the stack, adds them, and pushes the result to the stack.



`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 (<instruction list>, <i>)` removes the bottom `<i>` elements from the stack. These elements, along with the instruction list, form 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. `Apply` pushes these values onto the stack, then executes a `Roll <i>` instruction, where `<i>` is the number of values in $s_1 closure$, placing the argument to the function at the bottom of the stack.

Then, `Apply` executes the list of instructions. Because of linear constraints, the stack at the end of an `Apply` instruction should be the same as at the beginning of the instruction, except the closure and its argument are replaced with the result of the function application.

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 (<instruction list>, <i>)` instruction. The bottom `<i>` 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 (<instruction list>, <i>)` instruction with the instruction list 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.

Further Aims:

In many situations, it may be useful to define two functions, where one performs the inverse operation of the other. However, if you change one function, you will have to maintain the other such that they are still inverses. Though seemingly simple, this task can be tedious and difficult to debug for a programmer.

Thankfully, because of the linear constraints of the language, one should be able to compile a function in linear lambda calculus to instructions and reverse them to define an inverse function, which given the result of a function application should be able to produce the original argument. The beginning of the next semester of my research will involve implementing a bidirectional programming language and exploring its applications.