The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 7

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

1 April 2025

Table of Contents

1 Introduction

In this tutorial, we will introduce a *stack machine* for evaluating expressions. This material is for your self-study and *will not be covered* in the exam.

2 A Stack Machine

One disadvantage of our previous implementation of evaluation is that we make the control flow *implicit*. In other words, we are utilizing the semantics of the meta-language (Haskell) to help define the semantics of our language (SNH). Take the evaluation of addition for example:

```
evaluate (Add a b) = evaluate a + evaluate b
```

From the above code, we don't know which operand (`a` or `b`) is evaluated first. This is all controlled by Haskell. Of course, the order doesn't matter to addition (we get the same result either from left to right or from right to left.) But for `If` expressions:

```
evaluate (If a b c) =
  let BoolV test = evaluate a
  in if test then evaluate b else evaluate c
```

The semantics of `If` in SNH tells us that either `b` or `c` is evaluated but not both. Here we are utilizing the semantics of `if` in Haskell to help guarantee that. But what if the Haskell compiler has a bug and evaluates both branches of `if` no matter what the predicate is? Then we are in trouble.

That's our motivation to use a stack machine to make the control flow *explicit*. Java is one example of the real-world programming languages that use a stack machine for execution. The standard model is that we compile Java source code to Java bytecode, which is then interpreted by the Java virtual machine (JVM). One benefit of bytecode is that it is portable to any architecture as long as the JVM is available.

Here we introduce a different stack machine from Java called **K**. A state s of the K machine consists of a control stack k, which records the work remaining to be done after an instruction is executed, and a closed expression e. States are in either of the two forms:

1. An *evaluation* state of form $k \succ e$, which corresponds to the evaluation of a closed expression e on a control stack k;
2. A *return* state of form $k \prec v$ (we use v to imply that v must be a value), which returns a value v that has been computed.

The control stack represents the evaluation context. It records the "current location" of evaluation, where the value of the current expression is returned to. A control stack k may contain control frames f as well as evaluation environments Δ:

$$k ::= \epsilon \mid k \triangleright f \mid k \blacktriangleright \Delta$$

Formally speaking, a stack is either empty (ϵ), or a stack k plus a control frame f ($k \triangleright f$), or a stack k plus an evaluation environment Δ ($k \blacktriangleright \Delta$).

A frames f is defined as follows:

$$\begin{array}{lr}
f ::= u(-) & \text{unary operations} \\
\quad \mid o(-, e_2) \mid o(v_1, -) & \text{binary operations} \\
\quad \mid \texttt{call}(-, e_2) \mid \texttt{call}(v_1, -) & \text{function calls} \\
\quad \mid \texttt{decl}(x, -, e) & \text{variable declarations} \\
\quad \mid \texttt{if}(-, e_1, e_2) & \text{conditionals}
\end{array}$$

A hole (denoted as $-$) in the top frame is intended to hold the value returned by evaluating the current expression. It corresponds to the location in an expression where evaluation are taking place. By placing a hole in an expression, we effectively specify which part to evaluate first.

We often need to get the most recent (outermost) evaluation environment of a stack. This operation is denoted by $\Delta(k)$ and is recursively defined by:

$$\begin{aligned}
\Delta(\epsilon) &= \cdot \\
\Delta(k \triangleright f) &= \Delta(k) \\
\Delta(k \blacktriangleright \Delta') &= \Delta'
\end{aligned}$$

The main judgment defining the stack machine is:

$$s \longmapsto s'$$

It means that state s makes a transition to state s' in one step. An initial state of the stack machine has the form $\epsilon \succ e$, and a final state has the form $\epsilon \prec v$. Transitions of the stack machine are in a *small-step* style, and to establish a relation with big-step operational semantics taught in the lectures and tutorials, we have: if $\cdot \vdash e \longrightarrow v$, then

$$\epsilon \succ e \longmapsto \cdots \longmapsto \epsilon \prec v$$

We begin with the rule for literals. To evaluate a literal n, we simply return it:

$$k \succ n \longmapsto k \prec n$$

Next, we consider the rules for unary expressions:

$$k \succ u(e) \longmapsto k \triangleright u(-) \succ e$$

$$k \triangleright u(-) \prec v \longmapsto k \prec u(v)$$

To evaluate $u(e)$, we push the frame $u(-)$ onto the stack to record the pending unary operation and evaluate e; when a value v is returned, we calculate the result of $u(v)$.

The rules are similar for binary expressions:

$$k \succ o(e_1, e_2) \longmapsto k \triangleright o(-, e_2) \succ e_1$$

$$k \triangleright o(-, e_2) \prec v_1 \longmapsto k \triangleright o(v_1, -) \succ e_2$$

$$k \triangleright o(v_1, -) \prec v_2 \longmapsto k \prec o(v_1, v_2)$$

To evaluate $o(e_1, e_2)$, we first push the frame $o(-, e_2)$ to the stack and evaluate e_1. When v_1 is returned, we pop the top frame and push a new frame $o(v_1, -)$ to the stack. After e_2 is evaluated, we return the result of $o(v_1, v_2)$. This accounts for the left-to-right evaluation order of binary expressions.

Then it comes to the rules for conditionals:

$$k \succ \texttt{if}(e_1, e_2, e_3) \longmapsto k \triangleright \texttt{if}(-, e_2, e_3) \succ e_1$$

$$k \triangleright \texttt{if}(-, e_2, e_3) \prec \texttt{true} \longmapsto k \succ e_2$$

$$k \triangleright \texttt{if}(-, e_2, e_3) \prec \texttt{false} \longmapsto k \succ e_3$$

To evaluate $\texttt{if}(e_1, e_2, e_3)$, we first push the frame $\texttt{if}(-, e_2, e_3)$ to the stack and evaluate e_1. If `true` is returned, we evaluate e_2; otherwise, we evaluate e_3. Note that we don't need the help of Haskell's `if` expressions, we gain the control by ourselves!

The rule for functions is as simple as that for literals:

$$k \succ \texttt{function}(x)\{e\} \longmapsto k \prec (\texttt{function}(x)\{e\}, \Delta(k))$$

We pack the most recent environment $\Delta(k)$ together with the function definition to form a closure.

Now let's consider the rules for function calls:

$$k \succ \texttt{call}(e_1, e_2) \longmapsto k \triangleright \texttt{call}(-, e_2) \succ e_1$$

$$k \triangleright \texttt{call}(-, e_2) \prec v_1 \longmapsto k \triangleright \texttt{call}(v_1, -) \succ e_2$$

$$k \triangleright \texttt{call}((\texttt{function}(x)\{e\}, \Delta), -) \prec v_2 \longmapsto k \blacktriangleright (\Delta, x = v_2) \succ e$$

Just like our previous implementation of function calls, we first evaluate e_1 and then e_2. The result of evaluating e_1 must be a closure. We then push the environment $(\Delta, x = v_2)$ to the stack and continue to evaluate the body e.

The rules for variable declarations are as follows:

$$k \succ \texttt{decl}(x, e_1, e_2) \longmapsto k \triangleright \texttt{decl}(x, -, e_2) \succ e_1$$

$$k \triangleright \texttt{decl}(x, -, e_2) \prec v_1 \longmapsto k \blacktriangleright (\Delta(k), x = v_1) \succ e_2$$

We first push a new frame $\texttt{decl}(x, -, e_2)$ to the stack and evaluate the bound expression e_1. Then we pop the top frame and push the environment $(\Delta(k), x = v_1)$ to the stack, and evaluate e_2.

Finally, we have the rule for variables:

$$k \succ x \longmapsto k \prec \Delta(k)(x)$$

To evaluate x, we get the most recent environment $\Delta(k)$ from the stack and fetch the value from it.

The last rule is just to repeatedly pop environments:

$$k \blacktriangleright \Delta \prec v \longmapsto k \prec v$$

It takes quite a lot of time to write down all the rules! These rules will make much more sense when you try to evaluate an expression by yourself. I encourage you to do that with pen and paper before writing any code. As an example, below shows the evaluation derivation of `(function(x){x + 1})(3)`:

$$
\begin{array}{llll}
 & \epsilon & \succ & \texttt{call}(\texttt{function}(x)\{x+1\}, 3) \\
\longmapsto & \epsilon \triangleright \texttt{call}(-, 3) & \succ & \texttt{function}(x)\{x+1\} \\
\longmapsto & \epsilon \triangleright \texttt{call}(-, 3) & \prec & (\texttt{function}(x)\{x+1\}, \cdot) \\
\longmapsto & \epsilon \triangleright \texttt{call}((\texttt{function}(x)\{x+1\}, \cdot), -) & \succ & 3 \\
\longmapsto & \epsilon \triangleright \texttt{call}((\texttt{function}(x)\{x+1\}, \cdot), -) & \prec & 3 \\
\longmapsto & \epsilon \blacktriangleright x = 3 & \succ & x+1 \\
\longmapsto & \epsilon \blacktriangleright x = 3 \triangleright +(-, 1) & \succ & x \\
\longmapsto & \epsilon \blacktriangleright x = 3 \triangleright +(-, 1) & \prec & 3 \\
\longmapsto & \epsilon \blacktriangleright x = 3 \triangleright +(3, -) & \succ & 1 \\
\longmapsto & \epsilon \blacktriangleright x = 3 \triangleright +(3, -) & \prec & 1 \\
\longmapsto & \epsilon \blacktriangleright x = 3 & \prec & 4 \\
\longmapsto & \epsilon & \prec & 4
\end{array}
$$

Question 1. Write the evaluation derivation for the following expression:

```
var bar = 4;
var foo = function(x) { x + bar };
var bar = 8;
foo(bar)
```

Now let's turn to the implementation. To start off, we first consider the data structure for frames:

```haskell
data Frame = FUnary UnaryOp
           | FBin BinaryOp (Either Value Exp)
           | FIf Exp Exp
           | FCall1 ()          -- TODO: Give a proper type for both kinds
           | FCall2 ()          --       of call frames
           | FDecl ()           -- TODO: Give a proper type for decl frames
           | FEnv Env
```

Question 2. Replace `()` with proper types for `call` and `decl` frames in `Interp.hs`.

For the sake of convenience, we tactically make environments part of frames. That's why we have the `FEnv` constructor in `Frame`. Then a stack is just a list of frames:

```haskell
type Stack = [Frame]
```

States of our stack machine are represented as follows:

```haskell
data State = Eval Stack Exp
           | Return Stack Value
```

Operation $\Delta(k)$ is just a simple recursive function:

```haskell
getEnv :: Stack -> Env
getEnv [] = []
getEnv (FEnv env : k) = env
getEnv (_ : k) = getEnv k
```

The soul of our stack machine is the `step` function, which specifies how one state makes a transition to another:

```haskell
step :: State -> State
```

Question 3. Implement the `step` function in `Interp.hs` according to the rules we have elaborated above.

To run the stack machine, we just repeat `step` until it reaches the final state:

```haskell
execute' :: Exp -> Value
execute' e = go (Eval [] e)
  where go :: State -> Value
        go (Return [] v) = v
        go s = go (step s)
```

Question 4. Does the K machine support recursive definitions? If so, show the evaluation derivation of `fib(2)`; if not, modify the implementation to support recursion.