The University of Hong Kong

COMP3259: Principles of Programming Languages

Assignment 3

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

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

Deadline: 23:59, 8 May 2025

Table of Contents

Note

Your solution must be submitted in a single zip file, named as *A3_XXX.zip*, with *XXX* replaced by your UID. Please submit your solution on Moodle before the deadline. Your code should be well-written (e.g. proper indentation, readable names, and type signatures). You can test your code using `stack test`. Please at least make sure your code compiles!

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

1 Infinite Streams

1.1 Factorials

We have learned how to make clever use of *cyclic definitions* to define *infinite streams* (in Haskell, they are the same as Lists). For example, in Tutorial 6, it was shown that the Fibonacci sequence can be defined by the following expression:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

We can use `take` to get a finite sublist if we want to take a look. The idea behind the definition is that the sequence of Fibonacci numbers can be viewed as a composition of two streams:

				0
				1
0		1		1
1		1		2
1		2		3
2	+	3	=	5
3		5		8
5		8		13
8		13		21
`fibs`		`tail fibs`		`fibs`

Question 1. (5 pts.) Your first task is to write a cyclic definition `facts` involving infinite streams for factorials starting from 1! in `InfStream.hs`. Try to apply the above idea to think about how it can be decomposed.

Hint: You may need to use infinite streams other than `facts` itself in the definition.

1.2 Prime Factors

A famous problem is to enumerate all positive integers with no prime factors other than 2, 3, or 5, **in ascending order with no repetitions**. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than the three mentioned ones. But this is very inefficient since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, let's call the required stream of numbers S and notice the following facts about it:

- S begins with 1;
- The elements of S, after being multiplied by 2, are also elements of S;
- The same is true for 3 and 5;
- These are all the elements of S.

Now, all we have to do is combine elements from these sources. For this we define a `merge` function that combines two ascending streams into one ascending result stream, eliminating repetitions (unlike the conventional merge process):

Question 2. (5 pts.) Complete the `merge` function in `InfStream.hs`.

Question 3. (5 pts.) After finishing `merge`, construct the S stream with it.

2 Bytecode Compilation

2.1 The Big Picture

Programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine (JVM) all evaluate expressions using a stack. In Tutorial 7, we implemented a stack-based evaluator K, but it still works directly on the unmodified syntax, which introduces some extra complexity. What if we can have a compiled bytecode and a virtual machine consuming it, just like Java? Well, let's build one!

In this problem, we will introduce a stack virtual machine, $\mathcal{S}$, accompanied by a stack-based intermediate language, `SInst` (the "bytecode"). A state of the $\mathcal{S}$ machine consists of three components: a sequence of instructions to be executed, an environment storing local variables, and a stack of operands where the operations are taking place:

```
type SProg = [SInst]
type Env = [(Ident, Value)]
type EnvFrames = [Env]
type Stack = [Value]
type State = (SProg, Stack, EnvFrames)
```

The execution starts with an initial stack `[]` and an initial environment `[[]]` (only contains one frame which is empty). Each time, the machine will take one instruction and perform a transition on state accordingly. Execution terminates when there's no more instruction. In the final state, only the evaluation result will be left on the stack.

Note that during the execution, we actually maintain a list of environment frames for supporting function invocation. We create a new frame each time doing a function call. The top environment frame is attached to the currently executed function, while the below are for previous functions in the call graph. At any given time, only variables in the top-level frame are accessible, and the lower levels are inactivated.

Before going to the full specification, let's gain some intuition by a concrete example!

Note: You may take a glance at the evaluation algorithm for the Reverse Polish notation if you are not familiar with it, which can be seen as the minimal form of a stack machine.

2.2 The Example

An $\mathcal{S}$ program corresponding to the expression `var x = 1; 2 + x` will be:

```
[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
```

Let's simulate the execution process step-by-step:

```
[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
↑
Env = [[]]

|     |
 -----
 Stack

======= Initial state

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
 ↑
Env = [[]]

|     |
|  1  |
 -----
 Stack

======= Push 1 onto the stack

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
          ↑
Env = [[("x", 1)]]

|     |
 -----
 Stack

======= Pop 1 out of the stack, and store it in environment with name "x"

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
                      ↑
Env = [[("x", 1)]]
```

```
|     |
|  2  |
 -----
 Stack

======= Push 2 onto the stack

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
                               ↑
Env = [[("x", 1)]]

|     |
|  1  |
|  2  |
 -----
 Stack

======= Load x (equals to 1) from the environment, and push it to the stack

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
                                          ↑
Env = [[("x", 1)]]

|     |
|  3  |
 -----
 Stack

======= Pop two operands 1 & 2, calculate 2+1, and push result to the stack

[IPush 1, IStore "x", IPush 2, ILoad "x", IBin +, IStoreDead]
                                                  ↑
Env = [[]]

|     |
|  3  |
 -----
 Stack

======= Clear one variable from the environment; final state
```

From the stack, we get `3` as the evaluation result.

2.3 The Executor

The instruction set for our $\mathcal{S}$ language consists of the following instructions:

```
data SInst = IPush Value
           | IPushFun Ident SProg
           | ILoad Ident
           | IStore Ident
           | IStoreDead
           | IUnary UnaryOp
           | IBin BinaryOp
           | IIf SProg SProg
           | ICall
           | IReturn
```

In terms of the operational semantics, they are defined as:

- `IPush v`: push the value `v` onto the stack.
- `IPushFun para body`: akin to `IPush`, but specialized for pushing a `ClosureV` that can only be constructed at runtime, which means the executor should capture the active environment.
- `ILoad x`: load `x` from the environment, and push it to the stack.
- `IStore x`: pop a top value from the stack, and store it in the environment with the name `x`.
- `IStoreDead`: undefine the most recently defined (stored) variable, i.e., remove it from the environment.
- `IUnary op`: pop an operand, perform unary `op` on it, and push the result back to the stack.
- `IBin op`: pop two operands, perform binary `op` on them, and push the result back to the stack.
- `IIf n1 n2`: pop a value from the stack as the predicate. Depending on the predicate, the instruction will branch to either `n1` or `n2`, then hand over the control back to the remaining instructions when the execution in the branch ends. The recommended way is to paste the instructions inside a branch in front of the instructions to be executed.
- `ICall`: pop two values from the stack: one should be the callee function (a `ClosureV`), and another should be the argument. Then, construct a new environment frame with proper variable bindings and jump to the function body.
- `IReturn`: destroy the current environment frame. It will always be executed as the last instruction in a function body, serving as the cleanup before function returns to restore the environment.

In the $\mathcal{S}$ machine, we also adjusted the definition of `ClosureV` to `ClosureV Ident SProg`

`Env` to store compiled programs instead of `Exp` (the type information is omitted as well.)

Make sure you have grasped the execution process. Now it's your task to implement the executor for the $\mathcal{S}$ language.

Question 4. (30 pts.) Complete `sStep :: State -> State` in `StackVM.hs`, that pushes the machine from one state to the next.

Note:

> In the translation process, we will keep the original order of arguments, e.g., for `Bin op e1 e2`, the instructions for `e1` will always be presented before the instructions for `e2`. Whenever you need two operands from the stack, be cautious about their order.

Sidenote:

> Whether having the `IStoreDead` instruction is actually a design choice. Without it, we cannot correctly mark the definition scope of a declared variable and can only let it live for the whole remaining lifetime of the function, allowing expressions like `(var x: Int = 1; 2) + x` to be evaluated, or requires some similar treatment like an IIFE for handling it (recall that `let x = exp in body` can be transformed to `(\x -> body) exp`.)
>
> But we may argue that these malformed programs should be rejected by type-check (Java also uses a static check to achieve it.) Let's keep it anyway for environmental tidiness `^_^`

2.4 The Compiler

The executor has been finished, that's great! What's left to do is a compiler bridging the source language SNH (`Exp`) and our target $\mathcal{S}$ language (`SProg`). Let's push it over the finish line!

Recall our source language definition:

```
data Exp = Lit Value
         | Unary UnaryOp Exp
         | Bin BinaryOp Exp Exp
         | If Exp Exp Exp
         | Var Ident
         | Decl Ident Type Exp Exp
         | Call Exp Exp
         | Fun (Ident, Type) Exp
```

Basically, a sequence of instructions compiled from an expression will eventually produce a value on the stack.

- We begin with `Lit v`, which is translated to an `IPush v` instruction.
- For `Unary op e`, we translate `e` first and put an `IUnary op` at the end.
- The translation works similarly for `Bin op e1 e2`, except we need to process both `e1` and `e2` in turn.
- For `If pred e1 e2`, we get the instructions for `pred`, together with an `IIf` storing the compilation result of `e1` and `e2`.
- An `Var x` should just be an `ILoad x`; and for `Decl x _ e1 e2`, we compile `e1` first, followed by a compiled `e2` enclosed by a pair of `IStore x` and `IStoreDead`, marking that `x` is only effective in `e2`.
- A `Call e1 e2` is analogous to a binary operation: we process `e1`, then `e2`, finally an `ICall`.
- For `Fun (para, _) body`, we translate it to an `IPushFun` storing the compiled body, and stick an `IReturn` as the body's tail.

In this section, we will not consider any form of recursion in variable declarations, and `MultDecl` introduced in subsequent sections is also ignored. Your task here is to translate the process above to actual code.

Question 5. (20 pts.) Complete `sCompile :: Exp -> SProg` in `StackVM.hs`.

Sidenote:

> The `IStoreDead` instruction utilizes the fact that the underlying data structure of our environment can remember the definition order of variables, mak-

ing it possible to just remove the latest one. If that's not the case, we may need to change the format to `IStoreDead x` to undefine a variable explicitly.

Now we have fully completed the infrastructure for the $\mathcal{S}$ machine to execute our SNH language. This is a triumph! You can test your program with `run :: Exp -> Value`.

3 Evaluation Strategies, Revisited

In Tutorial 6, we have shown the difference between call-by-value and call-by-name evaluation and implemented a call-by-name evaluator. Now we are going to revisit this topic and ask you to implement a *call-by-need* evaluator instead.

Recall that call-by-value always evaluates arguments when evaluating function applications. As a result, it can waste a lot of resources on useless computation like:

```
var f = function(x: Int) { 3 };
f(<long-computation>)
```

As `x` is never used in the function body, we don't really need to evaluate the argument `<long-computation>`. Call-by-name delays the evaluation of arguments until it is used, and thus the long computation is avoided.

However, call-by-name is still not ideal because it sometimes duplicates the evaluation of arguments. It is even worse than call-by-value in the following example, as we will evaluate the long computation twice:

```
var f = function(x: Int) { x + x };
f(<long-computation>)
```

Therefore, real-world programming languages like Haskell use an optimized variant of the call-by-name evaluation strategy called *call-by-need*. With call-by-need evaluation, the computation results are cached, so the same argument will never be evaluated more than once.

When implementing a call-by-need evaluator, we need to update the definition of the evaluation environment again:

```
type Env = [(Ident, Either ClosureExp Value)]
```

When evaluating a function application (`f e`) or a declaration (`var x = e; body`), we don't evaluate `e` immediately. Instead, we create a `ClosureExp` that stores the due computation in the environment. Later, when we encounter `x` for the first time, we take the `ClosureExp` out, evaluate it to a `Value`, and update the environment with a new binding that shadows the previous one. Thus, when we encounter `x` again, we can directly get its value from the environment.

If you think twice about the algorithm, you may realize that the type signature of `evaluate` has to be changed too (yes, a `State` monad occurs here, but you don't need to know anything about it to finish this assignment):

```
evaluate :: Exp -> Env -> (Value, Env)
```

We need to include the environment in the output because the evaluation of subexpressions may update some bindings in it. Here we take (`e1 + e2`) for example. First, we evaluate `e1` under the original environment `env0` to get `v1` with an updated environment `env1`. Then we evaluate `e2` under `env1` and get `v2` with a further updated environment `env2`. Eventually, the result of `v1 + v2` with `env2` are returned.

A concrete example is to evaluate the following program:

```
var x = 4 + 8; x + x
```

Initially, the environment is empty. When we see the declaration, we store $x \mapsto 4 + 8$ (the right-hand side is a `ClosureExp`) in the environment. Then we continue to evaluate the body $x + x$. We look up the first x in the environment, evaluate $4 + 8$ to 12, and update the environment with $x \mapsto 12$ (the right-hand side is a `Value`). When looking up the second x, we retrieve the value 12 directly.

Question 6. (20 pts.) Implement the `evaluate` function in `Interp.hs` using the call-by-need evaluation strategy. Please ensure that your implementation respects *lexical scoping* and supports *recursion*.

Note: You can ignore the `MultDecl` case for the moment since it's introduced in the next section.

4 Mutual Recursion

We have seen how to type-check and evaluate recursive functions in SNH. But we are not yet satisfied. In Haskell, you can get not only self-recursive functions, but also mutually recursive functions. For example, the following code works perfectly in Haskell:

```
even :: Int -> Bool
even 0 = True
even n = odd (n - 1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n - 1)
```

And we can run the functions:

```
> even 6
True
> odd 6
False
```

This example is somewhat artificial, but mutual recursion is useful in the real world. To add this power to SNH, we extend `Exp` with a new data constructor `MultDecl`, which consists of a list of possibly mutually recursive declarations followed by the body of the declaration. You can just consider that it declares several variables simultaneously.

```
Exp = ...
    | MultDecl [(Ident, Type, Exp)] Exp
```

In addition, the parser has been modified to allow the following syntax of multiple declarations:

```
vars x: T1 = <exp_x>
and  y: T2 = <exp_y>
and  z: T3 = <exp_z>
......;
<exp_body>
```

For example, `even` and `odd` can be defined in SNH as:

```
vars even: Int -> Bool = function(x: Int) {
  if (x == 0) true;
  else odd(x-1)
}
and odd: Int -> Bool = function (x: Int) {
  if (x == 0) false;
  else even(x-1)
};
even(4)
```

Question 7. (5 pts.) Complete `evaluate (MultDecl xs body)` in `Interp.hs`.

Question 8. (10 pts.) Complete `tcheck (MultDecl xs body)` in `TypeCheck.hs`. For error cases, please report the error message `multi-declaration type mismatch` using the `Left` constructor of the `Either` type.