

Assignment 4: Topics in Functional Programming

CS 442/642

Due March 11th in class

Assignments 2 and 3 were mostly about learning, via implementation, how functional languages “work”. For this assignment, the focus is on the functional programming paradigm itself. You will use various functional programming techniques and languages to implement various different tasks, described below:

Part A—Lazy Programming

The function `lazymap` takes a function and a stream as parameters and returns the stream that results from applying the function to every item in the stream.

- a) Implement `lazymap` in Scheme. Use `delay` and `force`.
- b) Implement `lazymap` in ML. Use `thunking`.
- c) Implement `lazymap` in Haskell.

Part B—Haskell and Type Classes

Start-up code for this question can be found at <http://www.student.cs.uwaterloo.ca/~cs442/secd.hs>

Landin’s SECD Machine

A popular way to specify the semantics of a programming language is to describe an abstract “machine” that performs the computation specified by a program in the language. Several abstract machines exist for interpreting terms in the untyped λ -calculus. Of these, the most famous is Peter Landin’s SECD machine.

The SECD machine consists of four stacks: S, E, C, and D. These are called, respectively, the Stack, the Environment, the Control, and the Dump. The purpose of each is roughly as follows:

- the stack holds the results of computations already performed;
- the environment is a map from program variables to the terms to be substituted for them;
- the control holds the current and remaining computations to be performed—its contents “direct” the actions of the machine;
- the dump is a store for surrounding context when evaluating nested expressions.

We will use the notation $\langle S, E, C, D \rangle$ (denoting the contents of each of the four stacks) to represent a configuration of the machine. Initially, S, E, and D are empty, and C contains the expression to be reduced.

The behaviour of the SECD machine is defined by the following state transitions:

- $\langle S, E, x : C', D \rangle \rightarrow \langle \text{lookup}(x, E) : S, E, C', D \rangle$ —look up a variable in the environment and place it on the stack

- $\langle S, E, \lambda x.M : C', D \rangle \rightarrow \langle \langle x, M, E \rangle : S, E, C', D \rangle$ —convert an abstraction into a *closure*, containing the variable and body of the abstraction, and the current environment; place the closure on the stack
- $\langle S, E, (MN) : C', D \rangle \rightarrow \langle S, E, N : M : @ : C', D \rangle$ —split up the components of an application and put them on the control separately in reverse order (so that N is evaluated first), followed by a special directive, $@$, meaning “apply”
- $\langle S, E, \text{prim} : C', D \rangle \rightarrow \langle \text{prim} : S, E, C', D \rangle$ —if a primitive value is on the control, then transfer the primitive to the stack unchanged
- $\langle \text{prim} : N : S', E, @ : C', D \rangle \rightarrow \langle S', E, \text{prim}(N) : C', D \rangle$ —if a primitive function is on top of the stack and an “apply” directive is on the control, then apply the primitive function to the next item on the stack, and put the result on the control in place of them
- $\langle \langle x, M, E_1 \rangle : N : S', E, @ : C', D \rangle \rightarrow \langle (), \langle x, N \rangle : E_1, M, \langle S', E, C' \rangle : D \rangle$ —if a closure is on top of the stack and an “apply” directive is on the control, then push a triple consisting of the current stack (minus the top two elements), environment, and remaining control, onto the dump
- $\langle M : (), E, (), \langle S', E', C' \rangle : D' \rangle \rightarrow \langle M : S', E', C', D' \rangle$ —when a computation is exhausted (control is empty and stack has a single element on it), restore the previous context from the dump, and push the current result onto the top of the restored stack
- $\langle M : (), E, (), () \rangle \rightarrow \text{done}$: return M —if the dump is empty when the computation is exhausted, then the machine halts
- all other configurations are erroneous

1. Representing λ -terms

You will represent λ -terms using the following **data** declaration:

```
data Term = Var String | Abs String Term | App Term Term | Prim Primitive | INT Int
```

This divides the set of all terms into five classes, with the obvious meanings (for the meaning of **Prim** see the section on Primitives). Unfortunately, when we type, for example, **Abs "x" (Var "x")** at the Haskell prompt, Haskell returns an error, as it does not know how to display λ -terms on the screen. To remedy this problem, we need to make **Term** an instance of the class **Show**. To do this, you must provide an implementation of the function **show**, which maps λ -terms into strings. For example, typing the following at the Haskell prompt:

```
App (Abs "x" (App (Var "x") (Var "x"))) (Abs "x" (App (Var "x") (Var "x")))
```

should cause Haskell to return:

```
((\x.(x x)) (\x.(x x)))
```

BONUS: Arrange for the term to be printed with minimal parenthesization. If you attempt this, be sure to get it right, because if you leave out a pair of parentheses that are actually needed, you risk losing marks on the main question.

2. Representing stack contents

Take a close look at the kinds of information that are stored on each of the machine’s four stacks. Come up with four **data** declarations:

```

data SContents = ...
data EContents = ...
data CContents = ...
data DContents = ...

```

Each `data` declaration above indicates the type of the data that its respective stack can store. At the same time, implement a function `lookUp` that performs lookups in your environment stack. Note: do not be surprised if your `data` declarations are not quite right on your first attempt.

Also include the following `data` declaration for configurations of the entire machine:

```

data SECDConfig = SECD ([SContents], [EContents], [CContents], [DContents])

```

3. Implementing the machine

Code the state transitions of the SECD machine in Haskell, by writing a function `secdOneStep` that maps an SECD configuration (of type `SECDConfig`) to the one that immediately follows it, according to the rules given to you. Then write a function `reduce` that takes a λ -term as a parameter, and passes it through `secdOneStep` repeatedly until a final answer is produced, which it returns to the caller.

You will notice that the value returned by the machine is not quite a λ -expression of type `Term`. For example, if we feed $\lambda x.x$ into the machine, we get back $\langle x, x, \langle \rangle \rangle$, which is a closure of the function $\lambda x.x$ (the first x is the variable of the abstraction, and the second x is the body) in an empty environment. Thus, you will need to implement functionality to translate closures *back* into λ -expressions. In general, the closure $\langle x, M, E \rangle$ is translated back to $\lambda x.M\sigma$ where σ is the set of substitutions $\{[N/y] \mid \langle y, N \rangle \in E\}$. (Note that the N 's themselves may be closures, so this process may have to be iterated.)

To aid debugging and testing, make the type `SECDConfig` a member of class `Show`. When an SECD configuration is displayed on the screen, it should have a four-line representation, as follows:

```

S = <suitable representation of S>
E = <suitable representation of E>
C = <suitable representation of C>
D = <suitable representation of D>

```

4. Primitives

The way it is designed, the SECD machine is difficult to test unless we add some primitives to the λ -calculus. Therefore, we augment the λ -calculus to include integers and some primitive integer operations.

When the SECD machine encounters an integer or a primitive on the control, it simply transfers the integer or primitive, unchanged, onto the stack.

The only primitives you are required to support are `Succ`, which computes the successor of its argument, and `IsZero`, which tests whether its argument is the integer 0. You may support other primitives to provide additional functionality if you like, but this is not required. Supporting binary primitives is particularly difficult, but not impossible (and not required).

If you look at the `data` declaration for terms, you will see a type `Primitive`. You will need to define this type. Make it an enumeration over the primitives your implementation will support. To implement application of primitives to data, write a function `applyPrim` of type `Primitive -> Term -> Term` (this technique of implementing function application is known as *defunctionalization*), and use `applyPrim` when the SECD machine needs to apply a primitive.

Notes

1. The tasks in this portion of the assignment are not algorithmically difficult; your main difficulty is likely to be getting your expressions to type-check. Seek assistance from course staff as necessary.

2. It is indicated, in the description of the SECD machine's state transitions, that there exist erroneous states from which no transition is possible. Given well-formed input, it is not possible, assuming your machine is implemented correctly, to reach such states. Haskell does not have exceptions; therefore, if we wanted to catch these error states, we would need to use some other mechanism, e.g. `Maybe` types. However, you are encouraged to simply leave these cases unhandled. Then, if Haskell reaches such a configuration, you will get a runtime error, with reasonably useful debugging information.
3. To save you some time, you are not being asked to demonstrate any portion of this assignment. However, be sure to test your code thoroughly and carefully. If you are having difficulty devising suitable test cases, contact course staff and we can assist you.

Part C—Continuations

On Assignment 1, you were asked to code the Scheme function `exists` using `foldl` or `foldr`. You were then presented with the following, direct implementation of `exists`:

```
(define exists (lambda (p L)
  (if (null? L) #f
      (if (p (car L)) #t (exists p (cdr L)))
  ))
```

Most of you correctly observed that the direct version of `exists` is more efficient because it stops traversing the list as soon as it finds an element that satisfies `p`, whereas the version based on folding always traverses the entire list. For this question, you will correct this inefficiency inherent in folding by implementing a short-circuiting version of `foldr`, in ML, which you will call `scfoldr`. This function will take three parameters: a function `f`, a default value `i`, and a list `l`, as always. However, the function `f` will now be a ternary function, whose third parameter is a continuation `k` to which it can throw a value if you wish to abort computation immediately and return an answer. `scfoldr` will arrange that when `f` is called, it will be passed the continuation of `scfoldr` itself, so that throwing to this continuation will transfer control all the way out of `scfoldr`.

Implement `scfoldr` in ML, give its type (you can just report what SML/NJ returns, but make sure it makes sense to you!), and then use `scfoldr` to implement a short-circuiting version of `exists`.

Submission

Submit the code portion of your solutions to parts A, B, and C as three separate source files. You should submit your source code on paper **and** electronically, via the MFCF `submit` command. The distribution of the marks for this assignment is 25% each for parts A and C, and 50% for part B. The assignment is due by the beginning of class on March 11.

As always, you are encouraged to seek help from course staff as needed.