CS 360: Programming Languages
Lecture 16: Wrap-up

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- Final exam study guide posted on course schedule/BlackBoard.
- Exam format will be the same as before, but slightly longer (you have 2h instead of 1:20).
- 110 possible points, exam scores capped at 100.

Section 2

Monads and Continuations

Evaluating expressions

```
data Exp = Lit Double
         | Add Exp Exp
         | Mul Exp Exp
  deriving(Show)

eval :: Exp -> Double
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

- We'd like to generalize our interpreter so that it will run in any monad, which will allow us to add effects, like state and failure, without having to rewrite the entire interpreter.
- Let's start with a simple variation...

Evaluating Expressions: a Variation

```
eval :: Exp -> Double
eval (Lit n)      =  n
eval (Add e1 e2)  =  let  v1 = eval e1
                          v2 = eval e2
                     in
                       v1 + v2
eval (Mul e1 e2)  =  let  v1 = eval e1
                          v2 = eval e2
                     in
                       v1 * v2
```

- All we've done is bind intermediate results.
- Why "bind?" We're creating a new variable binding, i.e., we are binding a variable to a specific value.
- This looks a bit like **do** notation. . .

The Identity Monad

```haskell
eval :: Exp -> Id Double
eval (Lit n)      = return n
eval (Add e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x + y)
                       }
eval (Mul e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x * y)
                       }

newtype Id a = Id { runId :: a }

instance Monad Id where
    return x = Id x
    m >>= f  = f (runId m)
```

- The identity monad is just a wrapper around a value!
- This code is equivalent to the code from the previous slide.
- We need one more small change to make it general. . .

The Identity Monad: A Variation

```haskell
eval :: Monad m => Exp -> m Double
eval (Lit n)      = return n
eval (Add e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x + y)
                       }
eval (Mul e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x * y)
                       }

newtype Id a = Id { runId :: a }

instance Monad Id where
    return x = Id x
    m >>= f  = f (unId m)
```

- Only the type signature changed.
- Now our code *abstracts* over the underlying monad.

Continuations

- Long, long ago we looked at the amb interpreter, which was written in **continuation passing style**.
- Recall that a continuation represents "what to do next" in a program. That is, it represents the rest of the computational process.
- Have you ever used a callback in JavaScript? Guess what the callback is. . . .
- The amb interpreter was awkward to write in CPS. Fortunately, we can get the benefits of CPS without the awkward code. But first let's feel the pain.

Evaluating Expressions using Continuations

```haskell
data Exp = Lit Double
         | Add Exp Exp
         | Mul Exp Exp
  deriving(Show)

type Cont r = Double -> r

eval :: Exp -> Cont r -> r
eval (Lit n) k = k n

eval (Add e1 e2) k =
    eval e1 (\v1 -> eval e2 (\v2 -> k (v1 + v2)))

eval (Mul e1 e2) k =
    eval e1 (\v1 -> eval e2 (\v2 -> k (v1 * v2)))
```

Evaluating Expressions using Continuations: A Variation

```
eval :: Exp -> Cont r -> r
eval (Lit n)       k =  k n
eval (Add e1 e2)   k =  eval e1 $ \v1 ->
                        eval e2 $ \v2 ->
                        k (v1 + v2)
eval (Mul e1 e2)   k =  eval e1 $ \v1 ->
                        eval e2 $ \v2 ->
                        k (v1 * v2)
```

- The type `r` is the **result type**—it tells us the type of the result of the *entire* computation.
- At intermediate steps, we call continuations with values of type `Double`.
- Recall the operator for low-precedence function application

  ```
  ($) :: (a -> b) -> a -> b
  ```
- This is slightly less painful, but we still have to manually construct the continuation.

What Continuation to Pass?

```haskell
type Cont r = Double -> r

evalExp :: Exp -> Double
evalExp e = eval e ...

eval :: Exp -> Cont r -> r
eval (Lit n)      k =  k n
eval (Add e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 + v2)
eval (Mul e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 * v2)
```

What Continuation to Pass?

```haskell
type Cont r = Double -> r

evalExp :: Exp -> Double
evalExp e = eval e (\v -> ...)

eval :: Exp -> Cont r -> r
eval (Lit n)      k =  k n
eval (Add e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 + v2)
eval (Mul e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 * v2)
```

What Continuation to Pass?

```haskell
type Cont r = Double -> r

evalExp :: Exp -> Double
evalExp e = eval e (\v -> v)

eval :: Exp -> Cont r -> r
eval (Lit n)      k =  k n
eval (Add e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 + v2)
eval (Mul e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 * v2)
```

What Continuation to Pass?

```haskell
type Cont r = Double -> r

evalExp :: Exp -> Double
evalExp e = eval e id

eval :: Exp -> Cont r -> r
eval (Lit n)      k =  k n
eval (Add e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 + v2)
eval (Mul e1 e2)  k =  eval e1 $ \v1 ->
                       eval e2 $ \v2 ->
                       k (v1 * v2)
```

The Continuation Monad

```haskell
eval :: Monad m => Exp -> m Double
eval (Lit n)      = return n
eval (Add e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x + y)
                       }
eval (Mul e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x * y)
                       }

evalExp :: Exp -> Double
evalExp e = runCont (eval e) id

newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Monad (Cont r) where
    return x = Cont $ \k -> k x
    m >>= f  = Cont $ \k -> runCont m $ \x -> runCont (f x) k
```

The Continuation Monad

```haskell
newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Monad (Cont r) where
    return x = Cont $ \k -> k x
    m >>= f  = Cont $ \k -> runCont m $ \x -> runCont (f x) k
```

- The type `r` is the **result type**. It tells us the type of the value we get after the entire computation has been run.
- The type `a` represents an "intermediate" value.
- When sequencing computations in the continuation monad, we have to provide the proper continuation.
- This plumbing is taken care of once and for all in the definition of bind (`>>=`).

Call-with-current-continuation

```
eval :: Exp -> Cont r Double
eval (Lit n)      = return n
eval (Add e1 e2)  = do { x <- eval e1
                       ; y <- eval e2
                       ; return (x + y)
                       }
eval (Mul e1 e2)  = do { x <- eval e1
                       ; y <- do { eval e2
                                 ; callCC $ \k -> k 10
                                 }
                       ; return (x * y)
                       }

callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

- `callCC`, or "call-with-current-continuation," calls the provided function with the current continuation.
- What happens when we try to evaluate `Mul (Lit 2) (Lit 5)` with this evaluator?
- `callcc` is built-in to Scheme—in Haskell, we can define it as a library function.

An amb execution procedure

```
(lambda (env succeed fail)
  ;; succeed is (lambda (value fail) ...)
  ;; fail is (lambda () ...)
  ...)
```

- The success continuation receives a value and proceeds with the computation.
- Along with the value, the success continuation receives a failure continuation, which is called if the use of the received value leads to a dead end.
- The failure continuation tries another branch of the nondeterministic process.
- An operation with a side effect, like `set!`, may need to be undone before a new choice is made.

Implementation of `if`: amb interpreter

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             ;; success continuation for evaluating
             ;; the predicate to obtain pred-value
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             ;; failure continuation for evaluating
             ;; the predicate
             fail))))
```

amb interpreter: handling set!

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)        ; *1*
               (let ((old-value (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'ok
                          (lambda ()    ; *2*
                            (set-variable-value! var old-value env)
                            (fail2)))))
             fail))))
```

- Assignment is the first place we use continuations rather than just pass them around. Assignment must be undone when we backtrack!
- If evaluation of the value succeeds, we must call the succeed continuation.
- However, if the succeed continuation later reaches a dead end, we must restore the previous value of the variable.

amb and Continuations

```haskell
type SK r a = a -> FK r -> r

type FK r = r

newtype SFK r a = SFK { runSFK :: FK r -> SK r a -> r }

instance Monad (SFK r) where
    return x = SFK $ \fk sk -> sk x fk
    m >>= f  = SFK $ \fk sk ->
               runSFK m     fk  $ \x fk' ->
               runSFK (f x) fk' $ \y fk'' ->
               sk y fk''
```

- We can write a monad that handles the plumbing necessary for success and failure continuations. **Our interpreter doesn't have to change**.
- We must construct success continuations just as with the **Cont** monad, but we also have to pass the correct failure continuation.

amb and Choice

```haskell
type SK r a = a -> FK r -> r

type FK r = r

newtype SFK r a = SFK { runSFK :: FK r -> SK r a -> r }

instance MonadPlus (SFK r) where
    mzero = SFK $ \fk _sk -> fk

    m1 `mplus` m2 = SFK $ \fk sk -> runSFK m1 (runSFK m2 fk sk) sk
```

- The **SFK** monad also permits failure and choice!
- We could use a similar monad to run your Homework 6 interpreter and add support for backtracking search without modifying your interpreter at all.
- How is the **SFK** monad different from the list monad? The list monad allowed your interpreter to successfully run the logic puzzle example.

Continuations

- A continuation is an abstraction of the control state of a program.
- In other words, a continuation represents the state of a computation at a particular point during execution.
- Yet another way to look at a continuation: it represents the rest of the computation, i.e., the continuation of the current computation.
- Have you ever used callbacks? How does a callback relate to a continuation?
- The monad abstraction allows us to write "plain" imperative code and get a CPS version for free—we just have to use the CPS monad!
- Call-with-current-continuation allows us to *capture* the current continuation and do whatever we want with it.
- Success and failure continuations allow us to implement backtracking search.

Section 3

Programming Languages in the "Real World?"

Guess the Language. . . [1]

- Higher-order functions.
- Algebraic data types.
- Pattern matching.
- Strongly typed, with type inference.
- Distinction between values and references.
- Polymorphic polymorphism (parametric, ad-hoc, etc.).
- Monads and Maybes.
- See the book *Functional Programming in Swift* if you are interested.
- Swift *does not* have higher-kinded types.

What language do you think this is?

[1] Thanks to Wouter Swierstra.

Functional programming in the "Real World"

- Although Swift is a relatively recent entry, there are many other languages that have adopted—to a greater or lesser degree—functional features.
- Examples include F♯ (part of the .NET platform), Scala (used by Twitter, Scala, LinkedIn, and many others), Clojure, and Kotlin.
- Industry even uses Haskell! A prominent example is Facebook (they hired Simon Marlow away from MSR Cambridge while I was there). Many financial institutions also use Haskell, but this tends not to be advertised.
- C++11 now has lambdas!
- Writing pure code, whether or not you use a pure language, has many advantages!

PL in the "Real World?"

Does it matter if a language we study is widely used?
Does it matter if the *ideas* embodied by this language are widely used?
Does it matter if the ideas embodied by this language *make us think differently about programming*?

Models, Abstraction, and Compositionality

- To understand the world effectively, we build *models*.
- We build models to understand computers too! These models can range from the very informal, such as the mental models we all use when learning to program, to the very formal, such as the semantics we saw for the **While** language.
- *Models must be robust to be useful*. For example, if our model for what a program does only works for one particular version of one particular compiler on one particular model of a CPU. . . it's not very useful.
- *Abstraction* allows us to isolate complexity, thereby insulating components using the abstraction from both complexity *and change*.
- *Compositionality* is also extremely important. It lets us reason about a complex thing in terms of its constituent parts and the rules used to combine those parts.

My hope...

- *Invariants* are a very powerful aid to reasoning, period.
- *Types* are invariants. Even if a compiler doesn't check them, *thinking* in terms of types can help us reason about what a program is doing.
- Thinking *compositionally* helps us reliably build larger, more complex systems by breaking them down into smaller parts that are then composed together—we reason about the whole by reasoning about the parts and their methods of combination.
- My hope is that after taking this course the way you program—and the way you think about programming—has changed no matter what languages you may use in the future.
- If you happen to actually *like* PL...