Monads

CS242

Lecture 9

Pairs and Currying

- Pairs
  - Constructor: (e,e’) or <e,e’>
  - Destructors: p.l, p.r or p.1, p.2 or fst p, snd p
  - Type: A * B

- Consider a function f of type A * B → C
  - From f we can construct a function of type A → B → C
  - λa.λb.f (a,b)
  - Called *currying* the function

Review: Structural Operational Semantics

$$\frac{}{E \vdash x \rightarrow E(x)} \quad [\text{Var}]$$

$$\frac{}{E \vdash \lambda x.e \rightarrow \langle \lambda x.e, E \rangle} \quad [\text{Abs}]$$

$$\frac{}{E \vdash i \rightarrow i} \quad [\text{Int}]$$

$$\frac{E \vdash e_1 \rightarrow \langle \lambda x.e_0, E' \rangle \qquad E \vdash e_2 \rightarrow v \qquad E'[x: v] \vdash e_0 \rightarrow v'}{E \vdash e_1\, e_2 \rightarrow v'} \quad [\text{App}]$$

Note: $E[x: v]$ is the same environment as $E, x{:}v$. E is extended (or updated if x is already present) at point x to return v.

Review: State

Evaluation rules have the form

$E, S \vdash e \rightarrow v, S'$

Expressions evaluate to a value and update the state.

Review: Evaluation Rules with State

$$\frac{}{E, S \vdash x \rightarrow E(x), S} \quad [\text{Var}]$$

$$\frac{}{E, S \vdash \lambda x.e \rightarrow \langle \lambda x.e, E \rangle, S} \quad [\text{Abs}]$$

$$\frac{}{E, S \vdash i \rightarrow i, S} \quad [\text{Int}]$$

$$\frac{l \notin dom(S)}{E, S \vdash \text{new} \rightarrow l, S[l = 0]} \quad [\text{New}]$$

$$\frac{\begin{array}{c} E, S_0 \vdash e_1 \rightarrow \langle \lambda x.e_0, E' \rangle, S_1 \\ E, S_1 \vdash e_2 \rightarrow v, S_2 \\ E'[x: v], S_2 \vdash e_0 \rightarrow v', S_3 \end{array}}{E, S_0 \vdash e_1\ e_2 \rightarrow v', S_3} \quad [\text{App}]$$

Another Feature: Exceptions

Evaluation rules have one of two forms

$E \vdash e \rightarrow v$ evaluation produces a normal value

$E \vdash e \rightarrow Exc(v)$ evaluation produces an exception

In the second case further evaluation must be *strict* in the exception: Once produced the exception propagates through all other computation until caught or it is the result of the computation.

Evaluation Rules with Exceptions

$$\frac{}{E \vdash x \rightarrow E(x)} \text{[Var]}$$

$$\frac{}{E \vdash i \rightarrow i} \text{[Int]}$$

$$\frac{}{E \vdash \lambda x.e \rightarrow \langle \lambda x.e, E \rangle} \text{[Abs]}$$

$$\frac{E \vdash e \rightarrow v}{E \vdash \text{raise } e \rightarrow Exc(v)} \text{[Raise]}$$

$$\frac{E \vdash e_1 \rightarrow Exc(v)}{E \vdash e_1\, e_2 \rightarrow Exc(v)} \text{[AppE1]}$$

$$\frac{E \vdash e_1 \rightarrow \langle \lambda x.e_0, E' \rangle \quad E \vdash e_2 \rightarrow Exc(v)}{E \vdash e_1\, e_2 \rightarrow Exc(v)} \text{[AppE2]}$$

$$\frac{E \vdash e_1 \rightarrow \langle \lambda x.e_0, E' \rangle \quad E \vdash e_2 \rightarrow v \quad E'[x: v] \vdash e_0 \rightarrow v'}{E \vdash e_1\, e_2 \rightarrow v'} \text{[App]}$$

Beyond Pure Lambda Calculus

- What do lambda calculus+state and lambda calculus+exceptions have in common?

- Several things
  - They are both lambda calculus + “side information”
  - The side information is threaded through the computation in a specific order
  - There are new primitives for manipulating the side information
  - If the extra primitives are not used, the behavior is pure lambda calculus

- This is how programming languages are often described
  - A core functional part (lambda calculus)
  - Plus additional features that go beyond pure functions

But Why Not Pure Lambda Calculus?

- For the example with state, why not make the state an explicit argument to functions?
  - A function a → b that works on state type s could have a type a * s → b * s

- But this signature exposes the state
  - The programmer must explicitly manage it

- An alternative (curried) signature: a → (s → b * s)
  - s → b * s is a *state transformer*

- Factor out M b = s → b * s as an abstract data type

Language Features

- There are many non-functional language features that have similar properties:

- Continuations
- (Certain styles of) concurrency
- Nondeterminism
- Random numbers
- ...

Monads

- We can abstract the common part of these language features
  - Sequencing to thread the extra information through the computation
- Enables *programming* these features in pure lambda calculus
  - In a concise way
- More general than the state transformer abstraction
  - Monads are an abstraction for defining such abstractions

Types

- A monad M a is an abstract type
  - The implementation of M is hidden

- The ``normal'' functional type is a
  - The type of the normal value of the computation

- The extra or side information is hidden in the abstraction M

Operations

return: a → M a

A function for creating an element of a monad.

bind: M a → (a → M b) → M b

Sequencing: Take an element of a monad, unwrap the value inside, and apply a function returning an element of the monad with a value of possibly different type.

Bind is usually written v >>= f, for monad value v and function f.

Discussion

- One take: Not much here!
  - Pretty basic
- A second take: Just the right abstraction, and simple!
  - It turns out that return/bind are enough to implement many language features within the lambda calculus
- Keep in mind that return and bind are different for each monad
  - We have to find appropriate definitions

Partial Functions

- Start with a very simple monad

- An option type Maybe(a) is either a value of type a or nothing

- Useful for expressing the result of partial functions w/o exceptions

- Examples
  - head: List(a) -> Maybe(a) returns nothing if the list is empty
  - div: int -> int -> Maybe(int) returns nothing if the divisor is zero

Partial Functions

Maybe a =

Just a

| Nothing

Recall

Just = λa.λj.λn.j a

Nothing = λj.λn.n

Example use to compose partial functions f and g:

λx.let y = f x in

case y of

Nothing: Nothing

Just v: g(v)

Equivalent to y g Nothing

Partial Functions with Monads

```
Maybe a =
      Just a
    | Nothing

-- monad M = Maybe
return = Just
v >>= f  =  case v of
                Nothing -> Nothing
                Just x -> f x
```

Composing Partial Functions

Consider the composition of two partial functions f and g:

λx. x >>= f >>= g

The Maybe monad handles the Nothing case transparently

- The case analysis is hidden inside of >>=
- Automatically short-circuits the computation if f returns Nothing

Example

```
head x =  case x of
              Nil: Nothing
              Cons(a,as) : Just(a)

-- take the head of the first list of a list of lists
λl. return l >>= head >>= head
```

The State Monad

return: $a \rightarrow M\ a$

return = $\lambda v.\lambda s.(v,s)$ -- note $M\ a = s \rightarrow a * s$ where s is the state type

>>= : $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$

p >>= f = $\lambda s.$ let $(v,s') = p\ s$ in $f\ v\ s'$

Example Use

-- increment a global counter each time function foo is called

-- the state is a single integer

foo = λx. return 3 >>= λv. inc >>= λz.return v

bar = reset >>= foo >>= foo

-- inc and reset are new operations that manipulate the state

inc = λi.(i+1, i+1)

reset = λi.(0,0)

Nicer Syntax …

```
-- increment a global counter each time function foo is called
-- the state is just a single integer
// interpret assignment := as bind, taking a value of type M a
// unwrapping the value of type a
foo x = do {
        v := return 3
        z := inc
        return v }
```

First Principles ...

- We want a stateful function of type a → b
  - Which is a pure function of type a → s → (b,s) if we make the state explicit
- The second piece s → (b,s) is a state transformer
- How do we compose a state transformer s → (a,s) and a stateful function a → s → (b,s)?
  - This is what bind does.

Discussion

- Return & bind do just a few things:

- The e in return e is a pure computation
  - Doesn't know about the state, can be written normally

- Bind handles the "plumbing" of the monad
  - Hides the manipulation of the state except through state primitives
  - And correctly sequences it through the computation

Exceptions

```
Exceptional e a =
            Success a
      |   Exception e

-- monad M = Exceptional e
return:  a → M a
return =  Success

>>=: M a → (a → M b) → M b
v >>= f = case v of
        Exception l -> Exception l
        Success r   -> f r
```

```
throw = Exception

catch e h = case e of
                    Exception l -> h l
                    Success r   -> Success r
```

Using Exceptions

Consider composition of two functions f and g that can raise exceptions:

λx. return x >>= f >>= g

Easy to add a handler for f:

λx. (catch (return x >>= f) h) >>= g

Or for both f and g:

λx. catch (return x >>= f >>= g) h

The threading of the exceptions is tedious without bind

The Continuation Monad

Cont r a = (a → r) → r -- r is the result type of the computation

A continuation monad M = Cont r

return: a → M a

return = λa.λk. k a

>>=: M a → (a → M b) → M b

c >>= f = λk. c (λa. f a k)

return 6 >>= λi. return (7 * i)

The Continuation Monad

- Allows building continuations by extending existing continuations
  - Continuations are composed in pieces
- Note there is no automatic translation
  - This is not a CPS transformation!
- The programmer must build up the desired continuations by hand

Discussion

- Monads are an abstraction for programming language features
- And it's just programming!
  - No need for a compiler
  - Can add or remove features as desired
- Examples of good uses:
  - A small part of the program needs state
    - Use the State monad just in that portion
  - Part of the program needs State and Exceptions
    - Again, just use these monads in the parts where they are needed

Comments

- Three features are important to making monads work

- Higher-order functions
  - Bind is a higher order function
  - Many of the monads wrap higher order functions (continuations)

- Type checking
  - The type checker will complain if monads are used incorrectly
  - Necessary for most programmers to avoid getting tangled up

Upsides

- Since it is ``just programming'', users can write their own monads
  - And they do
  - Many programming patterns are usefully abstracted as monads
- Monads are ubiquitous in Haskell
  - Where they were pioneered
- And have appeared in many other settings
  - Again, easy to adopt new ways of structuring software
  - Even in languages without monads built-in

Downsides

- Monads are not a panacea
  - "It's just programming"

- There are three main limitations
  - Multiple monads don't always compose well
    - State(Exceptions(LC)) has different semantics than Exceptions(State(LC))
    - Monads don't commute
  - To use monads, your program must be structured using return/bind
    - Contagious: Whole program tends to end up being written monadically
    - Major hit when converting non-monadic code to monadic code
  - Performance is not what it could be if the features were built in
    - No free lunch – there is a reason compilers are large and complicated

- And the programs end up looking like C++!

A New View of Languages

- Monads were first used in language semantics
  - An idea borrowed from category theory in mathematics
  - Instead of messy environments with state, exceptions, continuations, use monads to structure the execution rules

- We now view languages as a pure core with monad extensions

- Most languages have the monads built in
  - State, Exceptions, Concurrency, ...
  - Better performance, debugging support, and error messages

- But now we realize many of these features can be implemented within a language with higher-order features
  - Bridges (one of) the divides between functional and Turing languages