Advanced Functional Programming, 1DL450

Lecture 9, 2012-12-03

Cons T Åhs

Modules

- Haskell has modules for separation of name spaces and information hiding
- A file contains a preamble (actually just an extended declaration) indicating which symbols are to be exported
- It is possible to export everything by omitting the the export list

```
module Lisp (Sexpr(Leaf, Cons), traverse, depth) where

data Sexpr a = Leaf a | Cons (Sexpr a) (Sexpr a)

traverse ..

depth ..

-- Not exported
find x (Leaf y) = x == y
..
```

Selective Export

- You need not export all constructors (or any) of a type
- This is good for writing ADTs, since it supports hiding representation
- Here we export just the type and abstract operations
  - We can change internal representation without affecting “users” of the module

```
module AbsList (AbsList,
                empty, isempty,
                cons, first, rest,
                append) where

data AbsList a = Empty | Cons a (AbsList a) | App (AbsList a)
(AbsList a)

empty = Empty

cons x l = Cons x l

append x y = App x y

...
```

Import

- Use `import` to use another module
- Argument similar to export
  - all or selected entities
- Unqualified import allows you to use exported entities as is
  - risk of name collision
  - shorter symbols
  - not clear which symbols are internal or external
- Qualified import means you have to include name of module in use
  - longer symbols (must include module name)
  - easy distinction of external symbols
  - no risk of name collision

```
module Foo (...) where
import Lisp (cons, null, append)
import qualified Erlang (send, receive, spawn)

foo msg queue = Erlang.send pid (cons msg queue)
```

ADT support

- Erlang
  - has support for `opaque` declaration of type to not expose representation
  - atoms are global, do not belong to any module
  - internal, non exported, functions are not accessible outside module
- Lisp
  - atoms belong to packages
  - internal, non exported, symbols are accessible outside package
    - `DEFSTRUCT` or `DEFCLASS` are suitable for ADTs
    - Difficult to *really* hide representation
- Haskell
  - Selective export can be used for hiding representation
  - no atoms, but constructors belong to modules
  - internal, non exported, are not accessible
- All languages support “unqualified” import
  - while making code more compact, it should be avoided for reasons of clarity

I/O is not functional

- Having to do I/O is a natural part of any real, larger program
  - Just doing function calls in a calculator like fashion is not enough
  - Need to communicate with user or other entities
  - Keep persistent state (on disk)
- Handling I/O is quite different between languages
  - Treat as any function that just happens to have a side effect
    - Functional purity breaks down, pragmatics win.
      - Lisp, Erlang
  - Separate from functional parts and try a pure approach
    - Enter *Monads* (from category theory) for I/O and other nifty/nasty stuff
    - Haskell
- When doing I/O there are some desired properties
  - It should be done. Once.
  - I/O statements should be done in sequence.

Handling State

- One way to handle "state" in a functional language is to model our state as one large key-value store
- We can now write functions that correspond to statements in an imperative, multiple assignment language
- A statement will then have the type `State -> State`
- An assignment of x to v would then be something like
  - `update state "x" v :: State`
  - Note: statements after the assignment have to use the new state to get the updated value!
  - If a program consists of statements (`s1; s2`) and these statements are translated to functions `f` and `g` then we want the computation to be
    - `program state = g (f state)`
- It is relatively straightforward to consider a more general imperative language with procedures that take arguments and "functions" that return a value as well as affecting the state.
  - Good exercise!

Handling State

- While this works in theory it breaks down as a model of the real world
  - State is a value that is passed around
  - We can use a state several times (with different effects)
- The above properties would be extremely nice for the real world, but has some serious repercussions as well.
- Enter *Monads* which
  - encapsulates the state, controlling access to it
  - effectively models *computation* (not only sequential)
  - clearly separates your pure functional parts from the impure

Monads

- Think of Monads as representing computation with an encapsulated state
- We introduce type class `Monad m`
- An *action* would have type (returning a value *and* affecting state)
  - `State -> (a, State)`
- The most basic (but far from trivial) operation needed is to perform one action and bind a name to an intermediate value for use in future actions
  - `x <- action1 ; action2`
  - We need a code transformation
  - This would be rather easy to describe using a Common Lisp macro
    - an action returns a cons of a value and new state

```
(defmacro bind (state (x action1) action2)
  (let ((pair (gensym)))
    `(let ((,pair (funcall ,action1 ,state))
      (let ((,x (car ,pair)))
        (funcall ,action2 (cdr ,pair)))))))
```

Monads

- Back to Haskell
- `x <- action1 ; action2`
  - an assignment to `x` (bind result of `action1`)
  - further action (which must take the bound value as an argument)
  - `x` is free in `action2`
  - `action2' = (\x -> action2)`
- The type class Monad defines the *bind* operation where we can execute a second action with the value return from the first action bound (as above) together with a state.
  - Note that it belongs to the monad `m a` (an instance of the Monad type class)
  - type for *bind*
    - `Monad m => m a -> (a -> m b) -> m b`
    - first argument is the monad (current state)
    - second argument is the second action
    - returns new monad (state) possibly of different type
    - bind is infix operator `(>>=)`

Monads

- Haskell introduces notation a `do` notation for working with monads, i.e., introducing sequences of computation with an implicit state.
  - `do expr1; expr2 ; ..`
- An “assignment”
  - `do x <- action1 ; action2`
  - “expands” to
  - `action1 >>= \x -> action2`
- A monad also requires the `return` operation for returning a value (or introducing it into the monad)
- There is also a sequencing operation that does not care for the value returned from the previous operation
  - Can be defined in terms of bind
  - `x >> y = x >>= (\_ -> y)`
  - This is the default definition

A Simple Monad

- A (memory) cell stores an integer
- A cell monad is thus something that contains functions over integers
- We need to define `bind` and `return`
  - `bind` is rather intimidating, but look at parts to see how it deconstructs, chains and reconstructs the state
  - `return` introduces a new value and possibly a new type

```
data CellM a = CellM (Integer -> (a, Integer))

instance Monad CellM where
  CellM c1 >>= fc2
    = CellM (\state -> let (value, newstate) = c1 state
                           CellM c2 = fc2 value
                       in
                       c2 newstate)
  return k = CellM (\state -> (k, state))
```

A Simple Monad

```
data CellM a = CellM (Integer -> (a, Integer))

instance Monad CellM where
  CellM c1 >>= fc2
    = CellM (\state -> let (value, newstate) = c1 state
                           CellM c2 = fc2 value
                       in
                       c2 newstate)
  return k = CellM (\state -> (k, state))

runCell :: Integer -> CellM a -> (a, Integer)
runCell i (CellM c) = c i
```

- Introduce a function that increments the state
- Introduce a function for applying (“running”) the monad

Using the Simple Monad

```
data CellM a = CellM (Integer -> (a, Integer))

instance Monad CellM where
  CellM c1 >>= fc2
    = CellM (\state -> let (value, newstate) = c1 state
                           CellM c2 = fc2 value
                       in
                       c2 newstate)
  return k = CellM (\state -> (k, state))

-- a statement, affects only state
add1 :: CellM ()
add1 = (CellM (\state -> ((), state+1)))

-- extract the single value from the state
value :: CellM Integer
value = CellM (\i -> (i, i))

-- expression that compares the state with a given value
isValue :: Integer -> CellM Bool
isValue x = CellM (\i -> (i == x, i))
```

- Introduce some “statements” of the monad - create a language

Using the Simple Monad

```
*Main> :type (do add1)
(do add1) :: CellM ()
*Main> runCell 0 (do add1; add1)
((),2)

*Main> :type (do add1; v<-value;add1;return v)
(do add1; v<-value;add1;return v) :: CellM Integer
*Main> runCell 0 (do add1; v<-value;add1;return v)
(1,2)

*Main> :type (do x <- (isValue 2); y <- value; add1; return (x, y == 1))
(do x <- (isValue 2); y <- value; add1; return (x, y == 1)) :: CellM (Bool, Bool)
*Main> runCell 0 (do x <- (isValue 2); y <- value; add1; return (x, y == 1))
((False,False),1)
```

- Run some “programs” on our single cell memory