Haskell & functional programming, some slightly more advanced stuff

Matteo Pradella
pradella@elet.polimi.it

IEIIT, Consiglio Nazionale delle Ricerche &
DEI, Politecnico di Milano

PhD course @ UniMi - Feb 7, 2011

Haskell

Born in 1990, designed by committee to be:

- purely functional
- with strong polymorphic static typing
- non-strict semantics

I will talk mainly about the marked issues.

Outline

- Evaluation: strict vs lazy
- Side effects and the problem of Input/Output
- Type classes and Monads
- (Semi-Explicit Parallelism)

Evaluation of functions

The basic computation mechanism in Haskell is function application.

We have already seen that, in absence of side effects (purely functional computations) from the point of view of the result the order in which functions are applied does not matter (almost).

However, it matters in other aspects, consider e.g. this function:

```
mult :: (Int, Int) -> Int
mult (x, y) = x*y
```

A possible evaluation:

```
mult (1 + 2, 2 + 3)       -- applying the first +
= mult (3, 2 + 3)         -- applying +
= mult (3, 5)             -- applying mult
= 3*5                     -- applying *
= 15
```

Another possible evaluation:

```
mult (1 + 2, 2 + 3)        -- applying mult
= (1 + 2) * (2 + 3)        -- applying the first +
= 3 * (2 + 3)              -- applying +
= 3*5                      -- applying *
= 15
```

The two evaluations differ in the order in which function applications are evaluated.
A function application ready to be performed is called a reducible expression (or redex) e.g. 3*5 is a redex, while 3*(f x) is not

Evaluation strategies: call-by-value

In the first example of evaluation of mult, redexes are evaluated according to an (leftmost) innermost strategy

i.e., when there is more than one redex, the leftmost one that does not contain other redexes is evaluated

e.g. in mult (1+2, 2+3) there are 3 redexes:
mult (1+2,2+3), 1+2 and 2+3
the innermost that is also leftmost is 1+2, which is applied, giving expression mult(3,2+3)

In this strategy, arguments of functions are always evaluated before evaluating the function itself - this corresponds to passing arguments by value.

Evaluation strategies: call-by-name

A dual evaluation strategy: redexes are evaluated in an outermost fashion

We start with the redex that is not contained in any other redex, i.e. in the example above, with mult (1+2,2+3), which yields (1+2)*(2+3)

In the outermost strategy, functions are always applied before their arguments, this corresponds to passing arguments by name (like in Algol 60).

Termination

Consider the following definition: inf = 1+inf
evaluating inf does not terminate, regardless of evaluation strategy:
inf = 1 + inf = 1 + (1 + inf) = . . .

On the other hand, consider the expression fst (1, inf)
(where fst (x,y) = x):

Call-by-value:
fst (1, inf) = fst (1, 1+inf) = fst (1, 1+(1+inf)) = ...
Call-by-name:
fst (1, inf) = 1

In general, if there is an evaluation for an expression that terminates, call-by-name terminates, and produces the same result.

Haskell is lazy: call-by-need

In call-by-name, if the argument is not used, it is never evaluated; if the argument is used several times, it is re-evaluated each time.

Call-by-need is a memoized version of call-by-name where, if the function argument is evaluated, that value is stored for subsequent uses.

In a "pure" (effect-free) setting, this produces the same results as call-by-name, and it is usually faster.

Tail calls: the bread and butter of functional loops

In strict functional languages (e.g. Scheme, ML, F#), it is common to write loops using tail-recursive functions, i.e. functions having the recursive call in the "tail".

For instance, the classical $foldl$ could be defined as:

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

(intuitively, the recursive call is the last operation to be performed)

Tail call optimization

Tail recursive functions can be compiled as simple loops, so they do not need to use the stack. In imperative pseudo-code:

```
result := z
while xs is not [] do
      result := f result (head xs)
      xs := tail xs
```

Indeed, $foldl$ is usually considered a (memory) efficient function.

Unfortunately, in Haskell this is not the case, because of laziness:

```
foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= (((0 + 1) + 2) + 3)
= 6
```

At each step, a bigger and bigger unevaluated function is built in the heap, and it is only evaluated at the last step.

Haskell is too lazy: an interlude on strictness

There are various ways to enforce strictness in Haskell (analogously there are classical approaches to introduce laziness in strict languages).

E.g. on data with bang patterns
(a datum marked with ! is considered strict).

```
data Float a => Complex a = Complex !a !a
```

(there are extensions for using ! also in function parameters)

Canonical operator to force evaluation is

```
pseq :: a -> t -> t
pseq x y
```

returns y, only if the evaluation of x terminates
(i.e. it performs x then returns y).

A strict version of $foldl$:

```
foldl' f z []     = z
foldl' f z (x:xs) = let z' = f z x
                    in pseq z' (foldl f z' xs)
```

(strict versions of standard functions are usually primed)

Input/Output is dysfunctional

What is the type of the standard function $getChar$, that gets a character from the user? `getChar :: theUser -> Char`?

First of all, it is not referentially transparent: two different calls of $getChar$ could return different characters.

In general, IO computation is based on state change (e.g. of a file), hence if we perform a sequence of operations, they must be performed in order (and this is not easy with laziness) `readChar` can be seen as a function `:: StateOfTheWorld -> Char`.

Indeed, it is an IO action (in this case for Input):
`getChar :: IO Char`

Quite naturally, to print a character we use $putChar$, that has type:
`putChar :: Char -> IO ()`

IO is an instance of the monad class, and in Haskell it is considered as an indelible stain of impurity.

Memento: type classes

provide ad hoc polymorphism and are conceptually similar to Java interfaces. The classical example from the Prelude:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
    x == y      = not (x /= y)
```

Every type that provides equality (i.e. $==$) is an instance of Eq.

To create an instance of the class, we have only to provide a method definition of $==$ or $/=$ (minimal complete definition).

A peculiar type class: Monad

Introduced by Eugenio Moggi in 1991, a monad is a kind of abstract data type used to represent computations (instead of data in the domain model).

Monads allow the programmer to chain actions together to build an ordered sequence, in which each action is decorated with additional processing rules provided by the monad and performed automatically.

Monads also can be used to make imperative programming easier in a pure functional language.

In practice, through them it is possible to define an imperative sub-language on top of a purely functional one.

There are many examples of monads and tutorials (many of them quite bad) available in the Internet.

The Monad Class

```
class Monad m where
    (>>)   :: m a -> m b -> m b
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
    fail   :: String -> m a
    m >> k = m >>= \_ -> k
    fail s = error s
```

`>>=` and `>>` are called bind. return is used to create a single monadic action, while bind operators are used to compose actions.

First note that m is a type constructor, and $m\ a$ is a type in the monad.

Intuitively, in an action there are usually two computations going on:

1. an explicit one, managed by the user of the monad (e.g. of type a);

2. an implicit one, that is automatically carried out by the monad (in a sense "hidden" in m).

In the monad IO, the first one is the data given to/obtained from an IO action, while the second one is used to "represent" the state of the universe.

The monadic laws

For a monad to behave correctly, method definitions must obey the following laws:

1) $return$ is the identity element:

```
(return x) >>= f   <=>   f x
m >>= return       <=>   m
```

2) associativity for binds:

```
(m >>= f) >>= g   <=>    m >>= ( \x -> (f x >>= g) )
```

(monads are analogous to monoids, with `return` $= 1$ and `>>=` $= \cdot$)

Syntactic sugar: the -do- notation

The essential translation of do is captured by the following two rules:

```
do e1 ; e2          <=>     e1 >> e2
do p <- e1 ; e2     <=>     e1 >>= \p -> e2
```

Caveat: -return- does not return

Indeed, a better name for it should be unit.

For example:

```
esp :: IO Integer
esp = do
           x <- return 4 ; return (x+1)

*Main> esp
5
```

An example of standard monad: Maybe

Maybe is used to represent computations that may fail: we either have $Just\ v$, if we are lucky, or $Nothing$.

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return         = Just
    fail           = Nothing
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
```

In this case, the information managed automatically by the monad is the "bit" which encodes the success of the action sequence.

How to design a monad: computations with resources

We will consider computations that “consume” resources. First of all, we define the resource:

```
type Resource = Integer
```

and the monadic data type:

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

Each computation is a function from available resources to remaining resources, coupled with either a result $\in a$ or a suspended computation $\in R\ a$, capturing the work done up to the point of exhaustion.

(Either represents choice: the data can either be `Left a` or `Right (R a)`, in this case. It can be seen as a generalization of $Maybe$)

```
instance Monad R where
    return v = R (\r -> (r, Left v))
```

i.e. we just put the value v in the monad as $Left\ v$.

```
    R c1 >>= fc2 = R (\r -> case c1 r of
            (r', Left v) -> let R c2 = fc2 v in c2 r'
            ...
```

we call $c1$ with resource r. If r is enough, we obtain the result $Left\ v$. Then we give v to $fc2$ and obtain the second R action, i.e. $c2$. The result is given by $c2\ r'$, i.e. we give the remaining resources to the second action.

If the resources in r are not enough:

```
R c1 >>= fc2 = R (\r -> case c1 r of
         ...
         (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
```

we just chain $fc2$ together with the suspended computation $pc1$.

Basic helper functions

run is used to evaluate $R \ \ p$ feeding resource s into it

```
run :: Resource -> R a -> Maybe a
run s (R p) = case (p s) of
      (_, Left v) -> Just v
       _    -> Nothing
```

step builds an $R\ a$ which "burns" a resource, if available:

```
step :: a -> R a
step v = c where
    c = R (\r -> if r /= 0
                 then (r-1, Left v)
                 else (r, Right c))
```

If $r = 0$ we have to suspend the computation as it is $(r, Right\ c)$.

Lifts

Lift functions are used to "lift" a generic function in the world of the monad. There are standard lift functions in Control.Monad, but we need to build variants which burn resources at each function application.

```
lift1 :: (a -> b) -> (R a -> R b)
lift1 f = \ra1 -> do a1 <- ra1 ; step (f a1)
```

We extract the value $a1$ from $ra1$, apply f to it, and then perform a $step$.

$lift2$ is the variant where f has two arguments:

```
lift2 :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f = \ra1 ra2 -> do a1 <- ra1
                         a2 <- ra2
                         step (f a1 a2)
```

Show

```
showR f = case run 1 f of
    Just v  -> "<R: " ++ show v ++ ">"
    Nothing -> "<suspended>"

instance Show a => Show (R a) where
    show = showR
```

Comparisons

```
(==*) :: Ord a => R a -> R a -> R Bool
(==*) = lift2 (==)
(>*) = lift2 (>)
```

For example:

```
*Main> (return 4) >* (return 3)
<R: True>
*Main> (return 2) >* (return 3)
<R: False>
```

Then numbers and their operations:

```
instance Num a => Num (R a) where
   (+)         = lift2 (+)
   (-)         = lift2 (-)
   negate      = lift1 negate
   (*)         = lift2 (*)
   abs         = lift1 abs
   signum      = lift1 signum
   fromInteger = return . fromInteger
```

In this way, we can operate on numbers inside the monad, but for each operation we perform, we pay a price (i.e. $step$).

Using our monad

Now we see R from the point of view of a typical user of the monad, with a simple example.

First we define if-then-else, then the usual factorial:

```
ifR :: R Bool -> R a -> R a -> R a
ifR tst thn els = do  t <- tst
                      if t then thn else els

fact :: R Integer -> R Integer
fact x = ifR (x ==* 0) 1 (x * fact (x - 1))
```

```
*Main> fact 4
<suspended>
*Main> fact 0 -- it does not need resources
<R: 1>
*Main> run 100 (fact 10) -- not enough resources
Nothing
*Main> run 1000 (fact 10)
Just 3628800
*Main> run 1000 (fact (-1)) -- all computations end
Nothing
```

In practice, thanks to laziness and monads, we built a domain specific language for resource-bound computations.

Semi-Explicit Parallelism

It is the "easier" form of parallelism: we explicitly indicate to the compiler computations that can be carried out in parallel.

```
par  :: a -> b -> b      -- note: par x y = y
```

We are suggesting to compute the first argument in parallel with the second (the one whose result we are keeping).

Example: Fibonacci + Euler

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

mkList n = [1..n-1]
relprime x y = gcd x y == 1
euler n = length (filter (relprime n) (mkList n))
sumEuler = sum . (map euler) . mkList

sumFibEuler a b = fib a + sumEuler b
```

Parallel version: 1st attempt

Both fib and $sumEuler$ are quite expensive, but they are independent, so it should be easy to parallelize them:

```
parSumFibEuler a b = f `par` (f + e) where
                     f = fib a ; e = sumEuler b
```

But if we compile and run the two versions on a multi-cores machine, we obtain roughly the same execution speed. . .

Where is the problem?

The current version of $+$ in GHC evaluates first its left argument, hence $f + e$ demands the value of f before starting e. This blocks the potential parallelization.

Indeed, if we change the implementation like this:

```
parSumFibEuler a b = f `par` (e + f) where
                     f = fib a ; e = sumEuler b
```

we obtain roughly a 2x speedup.

A very bad idea

Clearly, this solution is bad: we should not rely on the knowledge of evaluation order of system functions – if in the next version of the compiler the evaluation order of parameters of $+$ were changed, our gain would be lost.

(in many functional languages the evaluation order of functions arguments is left unspecified by design)

So we need a way to specify execution order, and the usual approach is based on $pseq$:

parSumFibEuler with pseq

```
parSumFibEulerP a b = f `par` (e `pseq` (f + e)) where
                        f = fib a ; e = sumEuler b
```

In this case, we are forcing the evaluation of e before $f + e$ (or $e + f$, it is the same).

In conclusion, it is quite easy to parallelize code with par and $pseq$, provided that
1) we have "expensive" computations that are clearly independent
2) we (probably) have to specify execution order when we build up the final result.

Acknowledgments and references

Many examples (or variations thereof) were taken from:

Hudak, Peterson, Fasel, *A Gentle Introduction to Haskell 98*, 1999

Peyton Jones, Singh, *A Tutorial on Parallel and Concurrent Programming in Haskell*, 2008

If you are interested in transational memory in Haskell:
Harris, Marlow, Peyton Jones, Herlihy, *Composable Memory Transactions* (post-publication version), 2006