Imperative functional programming

Lukas Bulwahn

29. November 2007

Theory Tea, WS 2007/08

Overview of this talk

- Motivation for functional programming
- The “State” - Problem in functional programs
- Other solutions (in short)
- The best* solution: Monads
- Conclusion

Disclaimer: I am not teaching how to program with monads! Look at Haskell tutorials on the web...

* this strongly reflects the speaker's opinion. People who think differently are asked to leave immediately.

What people call programming...

- C programs are mainly sequences of commands (assignment, cases, loops ...).
- Every command changes the program state.
- Computation can be described a sequence of states.

- So far, nothing new to you...

What is functional programming?

- Different view on computation:
- Functional programs are equations which describe mathematical functions.
- There is no state and no “imperative” variables.
- Computation is evaluation of an expression.

A simple example

- The list datatype:
  data List a = Nil | Cons a (List a)

  Cons x1 (Cons x2 (Cons x3 Nil))
  = x1:(x2:(x3:[])) = [x1,x2,x3]
- The filter function:

```
filter             :: (a->Bool)->[a]->[a]
filter p []        =  []
filter p (x:xs)    =  if p x then x : filter p xs
                      else filter p xs
```

A simple example (cont.)

- Quicksort:

```
qsort []     = []
qsort (x:xs) = qsort (filter (\y. y < x) xs)
               ++ [x] ++
               qsort (filter (\y. y >= x) xs)
```

- Remarks: ++ = appending two lists

Why Functional Programming matters

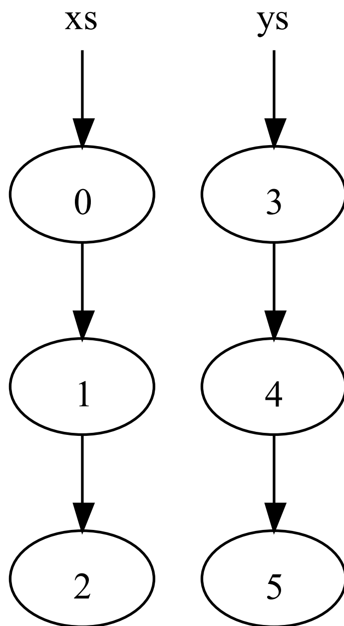
- Functional programs are often easier to understand and to reason about.
- Avoid bugs by writing easier programs (if you are a programmer who writes buggy programs)
- Higher-order functions (such as filter) give you a good abstraction level for describing algorithms

Why Functional Programming matters

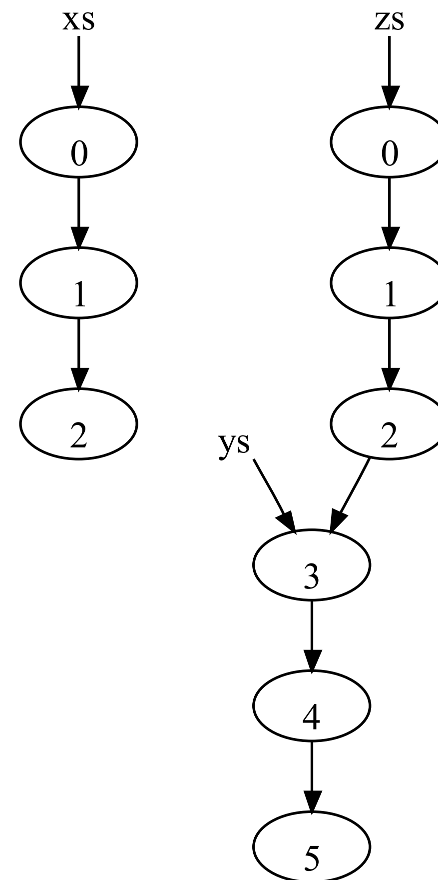
- Purely functional:
  Programs have no side effects.
- Referential transparency:
  An expression is referentially transparent if it can be replaced with its result value without changing the program.
- Purely Functional Data Structures
- many possibilities to optimize:
  memoization, common subexpression elimination, parallelization or other execution orders (evaluation strategies).

Purely Functional Data Structures

- xs = [0, 1, 2]
  ys = [3, 4, 5]
- in the heap:

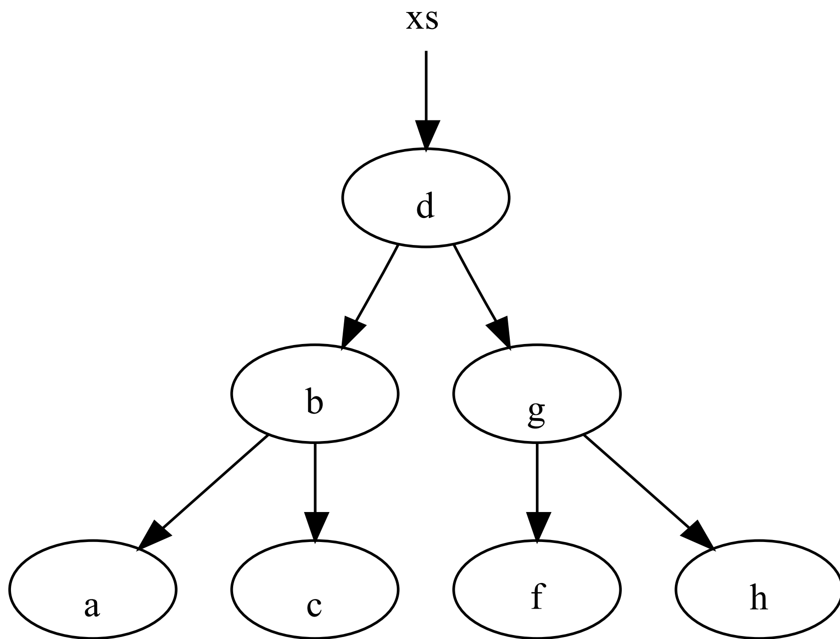


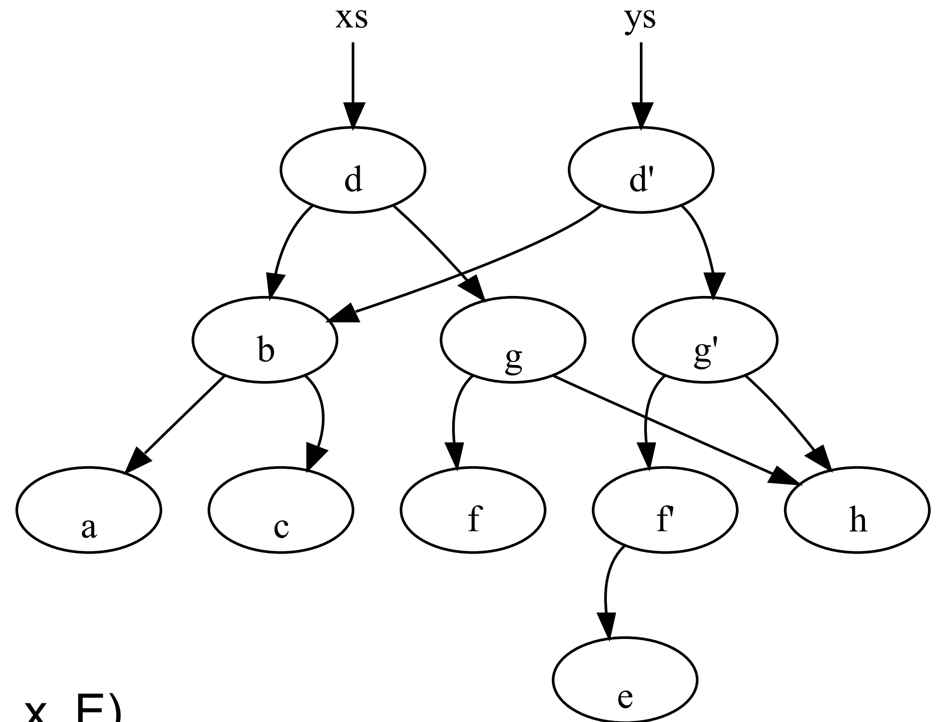
- zs = xs ++ ys
- in the heap:



Purely Functional Data Structures

- A sorted binary tree xs
- ys = insert ("e", xs)






```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
      if x < y then T (insert (x, a), y, b)
      else if x > y then T (a, y, insert (x, b))
      else s
```

Model state in functional languages

- State is still necessary because of Input/Output, User Interaction, ...
- Different approaches:
  - Become impure:
    ex. ReadKey :: unit -> char
    lose referential transparency and possible optimizations.
  - Lazy Streams/Dialogues
  - Continuations
  - Linear types
  - Monads

Our first approach

- State-modifying functions are represented:
  f :: State -> (ReturnType, State)
- fun echo world0 =
```
let (c , world1) = getChar world0
    ((), world2) = putChar c world1
in ((), world2)
```
- But what about this program?
```
let (c , world1) = getChar world0
    ((), world2) = putChar c world0
in ((), world2)
```

Linear/Unique types

- Linearity is ensured by the type system.
  The main function and all state-modifying functions are
  State -> (ReturnType, State)
  and we do not allow functions to duplicate or discard the state.

  No **copy a = (a,a)** or **kill a = ()** for State types.

- Extending the type system is difficult

History of Monads

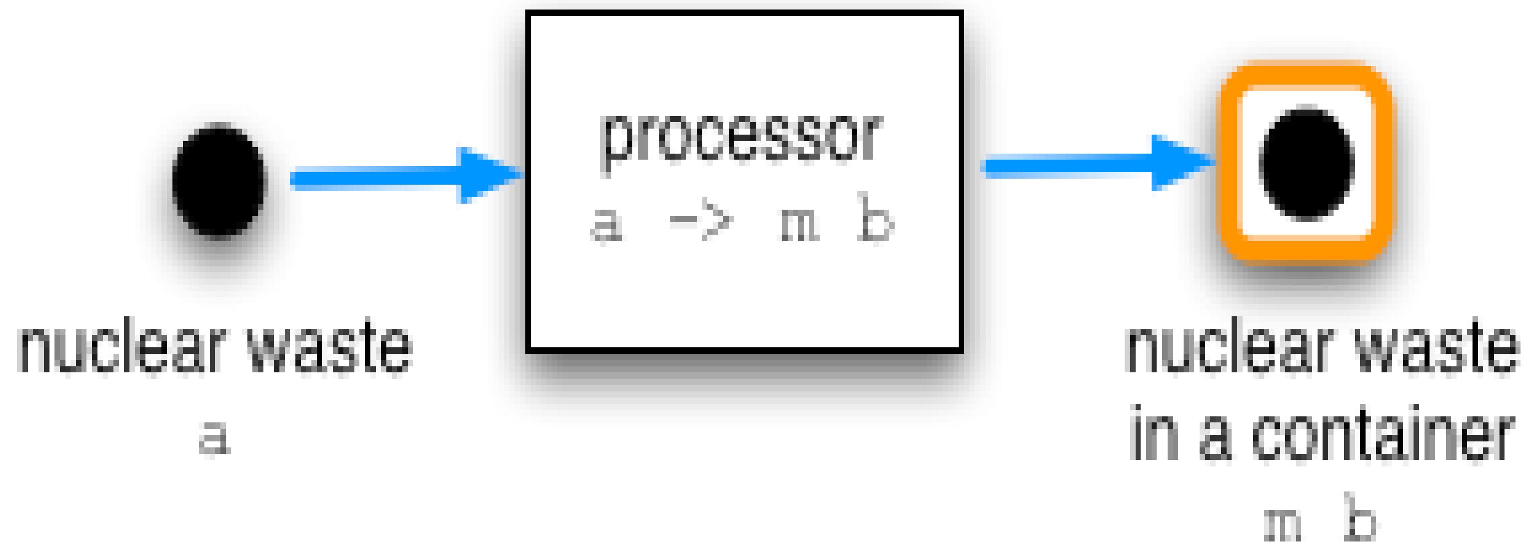
- 1988: Moggi used monads (invented in category theory) to model semantics of programming languages
- 1990: Wadler copied this concept to model state in Haskell
- 1991: Jones&Wadler used monads for IO and calling of C procedures in Haskell
- ... and many others followed them.

Monads

- An abstract datatype a M
  with 2 functions:
  - unit function: return :: a -> a M
  - binding operation: (>>=) :: a M -> (a -> M b) -> M b
- Three simple axioms:
  - (return x) >>= f ≡ f x (Left unit)
  - m >>= return ≡ m (Right unit)
  - (m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
    (binding is associative)

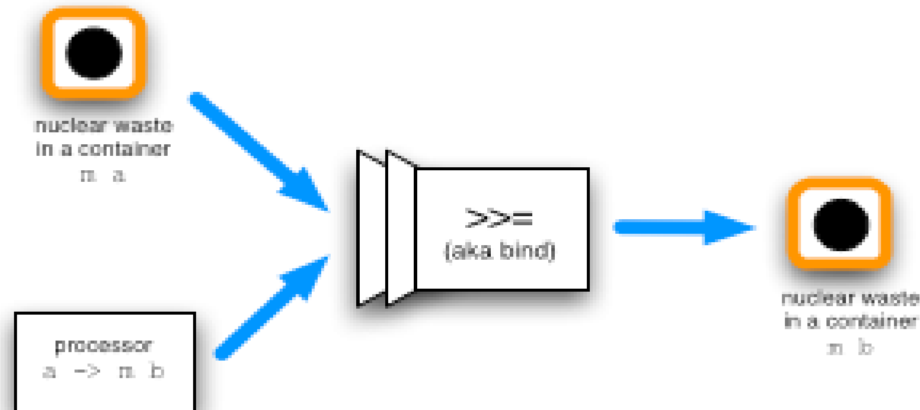
The Nuclear Waste Metaphor

- All (elementary) functions look as follows:



The Nuclear Waste Metaphor

- bind connects the processors to chains of processors



1. remove waste from container
2. put waste in processor
3. send out whatever the processor sends out (well... almost)

What will we put in the container

- We now want to simulate state transitions.
- Idea:

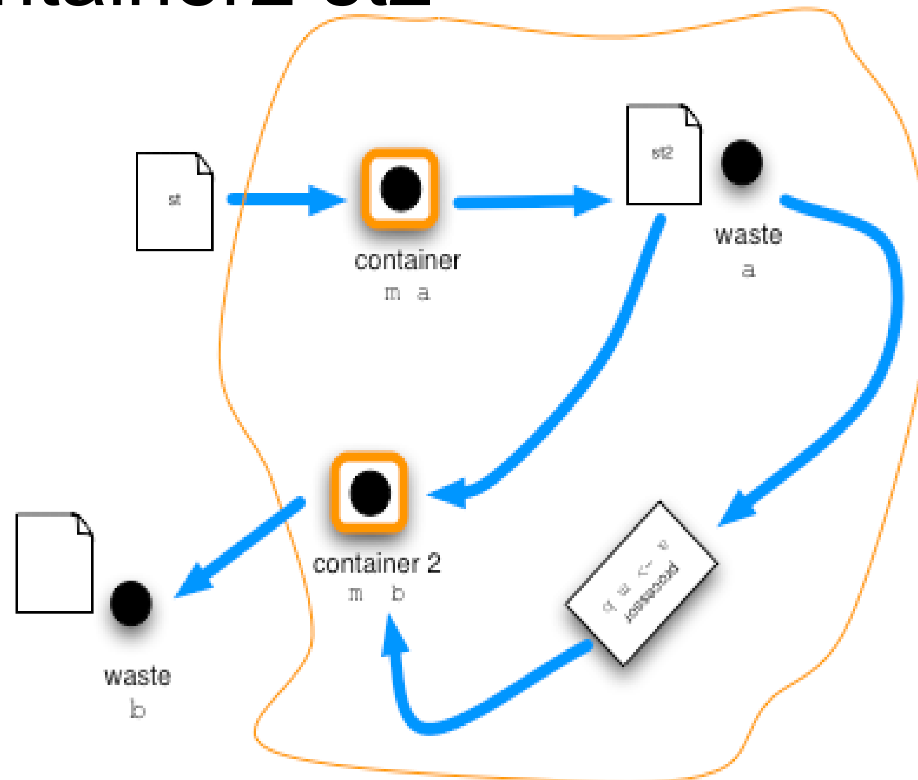
  The container hold action scripts.

- With bind we can compose smaller action scripts to bigger ones.
- The main function returns an action script which will “then” be executed.

State Monad

- Container: functions from state to (retType,state)
- return a = \st -> (a, st)
- container >>= fn =

```
\st -> let (a, st2)    = container st
               container2 = fn a
           in  container2 st2
```



Import Detail for State Monad

- Types are not correct:
  return :: a -> (st -> (a,st))
  but should be like:
  return :: a -> a M
- State functions must be wrapped with a constructor.
- data State s a = State (s -> (a,s))
- return x = State (\st -> (x, st))
- return :: a -> a State
- Same for >>= is left as exercise.
  Also check if monad axioms hold.

Simple Example

- readn :: Int -> IO String
- readn 0 = return []
  readn (n+1) = getChar >>=
  (\c -> readn n >>=
  (\cs -> return (c:cs)))

Simple Example

- readn :: Int -> IO String
- readn 0 = return []
  readn (n+1) = getChar >>= (\c ->
  readn n >>= (\cs ->
  return (c:cs)))

```
readn :: Int -> IO String
readn 0 = return []
readn (n+1) = getChar >>= (\c ->
                readn n >>= (\cs ->
                return (c:cs)))
```

- Ersetze “f >>= (\x ->” durch “x <- f” und setze “do” vor den Block.
  (rein syntaktische Transformation)

Simple Example

- readn :: Int -> IO String
- readn 0 = return []
  readn (n+1) = do
                    c <- getChar
                    cs <- readn n
                    return (c:cs)

- Ersetze "f >>= (\x." durch "x <- f" und setze "do" vor den Block.
  (rein syntaktische Transformation)

Simulate Imperative Programming

- Monads simulate imperative programming
  ... it feels like imperative programming.
  ... it looks like imperative programming.
- So why not program imperative in the first place?
- We only use monads with do-notation when it is necessary.
- You can still use higher-order functions.
- We preserve referential transparency, and therefore can still do many optimizations.

A higher-order Example

- write :: String -> IO()
- write [] = return ()
  write (c : cs) = putChar c >>= (\r -> write cs)
- Could be written in one line:
  write = foldr (>>) (return ()) (map putChar)
- >> is an anonymous bind:
  (>>) :: m a -> m b -> m b
  p >> q = p >>= \x -> q

How Can You This in C?

- List of monads: IO () list
  xs = [putChar 'a', putChar 'b',
  (readn 10 >> return (), putChar 'c']

- fun foo xs = foldr (>>) (return ()) xs
- fun foo' xs = foldr (>>) (return ()) (rev xs)

Conclusion

- Why program functional?
  - It's easy: shorter and less buggy programs
  - Cool optimizations
  - Microsoft thinks so, too: F# is an official MS product
- Why monads are great?
  - Simple abstract concept
  - With the right syntax, it looks like C.
  - Compiler is easy to implement and makes efficient code
  - You can use monads in many other ways...

Literature on FP and Monads

- Why Functional Programming Matters. (Hughes, 1989)
- wikibooks.org – Haskell/Understanding monads (old version)
- Imperative Functional Programming (Jones and Wadler, 1992)
- The Essence of Functional Programming (Wadler, 1992)
- Notions of Computation and Monads (Moggi, 1991)

Thank you!

Any Questions?