A functional specification of effects

Wouter Swierstra

SET seminar, 12/03/09

Functional programming
is great for writing
high assurance software.

Implement a stack.

```haskell
type Stack a = [a]

top :: Stack a -> Maybe a
top [] = Nothing
top (x : xs) = Just x

push :: a -> Stack a -> Stack a
push x xs = x : xs
```

Testing

```
lifoProp :: Int -> Stack Int -> Bool
lifoProp x xs =
  top (push x xs) == Just x
```

```
   Stacks> quickCheck lifoProp
   OK, passed 100 tests.
```

Equational reasoning

$$
\begin{aligned}
& top\ (push\ x\ xs) \\
= & \quad \{\text{definition of } push\} \\
& top\ (x : xs) \\
= & \quad \{\text{definition of } top\} \\
& Just\ x
\end{aligned}
$$

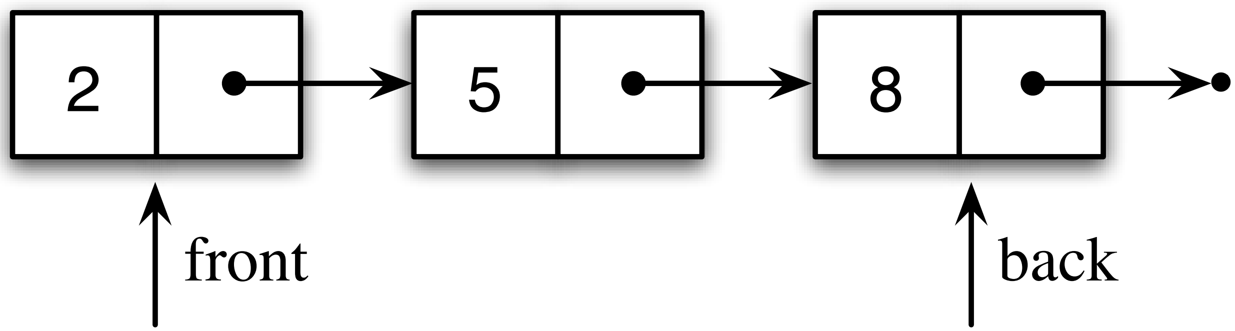
Proof assistants

```
Theorem fifo
  (a : Set) (x : a) (xs : Stack a) :
    top (push x xs) = Some x.
 Proof.
   trivial.
 Qed.
```

The Reasoning Toolkit

- Automatic testing;
- Equational reasoning;
- Proof assistants.

Implement a queue.


2
5
8
front
back

```haskell
data Cell = Cell Int (IORef Cell)
          | NULL

type Queue =
  (IORef Cell, IORef Cell)

enqueue :: Queue -> Int -> IO ()
dequeue :: Queue -> IO (Maybe Int)
empty   :: IO Queue
```

How can we show our program is correct?

The Reasoning Toolkit

- Automatic testing;
- Equational reasoning;
- Proof assistants.

The great divide

Pure & Total

- Easy to reason about.
- Clear semantics
- Tool support for verification, testing, and debugging.

Impure

- Not so much.
- Hardly.
- ...
- **Very useful!**

Pure specifications of impure functions.

Computer memory

```
type Loc  = Int
type Data = Int
type Heap = Loc -> Data
type Mem  = (Loc, Heap)
```

Syntax

```
data IO a =
      Return a
    | Read Loc (Data -> IO a)
    | Write Loc Data (IO a)
    | New Data (Loc -> IO a)
```

(a free monad)

Semantics

```
type Heap = Loc -> Data
type Mem  = (Loc,Heap)

eval :: IO a -> Mem -> (a,Mem)
```

(a monad morphism
from the free monad
to the state monad)

Semantics - Return

```
type Heap = Loc -> Data
type Mem  = (Loc,Heap)

eval :: IO a -> Mem -> (a,Mem)
eval (Return x) m = (x,m)
```

Semantics - Read

```
type Heap = Loc -> Data
type Mem  = (Loc,Heap)

eval :: IO a -> Mem -> (a,Mem)
eval (Read l rd) (l,h) =
  eval (rd (h l)) (l,h)
```

Semantics - Write

```
eval :: IO a -> Mem -> (a,Mem)
eval (Write l d wr) (fresh, heap) =
  eval wr (fresh,update l d m)

update l d heap =
  \l' -> if l == l' then d
         else heap l'
```

Semantics - New

```
eval :: IO a -> Mem -> (a,Mem)
eval (New d new) (fresh, heap) =
  eval (new fresh)
       (fresh + 1, update fresh d m)
```

Queues, revisited

- Now, if we choose:

  ```
  data Data = Cell Int Loc | NULL
  ```

- We can QuickCheck our queues...
- ... and even check that queue reversal is possible in constant memory.

Functional specifications

- In my thesis I present functional specifications in Haskell for:
    - teletype I/O;
    - mutable state;
    - concurrency (MVars and STM).
- and some machinery to syntactically combine specifications.

But...

- The Haskell specification is not **total**...
- so it cannot be transcribed to a proof assistant;
- and equational reasoning with these semantics is not obviously sound.

Problems

- The Haskell specification deals with one fixed type of data;
- and the programmer can access unallocated memory;
- the initial memory is “bogus”

```
type Heap = Loc -> Data

type Mem  = (Loc,Heap)
```

To explain why the functional specifications are total, we need a richer type structure.

Natural numbers

```
data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat

plus : Nat -> Nat -> Nat
plus Zero m = m
plus (Succ k) m = Succ (plus k m)
```

Lists

```
data List (a : Set) : Set where
  Nil : List a
  Cons : a -> List a -> List a

head : List a -> a
head Nil = ???
head (Cons x xs) = x
```

Vectors

```
data Vec (a : Set) : Nat -> Set where
  Nil : Vec a Zero
  Cons : a -> Vec a n -> Vec a (Succ n)

head : Vec a (Succ n) -> a
head (Cons x xs) = x
```

Memory model

- What types can we store on the heap?
- What is the heap?
- What is a reference?

Universes

- A universe is a pair of:
  - a type `U` and
  - a function `el : U -> Set`

Universes – example

```
data U : Set where
  NAT : U
  PAIR : U -> U -> U
  FUN : U -> U -> U

el : U -> Set
el NAT = Nat
el (PAIR s t) = (el s , el t)
el (FUN s t) = (el s) -> (el t)
```

The heap

For some universe...

```
Shape = List U

data Heap : Shape -> Set where
  Empty : Heap Nil
  Alloc : el u -> Heap us ->
          Heap (Cons u us)
```

References

```
data Ref : U -> Shape -> Set where
  Top : Ref u (Cons u us)
  Pop : Ref u us -> Ref u (Cons v us)
```

Syntax

```
data IO (a : Set) : Shape -> Shape -> Set
  Return : a -> IO a s s
  Write : Ref u s -> el u -> IO a s t
    -> IO a s t
  Read : Ref u s -> (el u -> IO a s t)
    -> IO a s t
  New : el u
    -> (Ref u (Cons u s)
           -> IO a (Cons u s) t)
    -> IO a s t
```

Return

```
eval : IO a s t -> Heap s -> (a, Heap t)
eval (Return x) h = (x,h)
```

Write

```
eval : IO a s t -> Heap s -> (a, Heap t)
eval (Write r x wr) h
  = eval wr (update r x h)

update : Ref u s -> el u ->
  Heap s -> Heap s
update Top x (Alloc _ h) = Alloc x h
update (Pop r) x (Alloc y h)
  = Alloc y (update r x h)
```

Read

```
eval : IO a s t -> Heap s -> (a, Heap t)
eval (Read r rd) h
  = eval (rd (lookup r h)) h

lookup : Ref u s -> Heap s -> el u
lookup Top (Alloc x _) = x
lookup (Pop r) (Alloc _ h) = lookup r h
```

New

```
eval : IO a s t -> Heap s -> (a, Heap t)
eval (New x new) h
  = eval (new Top) (Alloc x h)
```

Programming

- We can now define pure versions of functions such as `read` that program with this specification;
- and then use the `eval` function to reason about how such programs behave.
- So we can implement efficient queues, prove their correctness, and compile to Haskell.

Limitations

- Non-modular – you must always carry around the entire heap-shape in the types...
- No higher-order store:

```
Read : Ref u s ->
       (el u -> IO a s t) -> IO a s t
```

- The type of references change when memory is allocated.

Related work

- Hoare Type Theory takes a different approach:
  - postulate the existence of a Hoare Type;
  - add axioms for return and bind;
  - and axioms for read, write, new, fix, ...

Conclusions