Towards a functional specification of effects

Wouter Swierstra

Brighton – 17/09/08

How can we write better software?

Static typing

Model checking

Theorem proving

How can we write better software?

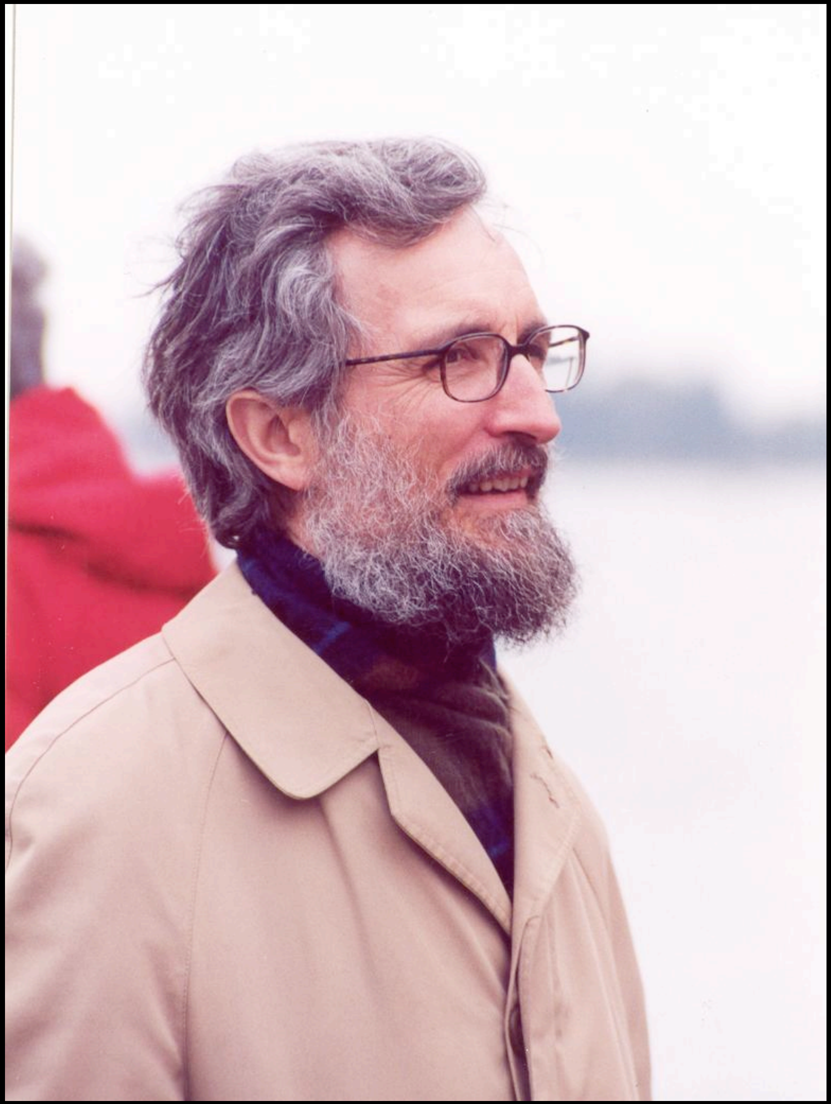
Automatic testing

Static analysis

Best software engineering practices

Type Theory
Per Martin-Löf

- A foundation of constructive mathematics;
- a functional programming language.



Curry-Howard isomorphism

```
isEven : Int -> Bool        5: Int
-----------------------------------
        isEven(5) : Bool
```

Curry-Howard isomorphism

$$\frac{\texttt{isEven : Int -> Bool} \qquad \texttt{5: Int}}{\texttt{isEven(5) : Bool}}$$

$$\frac{p \to q \qquad p}{q}$$ Modus ponens

Curry-Howard isomorphism

- A type system is a logic;
- a type is a proposition;
- a program is a proof.

Type theory

- The type system corresponding to constructive predicate logic.
- Strong mathematical roots;
- Foundations of proof assistants like Coq;
- and new, exciting functional languages.

Problem

- All functions are **pure** and **total**.
- What about:
  - concurrency?
  - Input/Output?
  - general recursion?
  - mutable state?

Haskell

- Primitives:
  - `new : a -> IO (Ref a)`
  - `read : Ref a -> IO a`
  - `write : a -> Ref a -> IO ()`
- IO monad:
  - `return : a -> IO a`
  - `>>= : IO a -> (a -> IO b) -> IO b`

Example

```
increment : Ref Int -> IO Int
increment r = read r >>= \x ->
              write r (x + 1) >>= \y ->
              return x
```

Mutable state

- A pure specification of:
  - creating new references;
  - writing to references;
  - reading from references.
- Implemented in Agda.

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
```

Vectors

```
data Vec (a : Set) : Nat -> Set where
  Nil : Vec a 0
  Cons : a -> Vec a n -> Vec a (Succ n)
```

Universes

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

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

Memory model

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

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

Syntax: key points

- Index `MS` by two shapes, representing the initial and final shape of the heap:

```
run : MS a s t -> Heap s -> (a, Heap t)
```

- We can only refer to allocated memory;
- and there is a canonical choice of empty heap.
- The `MS` type is a *parameterized monad.*

Syntax

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

Semantics: key points

- Plenty of gritty detail...
- ... but we exclusively use total functions.
- Always allocate "at the top of the heap"

Return

```
run : MS a s t -> Heap s -> (a, Heap t)
run (Return x) h = (x,h)
```

Read

```
run : MS a s t -> Heap s -> (a, Heap t)
run (Read r rd) h
  = run (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
```

Write

```
run : MS n m a -> Heap n -> (a, Heap m)
run (Write r x wr) h
  = run 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)
```

New

```
run : MS n m a -> Heap n -> (a, Heap m)
run (New x new) h
  = run (new maxRef) (snoc x h)

maxRef : Ref (suc n)
snoc : Int -> Heap n -> Heap (suc n)
```

Limitations

- You always carry around the entire heap.
- No higher-order store:

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

Further work

- Fancy logic:
  - model of HTT;
  - separation logic;
  - ...