Towards a functional specification of effects

Wouter Swierstra

Brighton – 17/09/08
How can we write better software?
How can we write better software?

Model checking
Automatic testing
Static typing
Theorem proving
Static analysis
Best software engineering practices
Type Theory

Per Martin-Löf

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

\[ \text{isEven} : \text{Int} \rightarrow \text{Bool} \quad 5 : \text{Int} \]

\[ \text{isEven}(5) : \text{Bool} \]
Curry-Howard isomorphism

```latex
isEven : Int -> Bool
5 : Int

\[ \text{isEven}(5) : \text{Bool} \]

\[
\frac{p \rightarrow q \quad p}{q} \quad \text{Modus ponens}
\]
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 \rightarrow\ IO\ (Ref\ a)
  • read : Ref\ a \rightarrow\ IO\ a
  • write : a \rightarrow\ Ref\ a \rightarrow\ IO\ ()

• IO monad:
  • return : a \rightarrow\ IO\ a
  • \triangleright\triangleright= : IO\ a \rightarrow (a \rightarrow\ IO\ b) \rightarrow\ 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)
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)
```
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 $\texttt{MS}$ by two shapes, representing the initial and final shape of the heap:

  $$\text{run : MS a s t --> Heap s --> (a, Heap t)}$$

- We can only refer to allocated memory;

- and there is a canonical choice of empty heap.

- The $\texttt{MS}$ 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)
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

\text{run} : \text{MS } n \; m \; a \rightarrow \text{Heap } n \rightarrow (a, \text{Heap } m)

\text{run (New } x \; \text{new)} \; h
= \text{run (new maxRef)} \; (\text{snoc } x \; h)

\text{maxRef} : \text{Ref } (\text{suc } n)

\text{snoc} : \text{Int} \rightarrow \text{Heap } n \rightarrow \text{Heap } (\text{suc } n)
Limitations

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

\[
\text{Read} : \text{Ref} \ u \ s \ \rightarrow \\
(\text{el} \ u \ \rightarrow \ \text{MS} \ a \ s \ t) \ \rightarrow \ \text{MS} \ a \ s \ t
\]
Further work

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