A total functional specification of mutable state

Wouter Swierstra
Joint work with Thorsten Altenkirch

EffTT – 14/12/07
Problem

- Program with effects in type theory;
- and reason about these programs.
- We don’t want to extend the type theory...
- ... but model effects internally.
Mutable state

• A pure specification of:
  • creating new references;
  • writing to references;
  • reading from references.
An approximation in Haskell
Haskell – syntax

type Ref = Int

data MS a =
    Return a
    | Write Ref Int (MS a)
    | Read Ref (Int -> MS a)
    | New Int (Ref -> MS a)
Smart constructors

new : Int -> MS Ref
new x = New x Return

read : Ref -> MS Int
read r = Read r Return

write : Ref -> Int -> MS ()
write r x = Write r x (Return ())

>>= : MS a -> (a -> MS b) -> MS b
Example

\[
\text{increment : Ref } \rightarrow \text{ MS Int}
\]

\[
\text{increment } r = \text{ do}
\]

\[
\quad x \leftarrow \text{ read } r
\]

\[
\quad \text{write } r (x + 1)
\]

\[
\quad \text{return } x
\]
Haskell – semantics

type Store = (Ref, Ref -> Int)

run :: MS a -> Store -> (a, Store)
run (Return x) store = (x, store)
run (Read r rd) store = ...
run (Write r x wr) store = ..
run (New x nw) store = ..
So what?

• We can already use these semantics for **testing and debugging**.

• Example: run a series of tests to check that applying **increment** twice is the same as adding two to a reference...

• ... but how do we prove these kind of properties?
Problems

• What is the initial store? We need a function \( \text{Ref} \rightarrow \text{Int} \) ...

• What happens when a programmer accesses unallocated memory?

• What if we want to store more than just integers in our references?

• How can we write this in a more formal, type-theoretic setting?
Solution

• Harness the power dependent types!

  We need to record the size of the heap in types

• As a result, a reference to unallocated memory fails to type check.
Heaps and references

data Heap : Nat -> Set where
    | empty : Heap 0
    | alloc : Int -> Heap n -> Heap (suc n)

data Ref : Nat -> Set where
    | top : Ref (suc n)
    | pop : Ref n -> Ref (suc n)
Syntax: key points

- Index $\text{MS}$ by two integers, representing the size of the initial and final heaps:

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

- We can only refer to allocated memory;

- and there is a canonical choice of empty heap.

- The $\text{MS}$ type is a *parameterized monad*. 
Syntax, revisited

data MS (a : Set) : Nat -> Nat -> Set
  | Return : a -> MS n n a
  | Write : Ref n -> Int -> MS n m a
            -> MS n m a
  | Read : Ref n -> (Int -> MS n m a)
           -> MS n m a
  | New : Int
            -> (Ref (suc n) -> MS (suc n) m a)
            -> MS n m a
Semantics: key points

- Plenty of gritty detail...
- ... but we exclusively use total functions.
- Use de Bruijn levels for references.
- Always allocate “at the end of the heap”
Return

run : MS n m a -> Heap n -> (a, Heap m)
run (Return x) h = (x,h)
Read

run : MS n m a -> Heap n -> (a, Heap m)
run (Read r rd) h
  = run (rd (lookup r h)) h

lookup : Ref n -> Heap n -> Int
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 n -> Int -> Heap n -> Heap n
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)
Fancy types

• It’s pretty easy to extend this idea to accommodate references of different types.
• We no longer keep track of the size of the heap...
• ... but now need to keep track of the shape of the heap, i.e., a list of types.
New problems...

• We can write smart constructors as before.
• But what goes wrong in the following fragment?

```haskell
silly : MS 0 2 Int
silly = new 1 >>= \ref1 ->
    new 3 >>= \ref2 ->
    read ref1 2
```
Price of precision

• As we allocate new memory the size of the heap grows...
• ...but this changes the type of valid references!
• We still want the same value – it just inhabits a different type.
• We need a cunning plan.
Not so smart constructors

\[
\text{read} \ : \ \text{Ref } n \rightarrow \text{MS Int } n \ n \\
\text{read } l = \text{Read } l \ \text{Return}
\]

- Idea: teach our smart constructors to weaken references automatically.
Less-than-or-equal

data LEQ : Nat -> Nat -> Set where

stop : LEQ n n

step : LEQ n k -> LEQ n (suc k)

inj : LEQ n k -> Ref n -> Ref k

- We can weaken references using inj
Deciding LEQ

So : Bool -> Set
So True = Unit
So False = Zero

<= : Nat -> Nat -> Bool
leqdec : So (n <= k) -> LEQ n k
Dirty tricks

read : \{\text{So} \ (n \leq k)\}
\rightarrow \text{Ref} \ n \ \rightarrow \ \text{MS} \ \text{Int} \ k \ k
read \ \{p\} \ \text{ref}
= \text{Read} \ (\text{inj} \ (\text{leqdec} \ p) \ \text{ref}) \ Return

• Note the proof is an implicit argument – a programmer never need write it...

• ...because \text{So True} is trivial, Agda fills in this argument itself.
Automatic weakening

• Now we never need to worry about the type of references changes...

• ... as long as we fix the size of our top-level function – i.e., start with a heap of size 0.

• It’d be much better if there was a more principled manner to achieve this.
Further work

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