Beauty in the Beast
A Functional Semantics of the Awkward Squad

Wouter Swierstra
joint work with Thorsten Altenkirch
Implement a stack.
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

\[ \text{lifoProp} :: \text{Int} \rightarrow \text{Stack Int} \rightarrow \text{Bool} \]
\[ \text{lifoProp} \ x \ \text{xs} = \text{top} \ (\text{push} \ x \ \text{xs}) \equiv \text{Just} \ x \]
Testing

\[\text{lifoProp} :: \text{Int} \to \text{Stack Int} \to \text{Bool}\]
\[\text{lifoProp} \ x \ xs = \text{top} (\text{push} \ x \ xs) \equiv \text{Just} \ x\]

Stacks> quickCheck lifoProp
OK, passed 100 tests.
Equational reasoning

\[
\begin{align*}
top \ (push \ x \ xs) & = \ \{ \text{definition of } push \} \\
top \ (x : xs) & = \ \{ \text{definition of } top \} \\
Just \ x &
\end{align*}
\]
Proof assistants

**Theorem** \( \text{Fifo} : \forall a : \text{Set}, \forall x : a, \forall xs : \text{Stack} a, \top (\text{push} x xs) = \text{Some} x. \)
Proof assistants

Theorem Fifo : \( \forall a : \text{Set}, \forall x : a, \forall xs : \text{Stack } a, \)
\( \text{top} \left( \text{push} \ x \ xs \right) = \text{Some } x. \)

Proof.

\text{trivial.}

Qed.
The Reasoning Toolkit

- QuickCheck
- Equational reasoning
- Proof assistants
Functional programming is great for writing high assurance software.
Implement a queue.
data Cell = Cell Int (IORef Cell) | NULL

type Queue = (IORef Cell, IORef Cell)

enqueue :: Queue → Int → IO ()
dequeue :: Queue → IO (Maybe Int)
emptyQueue :: IO Queue
How can we show our program is correct?
The Reasoning Toolkit

- QuickCheck
- Equational reasoning
- Proof assistants
The Reasoning Toolkit

- QuickCheck
The Reasoning Toolkit
The great divide

**Pure**
- Easy to reason about.
- ‘Clear semantics’
- Tool support for testing and debugging.

**Impure**
- Not so much.
- Hardly.
- ...

The great divide

**Pure**
- Easy to reason about.
- ‘Clear semantics’
- Tool support for testing and debugging.

**Impure**
- Not so much.
- Hardly.
- ...
- Very useful!
Pure specifications of impure functions.
Overview

- Pure specifications of:
  - teletype I/O;
  - mutable state; and
  - concurrency.
Plan of attack

• For every specification:
  • Define a **monad**.
  • Define a pure interface to this monad.
  • Define a “run function” for this monad.
A monad

type \textit{Loc} = Int

\textbf{type} \textit{Data} = Int

data \textit{IO}_s \, a =
    \begin{align*}
    \text{Write} \ \textit{Loc} \ \textit{Data} \ (\textit{IO}_s \ a) \\
    \text{Read} \ \textit{Loc} \ (\textit{Data} \rightarrow \textit{IO}_s \ a) \\
    \text{New Data} \ (\textit{Loc} \rightarrow \textit{IO}_s \ a) \\
    \text{Return} \ a
    \end{align*}
instance Monad IO where

  return = Return
  (Write l d io) >>= f = Write l d (io >>= f)
  (Read l rd) >>= f = Read l (λd → rd d >>= f)
  (New d nw) >>= f = New d (λl → nw l >>= f)
  (Return x) >>= f = f x
Plan of attack

• For every specification:
  • Define a **monad**.
  • Define a pure interface to this monad.
  • Define a “run function” for this monad.
Plan of attack

- For every specification:
  - Define a **Monad**.
  - Define a pure interface to this monad.
  - Define a “run function” for this monad.
Pure interface

\[
\begin{align*}
\text{writeIORef} & : \text{Loc} \rightarrow \text{Data} \rightarrow \text{IO}_s () \\
\text{writeIORef} l d &= \text{Write} l d (\text{Return} ()) \\
\text{readIORef} & : \text{Loc} \rightarrow \text{IO}_s \text{ Data} \\
\text{readIORef} l &= \text{Read} l \text{ Return} \\
\text{newIORef} & : \text{Data} \rightarrow \text{IO}_s \text{ Loc} \\
\text{newIORef} d &= \text{New} d \text{ Return}
\end{align*}
\]
Example

\[
\text{swap} :: \text{IORef} \to \text{IORef} \to \text{IO}_{\text{s}} ()
\]

\[
\text{swap refX refY} = \text{do}
\]
\[
\begin{align*}
  x & \leftarrow \text{readIORef refX} \\
  y & \leftarrow \text{readIORef refY}
\end{align*}
\]

\[
\begin{align*}
\text{writeIORef refX y} \\
\text{writeIORef refY x}
\end{align*}
\]
Plan of attack

• For every specification:
  • Define a **Monad**.
  • Define a pure interface to this monad.
  • Define a “run function” for this monad.
Plan of attack

• For every specification:
  • Define a **Monad**.
  • Define a pure interface to this monad.
  • Define a “run function” for this monad.
See Monad Run.

**Idea:** Use the state monad to model how our pure interface behaves.

\[
\begin{align*}
\text{run} &:: IO_s \ a \rightarrow a \\
\text{run} \ io &= \text{evalState} \ (\text{runIOState} \ io) \ \text{emptyStore} \\
\text{runIOState} &:: IO_s \ a \rightarrow \text{State} \ \text{Store} \ a \\
\text{runIOState} &= \ldots
\end{align*}
\]
data Store = Store
{ fresh :: Loc,
  heap :: Loc → Data }
emptyStore :: Store
emptyStore = Store{ fresh = 0 }
Return

\[ \text{runIOState} :: IO_s \ a \rightarrow \text{State} \ \text{Store} \ a \]
\[ \text{runIOState} \ (\text{Return} \ x) = \text{return} \ x \]
Read

\[
\begin{align*}
\text{runIOState} :: & \ IO_s \ a \rightarrow \ State \ Store \ a \\
\text{runIOState} \ (\text{Read} \ l \ \text{rd}) = \ do \\
& h \leftarrow \text{gets heap} \\
& \text{runIOState} \ (\text{rd} \ (h \ l))
\end{align*}
\]
Write

\[
\text{runIOState} :: \text{IO}_s a \rightarrow \text{State \, Store \, a}
\]

\[
\text{runIOState} \left( \text{Write \, l \, d \, wr} \right) = \text{do}
\]

\[
\begin{align*}
\text{store} & \leftarrow \text{get} \\
\text{put} \left( s\{\text{heap} = \text{update \, l \, d \, (heap \, s)}\} \right)
\end{align*}
\]

\[
\text{runIOState \, wr}
\]

\[
\text{update} :: \text{Loc} \rightarrow \text{Data} \rightarrow \text{Heap} \rightarrow \text{Heap}
\]

\[
\text{update \, l \, d \, h \, k}
\]

\[
\begin{align*}
& | \quad l \equiv k \quad = d \\
& | \quad \text{otherwise} \quad = h \, k
\end{align*}
\]
New

\[
\text{runIOState} :: IO_s a \rightarrow \text{State Store} a \\
\text{runIOState} (\text{New } d \text{ nw}) = \text{do} \\
\quad l \leftarrow \text{gets fresh} \\
\quad \text{put} (s\{\text{fresh} = l + 1\}) \\
\quad \text{extendHeap} l \text{ d} \\
\quad \text{runIOState} (\text{nw } l)
\]
Queues, revisited

- Now, if we choose:

  ```haskell
  data Data = Cell Int IORef |
  | NULL
  ```

- We can QuickCheck our queues...

- ...and even check that queue reversal is possible in constant memory.
Limitations

- The heap only stores integers:
- Define your own Data type;
- Use Data.Dynamic.
What else?

- Teletype (getChar, putChar)
- **Input:** stream of characters
- **Output:** list of Maybe Chars, possibly returning a final value.
• Concurrency (MVars and forkI/O)

• **Input:** a scheduler

```
newtype Scheduler = Scheduler (Int -> (Int, Scheduler))
```

• **Output:** final heap and result
The Reasoning Toolkit

- QuickCheck
- Equational reasoning
- Proof assistants
The Reasoning Toolkit

• QuickCheck
The Reasoning Toolkit
Real problems

• I’m using undefined values:

• What is the initial heap?

• What happens when you access unallocated memory?

• How can we store heterogeneous values, without using Data.Dynamic?
• We need to talk about:
  • the **size** of the heap;
  • the **types** of data stored on the heap;
  • what is a **reference** into a heap of size $n$.

Sexy types?
• We need to talk about:
  • the **size** of the heap;
  • the **types** of data stored on the heap;
  • what is a **reference** into a heap of size $n$.

**Sexy types?**

**Dependent types!**
Related work

- Pre-monadic versions of the Haskell Report
- The Awkward Squad.
- ... many many others
IOSpec

- Code from the paper is available:
  - on Hackage;
  - homepage:
    www.cs.nott.ac.uk/~wss/repos/IOSpec/
- Watch out for 0.2 with IO à la carte, STM, ...
Summary
Summary

• Pure specification of impure functions
Summary

• Pure specification of impure functions
• Define a monad; pure interface; and run function.
Summary

- Pure specification of impure functions
- Define a monad; pure interface; and run function.
- Dependent types can help make run total.