Beauty in the Beast

A Functional Semantics of the Awkward Squad

Wouter Swierstra
joint work with Thorsten Altenkirch

Implement a stack.

$$\textbf{type}\ Stack\ a = [a]$$

$$\begin{aligned} &top :: Stack\ a \to Maybe\ a \\ &top\ [\,] \qquad\quad = Nothing \\ &top\ (x : xs) = Just\ x \end{aligned}$$

$$\begin{aligned} &push :: a \to Stack\ a \to Stack\ a \\ &push\ x\ xs = x : xs \end{aligned}$$

Testing

$$lifoProp :: Int \rightarrow Stack\ Int \rightarrow Bool$$
$$lifoProp\ x\ xs = top\ (push\ x\ xs) \equiv Just\ x$$

Testing

$$lifoProp :: Int \rightarrow Stack\ Int \rightarrow Bool$$
$$lifoProp\ x\ xs = top\ (push\ x\ xs) \equiv Just\ x$$

```
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

$$\textbf{Theorem } \mathit{Fifo} : \forall a : \mathit{Set}, \forall x : a, \forall xs : \mathit{Stack}\ a,$$
$$\quad \mathit{top}\ (\mathit{push}\ x\ xs) = \mathit{Some}\ x.$$

Proof assistants

$$\textbf{Theorem}\ Fifo : \forall a : Set, \forall x : a, \forall xs : Stack\ a,$$
$$top\ (push\ x\ xs) = Some\ x.$$
$$\textbf{Proof}.$$
$$trivial.$$
$$\textbf{Qed}.$$

The Reasoning Toolkit

- QuickCheck
- Equational reasoning
- Proof assistants

Functional programming is great for writing high assurance software.

Implement a queue.

$$\textbf{data}\ Cell = Cell\ Int\ (IORef\ \ Cell)$$
$$\quad \mid NULL$$

$$\textbf{type}\ Queue = (IORef\ \ Cell, IORef\ \ Cell)$$

$$
\begin{aligned}
&enqueue &&:: Queue \rightarrow Int \rightarrow IO\ ()\\
&dequeue &&:: Queue \rightarrow IO\ (Maybe\ Int)\\
&emptyQueue &&:: IO\ \ Queue
\end{aligned}
$$

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

$$\begin{aligned}
&\textbf{type}\ Loc\quad = Int\\
&\textbf{type}\ Data = Int
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ IO_s\ a =\\
&\qquad Write\ Loc\ Data\ (IO_s\ a)\\
&\quad \mid Read\ Loc\ (Data \rightarrow IO_s\ a)\\
&\quad \mid New\ Data\ (Loc \rightarrow IO_s\ a)\\
&\quad \mid Return\ a
\end{aligned}$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ \mathit{IO}_s \textbf{ where} \\
\quad \mathit{return} = \mathit{Return} \\
\quad (\mathit{Write}\ l\ d\ \mathit{io}) \gg\!= f = \mathit{Write}\ l\ d\ (\mathit{io} \gg\!= f) \\
\quad (\mathit{Read}\ l\ \mathit{rd}) \quad\ \gg\!= f = \mathit{Read}\ l\ (\lambda d \rightarrow \mathit{rd}\ d \gg\!= f) \\
\quad (\mathit{New}\ d\ \mathit{nw}) \quad \gg\!= f = \mathit{New}\ d\ (\lambda l \rightarrow \mathit{nw}\ l \gg\!= f) \\
\quad (\mathit{Return}\ x) \quad\ \gg\!= f = f\ x
\end{array}
$$

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

$$writeIORef :: Loc \rightarrow Data \rightarrow IO_s\ ()$$
$$writeIORef\ l\ d = Write\ l\ d\ (Return\ ())$$

$$readIORef :: Loc \rightarrow IO_s\ Data$$
$$readIORef\ l = Read\ l\ Return$$

$$newIORef :: Data \rightarrow IO_s\ Loc$$
$$newIORef\ d = New\ d\ Return$$

Example

$$
\begin{aligned}
& swap :: IORef \to IORef \to IO_s\ () \\
& swap\ refX\ refY = \mathbf{do} \\
& \quad x \leftarrow readIORef\ refX \\
& \quad y \leftarrow readIORef\ refY \\
& \quad writeIORef\ refX\ y \\
& \quad writeIORef\ refY\ x
\end{aligned}
$$

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.

$$run :: IO_s\ a \rightarrow a$$
$$run\ io = evalState\ (runIOState\ io)\ emptyStore$$

$$runIOState :: IO_s\ a \rightarrow State\ Store\ a$$
$$runIOState = ...$$

Store

$$\mathbf{data}\ Store = Store$$
$$\{fresh :: Loc,$$
$$heap :: Loc \rightarrow Data\}$$

$$emptyStore :: Store$$
$$emptyStore = Store\{fresh = 0\}$$

Return

$$runIOState :: IO_s\ a \rightarrow State\ Store\ a$$
$$runIOState\ (Return\ x) = return\ x$$

Read

$$
\begin{array}{l}
runIOState :: IO_s\ a \rightarrow State\ Store\ a \\
runIOState\ (Read\ l\ rd) = \mathbf{do} \\
\quad h \leftarrow gets\ heap \\
\quad runIOState\ (rd\ (h\ l))
\end{array}
$$

Write

$$
\begin{aligned}
&runIOState :: IO_s\ a \rightarrow State\ Store\ a\\
&runIOState\ (Write\ l\ d\ wr) = \mathbf{do}\\
&\quad store \leftarrow get\\
&\quad put\ (s\{heap = update\ l\ d\ (heap\ s)\})\\
&\quad runIOState\ wr
\end{aligned}
$$

$$
\begin{aligned}
&update :: Loc \rightarrow Data \rightarrow Heap \rightarrow Heap\\
&update\ l\ d\ h\ k\\
&\quad \mid l \equiv k \qquad\quad = d\\
&\quad \mid otherwise = h\ k
\end{aligned}
$$

New

$$
\begin{aligned}
&runIOState :: IO_s\ a \rightarrow State\ Store\ a \\
&runIOState\ (New\ d\ nw) = \mathbf{do} \\
&\quad l \leftarrow gets\ fresh \\
&\quad put\ (s\{fresh = l + 1\}) \\
&\quad extendHeap\ l\ d \\
&\quad runIOState\ (nw\ l)
\end{aligned}
$$

Queues, revisited

- Now, if we choose:

$$\mathbf{data}\ Data = Cell\ Int\ IORef \\ \quad | \ NULL$$

- 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 forkIO)
  - **Input:** a scheduler

$$\mathbf{newtype}\ Scheduler = \\ \quad Scheduler\ (Int \rightarrow (Int, Scheduler))$$

  - **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 of 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.