Beauty in the Beast

*Functional specifications of effects*

based on joint work with Wouter Swierstra

Thorsten Altenkirch
University of Nottingham
Math vs. Programming

• My vision: programming is constructive Mathematics
• No need for mathematical models of (pure) functional programs.
• No difference between a mathematical function and a function in programming.
• Pure functions have no effects . . .
• . . . and always give an answer.
Real world

- Real programs have effects.
- Real programs don’t always terminate.
- How can effects be integrated in pure functional programming?
- How can we specify effects using pure functional programs?
Review: monads in Haskell

\[\text{class Monad } m \text{ where}\]
\[
(\gg\gg) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\]
\[
\text{return} :: a \rightarrow m a
\]

Equations:
\[
\text{return } a \gg\gg f = f a
\]
\[
c \gg\gg \text{return} = c
\]
\[
(c \gg\gg f) \gg\gg g = c \gg\gg \lambda a \rightarrow f a \gg\gg g
\]

Computations are represented by morphisms in the Kleisli category
\[
a \rightarrow_{\text{Kleisli}} b = a \rightarrow m b
\]
newtype State s a = State (s → (a, s))

instance Monad (State s) where
  return a = State (λs → (a, s))
  (State f) >>= g = State (λs → let (a, s′) = f s
    in h s′)
  get :: State s s -- get :: () → Kleisli s
  get = State (λs → (s, s))
  put :: s → State s () -- put :: s → Kleisli ()
  put s = State (λ_ → (() , s))
  evalState :: State s a → s → a
  evalState (State f) s = a where (a, _) = f s
Haskell’s IO monad

instance Monad IO

Stream IO:
  getChar :: IO Char
  putChar :: Char → IO ()

Example:
  echo :: IO ()
  echo = getChar >>= (λc → putChar c) >>= echo
Referential transparency

\[
\text{\textit{dotwice}} :: \text{\textbf{IO}} () \rightarrow \text{\textbf{IO}} ()
\]
\[
\text{\textit{dotwice}} \ p = p \gg p
\]

The two following lines have the same behaviour:

\[
\text{\textit{dotwice}} \ (\text{\textit{putStrLn}} \ \text{"Hello"})
\]
\[
(\text{\textit{putStrLn}} \ \text{"Hello"}) \gg (\text{\textit{putStrLn}} \ \text{"Hello"})
\]
Reasoning about effects

- How to reason about programs with IO? E.g. the implementations of queues using forkIO and MVars.
- In *Tackling the Awkward Squad* Simon Peyton Jones explains the meaning of Haskell with IO by translating it into a process calculus.
- We could use this translation to reason about Haskell’s programs with IO.
Dependent types and IO

• *Insert Epigram Ad* ([www.e-pig.org](http://www.e-pig.org)).
• How do we integrate IO into a language with dependent types.
• The epigram type checker has to evaluate programs appearing in type.
• What should the type checker do if the program `formatHD` appears in a type?
Functional specifications of IO

- Use (pure) functional programming to specify the IO monad.
- Reasoning about IO can be reduced to reasoning about pure programs.
- Dependent types: use functional spec at compile time but execute effects at run time.
- Stealing ideas from Koen Classen, Andy Gordon, Peter Hancock, Graham Hutton, Simon Peyton Jones, Amr Sabry, Toni Setzer, ...
Overview

- Use functional specification to tackle the Awkward Squad
  - Stream IO
  - IORefs
  - Concurrency with MVars
- Discuss the issues arising:
  - Totality
  - Generics
  - Full abstraction
- Run out of time to do:
  - Partiality as an effect
  - Quantum IO
Implementation of Stream IO

```haskell
data IO a =
  GetChar (Char → IO a)
| PutChar Char (IO a)
| Return a

instance Monad IO where
  return = Return
  (Return a) >>= g = g a
  (GetChar f) >>= g = GetChar (λc → f c >>= g)
  (PutChar c a) >>= g = PutChar c (a >>= g)

getChar :: IO Char
getChar = GetChar Return

putChar :: Char → IO ()
putChar c = PutChar c (Return ())
```

Cambridge June 06 – p.12/33
Semantics

data \([a]_b = a : [a]_b \mid []_b\)

\(run :: \text{IO } a \rightarrow [\text{Char}]_\emptyset \rightarrow [\text{Char}]_a\)

\(run \ (\text{Return } a) \ cs = []_a\)

\(run \ (\text{GetChar } f) \ (c : cs) = run \ (f \ c) \ cs\)

\(run \ (\text{PutChar } c \ p) \ cs = c : run \ p \ cs\)
• We have to differentiate between *initial algebra* and *terminal coalgebra* interpretation of data types.

• We could interpret \([a]_b\) as:
  \[
  \mu X. a \times X + b \quad \text{permitting structural recursion, e.g.}
  \]
  \[
  \text{getTip :: } [a]_b \rightarrow b
  \]
  \[
  \text{getTip (_, bs) } = \text{getTip bs}
  \]
  \[
  \text{getTip ([]_b) } = b
  \]

  \[
  \nu X. a \times X + b \quad \text{permitting guarded corecursion.}
  \]
  \[
  \text{repeat :: } a \rightarrow [a]_b
  \]
  \[
  \text{repeat } a = a : \text{repeat } a
  \]

• I will annotate the declaration:
  \[
  \textbf{data } [a]_b = a : [a]_b^\infty \mid []_b
  \]
  to indicate that we mean \(\nu X. a \times X + b\).
How to annotate IO?

```
data IO a =
    GetChar (Char → IO a)
  | PutChar Char (IO a)
  | Return a
```
How to annotate IO!

\[ \text{data } \text{IO} \ a = \]
\[ \quad \text{GetChar} \ (\text{Char} \rightarrow \text{IO} \ a) \]
\[ \quad | \quad \text{PutChar} \ \text{Char} \ (\text{IO}^{\infty} \ a)) \]
\[ \quad | \quad \text{Return} \ a \]

- We interpret this as:

\[ \text{IO} \ a = \nu X. \mu Y. \text{Char} \rightarrow Y + \text{Char} \times X + a \]

- \textit{run} and \textit{copy} are total functions.
- Indeed, any IO performing function which never gets stuck is total.
IORefs

\[\text{newIORef} :: a \rightarrow \text{IO (IORef } a)\]
\[\text{writeIORef} :: \text{IORef } a \rightarrow a \rightarrow \text{IO ()}\]
\[\text{readIORef} :: \text{IORef } a \rightarrow \text{IO } a\]

\textbf{type } \text{Data} = \mathbb{Z}
\textbf{type } \text{Loc} = \mathbb{Z}

\textbf{data } \text{IO } a =
\phantom{=} \text{NewIORef Data (Loc } \rightarrow \text{IO } a\)
| \text{ReadIORef Loc (Data } \rightarrow \text{IO } a\)
| \text{WriteIORef Loc Data (IO } a\)
| \text{Return } a
Mutable state semantics

\[
\text{type } \text{Heap} = \text{Loc} \rightarrow \text{Data} \\
\text{data } \text{Store} = \text{Store}\{\text{free} :: \text{Loc}, \text{heap} :: \text{Heap}\} \\
\text{emptyStore} :: \text{Store} \\
\text{emptyStore} = \text{Store}\{\text{free} = 0\} \\
\text{run} :: \text{IO } a \rightarrow a \\
\text{run } \text{io} = \text{evalState} \left(\text{runState } \text{io}\right) \text{emptyStore} \\
\text{runState} :: \text{IO } a \rightarrow \text{State } \text{Store } a
\]
Generics?

- `IORef` should work with any type.
Use a type class?

```haskell
class Marshall b where
    marshall :: b → Data
    unmarshall :: Data → b

data IO a =
    Return a
    | ∀ b. Marshall b ⇒ NewIORef b (Loc → IO a)
    | ∀ b. Marshall b ⇒ ReadIORef Loc (b → IO a)
    | ∀ b. Marshall b ⇒ WriteIORef Loc b (IO a)

data Data a where
    Z :: Z → Data
    Pair :: Data → Data → Data (a, b)
...

instance Marshall Z

instance (Marshall a, Marshall b) ⇒ Marshall (a, b)
```
Generics

- How can we see that our code is type safe?
- Use a GADT?
  
  ```haskell
data Data a where
  \textbf{Z} :: \textbf{Z} \rightarrow \text{Data} \textbf{Z}
  \textbf{Pair} :: \text{Data} \ a \rightarrow \text{Data} \ b \rightarrow \text{Data} \ (a, b)
  ...
  ```

- But how to implement:
  
  ```haskell
  \text{update} :: \textbf{IORef} \ a \rightarrow \text{Data} \ a
  \rightarrow (\forall \ b . (\textbf{IORef} \ b \rightarrow \text{Data} \ b) \rightarrow (\textbf{IORef} \ b \rightarrow \text{Data} \ b))
  ```

- Use a more expressive type system (e.g. Epigram’s).
Total ?

- We interpret IO as an inductive type.
- \texttt{runState} is total, any function using IO which doesn’t get stuck is total.
- However, \texttt{heap :: \mathbb{Z} \rightarrow Data} is undefined for \( i > \text{free} \).
- We have to convince ourselves, that we never access the heap beyond \text{free}.
- This could be achieved by using dependent types:
  \[
  \textbf{data} \ \text{Store} = \text{Store}\{\text{free :: N}, \text{heap :: Fin free \rightarrow Data}\}
  \]
  where \( \text{Fin n} = \{0, \ldots, n - 1\} \).

Cambridge June 06 – p.22/33
Concurrent Haskell

\[
\text{forkIO} :: \text{IO } a \rightarrow \text{IO ThreadId} \\
\text{newEmptyMVar} :: \text{IO (MVar } a) \\
\text{putMVar} :: \text{MVar } a \rightarrow a \rightarrow \text{IO ()} \\
\text{takeMVar} :: \text{MVar } a \rightarrow \text{IO } a
\]
**Implementation**

\[
\text{type } Data = \mathbb{Z} \\
\text{type } Loc = \mathbb{Z} \\
\text{type } ThreadId = \mathbb{Z} \\
\]

data \( IO \ a \) =

\[
\begin{align*}
\text{Return } a \\
\mid \text{NewEmptyMVar } (Loc \to IO \ a) \\
\mid \text{TakeMVar } Loc \ (Data \to IO \ a) \\
\mid \text{PutMVar } Loc \ Data \ (IO \ a) \\
\mid \forall b . \text{Fork } (IO \ b) \ (ThreadId \to IO \ a)
\end{align*}
\]

**instance Monad \( IO \)**
newtype Scheduler = Scheduler (\mathbb{Z} \rightarrow (\mathbb{Z}, Scheduler))

data ThreadStatus = \forall b . Running (\text{IO} b) | Finished

data Store = Store \{ 
  free :: \text{Loc}, 
  heap :: \text{Loc} \rightarrow \text{Maybe Data}, 
  nextId :: \text{ThreadId}, 
  soup :: \text{ThreadId} \rightarrow \text{ThreadStatus}, 
  scheduler :: \text{Scheduler} \}

initStore :: \text{Scheduler} \rightarrow \text{Store}
initStore s = Store \{ free = 0, nextId = 1, scheduler = s \}

run :: \text{IO} a \rightarrow \text{Scheduler} \rightarrow \text{Maybe} a
run main s = evalState (interleave main) (initStore s)
Implementation

\[ \text{interleave} :: \text{IO } a \rightarrow \text{State Store } (\text{Maybe } a) \]

\text{interleave main} interleaves \text{main} with the threads in \text{soup} depending on scheduler using \text{step}. \text{interleave} returns \text{Nothing}, in case of a deadlock.

\textbf{data} \ \text{Status } a = \text{Stop } a \mid \text{Step } (\text{IO } a) \mid \text{Blocked}

\text{step} :: \text{IO } a \rightarrow \text{State Store } (\text{Status } a)

\text{step thread} attempts to execute one step of \text{thread}.
Non determinism

The type of run

\[ \text{run}^{\text{IO}_c} :: \text{IO} \ a \rightarrow \text{Scheduler} \rightarrow \text{Maybe} \ a \]

is too intensional, because in practice we view the scheduler as externally given.

We define a simulation preorder on \( \text{Scheduler} \rightarrow \text{Maybe} \ a \):

\[ f \sqsubseteq g \iff \forall s :: \text{Scheduler}. \exists c \ s' : \text{Scheduler}. f \ s = g \ s' \]

and bisimulation:

\[ f \simeq g \iff f \sqsubseteq g \wedge g \sqsubseteq f \]
Total?

- **IO** is inductively defined, ... 
- hence we have no infinitely running processes (yet)! 
- *run* is total, and all concurrent programs which don’t get stuck are total. 
- We assume that the *MVars* are private to our program.
Combining effects

We can combine StreamIO and concurrency:

```
data IO a =
    Return a |
    GetChar (Char -> IO a) |
    PutChar Char (IO^\infty a)) |
    NewEmptyMVar (Loc -> IO a) |
    TakeMVar Loc (Data -> IO a) |
    PutMVar Loc Data (IO a) |
    \forall b . Fork (IO b) (ThreadId -> IO a)
```

The type of `run` becomes:

```
run :: IO a -> Scheduler -> [Char] a -> [Char](Maybe a)
```

We can have infinite processes now.
Full abstraction

- We would like to identify elements of \texttt{IO} a which show the same observable behaviour.
- However, we cannot identify programs which are given the same behaviour under \textit{run}. Why not?
- An implementation of \texttt{IO} a has to:
  - not identify any programs which can be separated by \textit{run}.
  - support an algebra defining all functions in the API.
- We say an implementation of \texttt{IO} a is fully abstract, if the algebra is maximal.
• The definition of Stream IO is almost fully abstract. We can identify \( \text{GetChar} f \) and \( \text{Return} a \), iff for all \( c :: \text{Char} \)
\[
\text{run} :: \text{IO} a \rightarrow [\text{Char}]_0 \rightarrow [\text{Char}]([\text{Char}]_0 , a)
\]
This may be a bug, maybe \( \text{run} \) should return the rest of the input:

• Stateful can be easily given an almost fully abstract semantics by using

\[
\text{type IO } a = \text{State Store } a
\]
directly.

\textit{see work by Andy Pitts, Ian Stark and others how to fix the almost...}

• Full abstraction for concurrency?
Left out:

- the partiality monad *Partial* which allows us to express partial (i.e. potentially diverging functions) as elements of \( a \rightarrow Partial \ b \).

  *joint work with Venanzio Capretta and Tarmo Uustalu.*

- the quantum IO monad,
Conclusions and further work

- Need examples, apply semantics to verify effectful programs.
- Combine effects using coproducts or monad transformers.
- Integrate into Epigram,
  Goal: specify and implement real programs in Epigram.
- Exploit dependent types to structure effects, e.g. regions.
- Discuss: difference between internal effects (e.g. IORefs) and IO (e.g. streams).
- Obligation: show that the specified semantics agrees with the actual implementation.
  compiler correctness.