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

$$\begin{array}{l}\textbf{class } \mathit{Monad}\ m \textbf{ where} \\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b \\ \quad \mathit{return} :: a \to m\ a\end{array}$$

Equations:

$$\begin{array}{rcl}\mathit{return}\ a \gg\!= f &=& f\ a \\ c \gg\!= \mathit{return} &=& c \\ (c \gg\!= f) \gg\!= g &=& c \gg\!= \lambda a \to f\ a \gg\!= g\end{array}$$

Computations are represented by morphisms in the Kleisli category

$$a \to_{\text{Kleisli}} b = a \to m\ b$$

The state monad

$\textbf{newtype}\ State\ s\ a = State\ (s \to (a, s))$

$\textbf{instance}\ Monad\ (State\ s)\ \textbf{where}$

$\quad return\ a = State\ (\lambda s \to (a, s))$

$\quad (State\ f) \gg\!= g = State\ (\lambda s \to \textbf{let}\ (a, s') = f\ s$

$\qquad (State\ h) = g\ a$

$\qquad \textbf{in}\ h\ s')$

$get :: State\ s\ s \quad \text{-- } get :: () \to_{\text{Kleisli}} s$

$get = State\ (\lambda s \to (s, s))$

$put :: s \to State\ s\ () \quad \text{-- } put :: s \to_{\text{Kleisli}} ()$

$put\ s = State\ (\lambda\_ \to ((), s))$

$evalState :: State\ s\ a \to s \to a$

$evalState\ (State\ f)\ s = a\ \textbf{where}\ (a, \_) = f\ s$

Haskell's IO monad

$$\textbf{instance}\ \mathit{Monad}\ \mathbf{IO}$$

Stream IO:

$$\mathit{getChar} :: \mathbf{IO}\ \mathit{Char}$$

$$\mathit{putChar} :: \mathit{Char} \to \mathbf{IO}\ ()$$

Example:

$$\mathit{echo} :: \mathbf{IO}\ ()$$

$$\mathit{echo} = \mathit{getChar} \gg\!= (\lambda c \to \mathit{putChar}\ c) \gg \mathit{echo}$$

Referential transparency

$$dotwice :: \mathbf{IO}\ () \to \mathbf{IO}\ ()$$
$$dotwice\ p = p \gg p$$

The two following lines have the same behaviour:

$$dotwice\ (putStrLn\ \texttt{"Hello"})$$
$$(putStrLn\ \texttt{"Hello"}) \gg (putStrLn\ \texttt{"Hello"})$$

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`).
- 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

$$
\begin{aligned}
&\textbf{data IO } a = \\
&\qquad GetChar\ (Char \to \textbf{IO}\ a) \\
&\quad\ \mid PutChar\ Char\ (\textbf{IO}\ a) \\
&\quad\ \mid Return\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } Monad\ \textbf{IO where} \\
&\quad return = Return \\
&\quad (Return\ a) \gg\!= g \qquad\ = g\ a \\
&\quad (GetChar\ f) \gg\!= g \quad\ = GetChar\ (\lambda c \to f\ c \gg\!= g) \\
&\quad (PutChar\ c\ a) \gg\!= g = PutChar\ c\ (a \gg\!= g)
\end{aligned}
$$

$$
\begin{aligned}
getChar &:: \textbf{IO}\ Char \\
getChar &= GetChar\ Return
\end{aligned}
$$

$$
\begin{aligned}
putChar &:: Char \to \textbf{IO}\ () \\
putChar\ c &= PutChar\ c\ (Return\ ())
\end{aligned}
$$

Semantics

$$\mathbf{data}\ [a]_b = a : [a]_b \mid [\,]_b$$

$$\begin{aligned}
&run :: \mathbf{IO}\ a \rightarrow [Char]_\emptyset \rightarrow [Char]_a\\
&run\ (Return\ a) && cs && = [\,]_a\\
&run\ (GetChar\ f) && (c : cs) && = run\ (f\ c)\ cs\\
&run\ (PutChar\ c\ p) && cs && = c : run\ p\ cs
\end{aligned}$$

Total ?

- We have to differentiate between *initial algebra* and *terminal coalgebra* interpretation of data types.
- We could interpret $[\,a\,]_b$ as:

  $\mu X.a \times X + b$ permitting structural recursion, e.g.

  $$\begin{aligned} &getTip :: [\,a\,]_b \longrightarrow b \\ &getTip\ (\_ : bs) = getTip\ bs \\ &getTip\ ([\,]_b) \quad\ = b \end{aligned}$$

  $\nu X.a \times X + b$ permitting guarded corecursion.

  $$\begin{aligned} &repeat :: a \longrightarrow [\,a\,]_b \\ &repeat\ a = a : repeat\ a \end{aligned}$$

- I will annotate the declaration:

  $$\mathbf{data}\ [\,a\,]_b = a : [\,a\,]_b{}^{\infty} \mid [\,]_b$$

  to indicate that we mean $\nu X.a \times X + b$.

How to annotate IO?

$$
\begin{aligned}
&\mathbf{data}\ \mathbf{IO}\ a = \\
&\qquad \mathit{GetChar}\ (\mathit{Char} \rightarrow \mathbf{IO}\ a) \\
&\quad \mid \mathit{PutChar}\ \mathit{Char}\ (\mathbf{IO}\ a) \\
&\quad \mid \mathit{Return}\ a
\end{aligned}
$$

How to annotate IO!

$$
\begin{aligned}
&\mathbf{data}\ \mathbf{IO}\ a = \\
&\qquad\quad GetChar\ (Char \to \mathbf{IO}\ a) \\
&\qquad\ \mid PutChar\ Char\ (\mathbf{IO}^{\infty}\ a)) \\
&\qquad\ \mid Return\ a
\end{aligned}
$$

- We interpret this as:

$$\mathbf{IO}\ a = \nu X.\mu Y.Char \to Y + Char \times X + a$$

- *run* and *copy* are total functions.
- Indeed, any IO performing function which never gets stuck is total.

IORefs

$$\begin{aligned}
\mathit{newIORef} &:: a \to \mathbf{IO}\ (\mathbf{IORef}\ a) \\
\mathit{writeIORef} &:: \mathbf{IORef}\ a \to a \to \mathbf{IO}\ () \\
\mathit{readIORef} &:: \mathbf{IORef}\ a \to \mathbf{IO}\ a
\end{aligned}$$

$$\begin{aligned}
&\mathbf{type}\ \mathit{Data} = \mathbb{Z} \\
&\mathbf{type}\ \mathit{Loc} \quad = \mathbb{Z}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{data}\ \mathbf{IO}\ a = \\
&\qquad \mathit{NewIORef}\ \ \mathit{Data}\ (\mathit{Loc} \to \mathbf{IO}\ a) \\
&\quad\ \mid \mathit{ReadIORef}\ \ \mathit{Loc}\ (\mathit{Data} \to \mathbf{IO}\ a) \\
&\quad\ \mid \mathit{WriteIORef}\ \ \mathit{Loc}\ \mathit{Data}\ (\mathbf{IO}\ a) \\
&\quad\ \mid \mathit{Return}\ \ a
\end{aligned}$$

Mutable state semantics

$$\textbf{type } Heap = Loc \to Data$$

$$\textbf{data } Store = Store\{free :: Loc, heap :: Heap\}$$

$$emptyStore :: Store$$

$$emptyStore = Store\{free = 0\}$$

$$run :: \quad \mathbf{IO}\ a \to a$$

$$run\ io = \quad evalState\ (runState\ io)\ emptyStore$$

$$runState :: \mathbf{IO}\ a \to State\ Store\ a$$

Generics ?

- **IORef** should work with any type.

Use a type class?

$\mathbf{class}\ Marshall\ b\ \mathbf{where}$

$\quad marshall :: b \rightarrow Data$

$\quad unmarshall :: Data \rightarrow b$

$\mathbf{data}\ \mathbf{IO}\ a =$

$\qquad Return\ \ a$

$\quad \mid \forall b\ .\ Marshall\ b \Rightarrow NewIORef\ \ b\ (Loc \rightarrow \mathbf{IO}\ \ a)$

$\quad \mid \forall b\ .\ Marshall\ b \Rightarrow ReadIORef\ \ Loc\ (b \rightarrow \mathbf{IO}\ \ a)$

$\quad \mid \forall b\ .\ Marshall\ b \Rightarrow WriteIORef\ \ Loc\ b\ (\mathbf{IO}\ \ a)$

$\mathbf{data}\ Data\ a\ \mathbf{where}$

$\quad \mathbb{Z} :: \mathbb{Z} \rightarrow Data$

$\quad Pair :: Data \rightarrow Data \rightarrow Data\ (a, b)$

$\quad \ldots$

$\quad \mathbf{instance}\ Marshall\ \mathbb{Z}$

$\quad \mathbf{instance}\ (Marshall\ \ a, Marshall\ \ b) \Rightarrow Marshall\ \ (a, b)$

Generics

- How can we *see* that our code is type safe?
- Use a GADT?

$$\begin{aligned}&\mathbf{data}\ Data\ a\ \mathbf{where}\\&\quad \mathbb{Z} :: \mathbb{Z} \rightarrow Data\ \mathbb{Z}\\&\quad Pair :: Data\ a \rightarrow Data\ b \rightarrow Data\ (a, b)\\&\quad \ldots\end{aligned}$$

- But how to implement:

$$\begin{aligned}&update :: \mathbf{IORef}\ a \rightarrow Data\ a\\&\quad \rightarrow (\forall\, b\ .\ (\mathbf{IORef}\ b \rightarrow Data\ b) \rightarrow (\mathbf{IORef}\ b \rightarrow Data\ b))\end{aligned}$$

- Use a more expressive type system (e.g. Epigram's).

Total ?

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

Concurrent Haskell

$$
\begin{aligned}
\mathit{forkIO} &:: \mathbf{IO}\ a \rightarrow \mathbf{IO}\ \mathit{ThreadId} \\
\mathit{newEmptyMVar} &:: \mathbf{IO}\ (\mathit{MVar}\ a) \\
\mathit{putMVar} &:: \mathit{MVar}\ a \rightarrow a \rightarrow \mathbf{IO}\ () \\
\mathit{takeMVar} &:: \mathit{MVar}\ a \rightarrow \mathbf{IO}\ a
\end{aligned}
$$

Implementation

$$
\begin{aligned}
&\textbf{type } \mathit{Data} &&= \mathbb{Z} \\
&\textbf{type } \mathit{Loc} &&= \mathbb{Z} \\
&\textbf{type } \mathit{ThreadId} &&= \mathbb{Z}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data IO } a = \\
&\quad\quad \mathit{Return}\ a \\
&\quad\ \mid \mathit{NewEmptyMVar}\ (\mathit{Loc} \rightarrow \textbf{IO}\ a) \\
&\quad\ \mid \mathit{TakeMVar}\ \mathit{Loc}\ (\mathit{Data} \rightarrow \textbf{IO}\ a) \\
&\quad\ \mid \mathit{PutMVar}\ \mathit{Loc}\ \mathit{Data}\ (\textbf{IO}\ a) \\
&\quad\ \mid \forall\, b\ .\ \mathit{Fork}\ (\textbf{IO}\ b)\ (\mathit{ThreadId} \rightarrow \textbf{IO}\ a)
\end{aligned}
$$

$$\textbf{instance } \mathit{Monad}\ \textbf{IO}$$

Implementation

$$\textbf{newtype}\ \mathit{Scheduler} = \mathit{Scheduler}\ (\mathbb{Z} \to (\mathbb{Z}, \mathit{Scheduler}))$$

$$\textbf{data}\ \mathit{ThreadStatus} = \forall\, b\ .\ \mathit{Running}\ (\mathbf{IO}\ b) \mid \mathit{Finished}$$

$$\begin{aligned}\textbf{data}\ \mathit{Store} = \mathit{Store}\{&\mathit{free} :: \mathit{Loc},\\ &\mathit{heap} :: \mathit{Loc} \to \mathit{Maybe}\ \mathit{Data},\\ &\mathit{nextId} :: \mathit{ThreadId},\\ &\mathit{soup} :: \mathit{ThreadId} \to \mathit{ThreadStatus},\\ &\mathit{scheduler} :: \mathit{Scheduler}\}\end{aligned}$$

$$\begin{aligned}&\mathit{initStore} :: \mathit{Scheduler} \to \mathit{Store}\\ &\mathit{initStore}\ s = \mathit{Store}\{\mathit{free} = 0, \mathit{nextId} = 1, \mathit{scheduler} = s\}\end{aligned}$$

$$\begin{aligned}&\mathit{run} :: \mathbf{IO}\ a \to \mathit{Scheduler} \to \mathit{Maybe}\ a\\ &\mathit{run}\ \mathit{main}\ s = \mathit{evalState}\ (\mathit{interleave}\ \mathit{main})\ (\mathit{initStore}\ s)\end{aligned}$$

Implementation

$$interleave :: \mathbf{IO}\ a \rightarrow State\ Store\ (Maybe\ a)$$

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

$$\mathbf{data}\ Status\ a = Stop\ a \mid Step\ (\mathbf{IO}\ a) \mid Blocked$$

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

$step\ thread$ attempts to execute one step of $thread$.

Non determinism

The type of run

$$runIO_c :: \mathbf{IO}\ a \to Scheduler \to Maybe\ a$$

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

We define a simulation preorder on $Scheduler \to Maybe\ a$:

$$f \sqsubseteq g \iff \forall s :: Scheduler.\exists^c s' : Scheduler.f\ s = g\ s'$$

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:

$$
\begin{aligned}
&\mathbf{data}\ \mathbf{IO}\ a = \\
&\qquad \mathit{Return}\ a \\
&\quad\ \mid \mathit{GetChar}\ (\mathit{Char} \to \mathbf{IO}\ a) \\
&\quad\ \mid \mathit{PutChar}\ \mathit{Char}\ (\mathbf{IO}^{\infty}\ a)) \\
&\quad\ \mid \mathit{NewEmptyMVar}\ (\mathit{Loc} \to \mathbf{IO}\ a) \\
&\quad\ \mid \mathit{TakeMVar}\ \mathit{Loc}\ (\mathit{Data} \to \mathbf{IO}\ a) \\
&\quad\ \mid \mathit{PutMVar}\ \mathit{Loc}\ \mathit{Data}\ (\mathbf{IO}\ a) \\
&\quad\ \mid \forall\, b\ .\ \mathit{Fork}\ (\mathbf{IO}\ b)\ (\mathit{ThreadId} \to \mathbf{IO}\ a)
\end{aligned}
$$

The type of run becomes:

$$run :: \mathbf{IO}\ a \to \mathit{Scheduler} \to [\mathit{Char}]_a \to [\mathit{Char}]_{(Maybe\ a)}$$

We can have infinite processes now.

Full abstraction

- We would like to identify elements of **IO** a which show the same observable behaviour.
- However, we cannot identify programs which are given the same behaviour under run.
  Why not?
- An implementation of **IO** a has to:
  - not identify any programs which can be separated by run.
  - support an an algebra defining all functions in the API.
- We say an implementation of **IO** a is fully abstract, if the algebra is maximal.

Full abstraction

- The definition of Stream IO is almost fully abstract.
  We can identify $GetChar\ f$ and $Return\ a$, iff for all $c :: Char$
  $f\ c = Return\ a$
  This may be a bug, maybe run should return the rest of the input:
  $$run :: \mathbf{IO}\ a \rightarrow [\,Char\,]_{\emptyset} \rightarrow [\,Char\,]_{([\,Char\,]_{\emptyset}, a)}$$
- Stateful can be easily given an almost fully abstract semantics by using
  $\mathbf{type\ IO}\ a = State\ Store\ a$
  directly.
  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 $a \longrightarrow Partial\ b$.
  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.