The Beauty and the Beast: A Happy End?

based on joint work with Wouter Swierstra

Thorsten Altenkirch

School of Computer Science
University of Nottingham

December 15, 2007

Overcoming the ASCII-greek dichotomy

- Programs (ASCII) vs. Maths (greek)
- Programming **is** constructive Mathematics.
- No need for mathematical models of (pure) functional programs.
- **Type Theory**: No difference between a *mathematical function* and a function in programming.

Real World?

- Real Programs are not pure functions.
- Real programs have effects.
- Real programs don’t always terminate.
- How can effects be integrated in Type Theory?

The Awkward Squad

- Simon Peyton Jones (2000) in Marktoberdorf:
  Tackling the awkward squad
- Some Squad members:
  1. Stream I/O (`getChar`, `putChar`)
  2. Updatable references (`IOVar`)
  3. Concurrency (`forkIO`, `MVar`)
- Approach: Translate impure Haskell (ASCII) into a process calculus (greek).

Beauty in the Beast

- Functional specifications of effects.
- Use pure Haskell to explain impure Haskell.
- Takes place in a total fragment of Haskell (Ask).
- *Quick check* impure programs.
- Warm up for Effects in Type Theory
  Haskell for the lazy Type Theoretician.
- See our Haskell Workshop (2007) paper.

Implementation of Stream IO

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

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

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

Semantics

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

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

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.

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

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

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

- 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?

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

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

- We interpret this as:

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

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

Pipes and switches (with Varmo Vene and Tarmo Uustalu)

$$
\begin{array}{l}
\textbf{data}\ IO\ i\ o\ a = \\
\quad\quad Get\ (i \to IO\ i\ o\ a) \\
\quad\ \mid Put\ o\ (IO\ i\ o\ a)^{\infty} \\
\quad\ \mid Return\ a
\end{array}
$$

$$
(\ggg) :: IO\ i\ r\ a \to IO\ r\ o\ a \to IO\ i\ o\ a
$$

$$
\begin{array}{llll}
Return\ a & \ggg q & & = Return\ a \\
Get\ f & \ggg q & & = Get\ (\lambda i \to f\ i \ggg q) \\
Put\ h\ p & \ggg & Return\ a & = Return\ a \\
Put\ h\ p & \ggg & Get\ f & = p \ggg f\ h \\
Put\ h\ p & \ggg & Put\ o\ q & = Put\ o\ (Put\ h\ p \ggg q)
\end{array}
$$

Arrows?

- Conjecture: This is an **arrow** and a monad.
- Without `Return`: Example of an Arrow in John Hughes' paper.
- Wouter: It is not an arrow (even without `Return`).
- There seems to be no easy fix.

IORefs

$$
\begin{array}{l}
\textbf{type } \mathit{Data} = \mathit{Int} \\
\textbf{type } \mathit{Loc} \;\; = \mathit{Int} \\
\textbf{data } \mathit{IO}\ a = \\
\quad\;\; \mathit{NewIORef}\ \mathit{Data}\ (\mathit{Loc} \rightarrow \mathit{IO}\ a) \\
\quad \mid \mathit{ReadIORef}\ \mathit{Loc}\ (\mathit{Data} \rightarrow \mathit{IO}\ a) \\
\quad \mid \mathit{WriteIORef}\ \mathit{Loc}\ \mathit{Data}\ (\mathit{IO}\ a) \\
\quad \mid \mathit{Return}\ a
\end{array}
$$

Mutable state semantics

$$\textbf{type}\ \mathit{Heap} = \mathit{Loc} \rightarrow \mathit{Data}$$
$$\textbf{data}\ \mathit{Store} = \mathit{Store}\{\mathit{free} :: \mathit{Loc}, \mathit{heap} :: \mathit{Heap}\}$$

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

$$\mathit{run} :: \mathit{IO}\ a \rightarrow a$$
$$\mathit{run}\ \mathit{io} = \mathit{evalState}\ (\mathit{runState}\ \mathit{io})\ \mathit{emptyStore}$$

$$\mathit{runState} :: \mathit{IO}\ a \rightarrow \mathit{State}\ \mathit{Store}\ a$$

Issues

- `Heap` is partial, we could access an unallocated memory location.
- We want to store different datatypes. . .
- Memory access should be type-safe.
- See next talk by Wouter.
- Other examples: Concurrent Haskell, Quantum IO, . . .
- Do we need 2 levels (*IO*,*run*)?

The Partiality Monad with Venanzio Capretta and Tarmo Uustalu

- So far all operations were total.
- Partiality is an effect: abstraction of time in the real world.
- Give a functional specification of partiality.
- We first define the delay monad $D :: * \longrightarrow *$ and then partiality $P\ a = D\ a/\simeq$ as a quotient.

The Delay monad

$$\textbf{data}\ D\ a = Now\ a \mid Later\ (D\ a)^{\infty}$$

$$\textbf{instance}\ Monad\ D\ \textbf{where}$$

$$return = Now$$

$$\begin{aligned} Now\ a \gg\!= k &= k\ a \\ Later\ d \gg\!= k &= Later\ (d \gg\!= k) \end{aligned}$$

$$\bot :: D\ a$$

$$\bot = Later\ \bot$$

Fixpoints with Delay

$$
\begin{array}{l}
\mathit{rec} :: ((a \to D\ b) \to (a \to D\ b)) \to a \to D\ b \\
\mathit{rec}\ \mathit{phi}\ a = \mathit{aux}\ (\lambda\_ \to \bot) \\
\quad \textbf{where}\ \mathit{aux} :: (a \to D\ b) \to D\ b \\
\qquad \mathit{aux}\ k = \mathit{race}\ (k\ a)\ (\mathit{Later}\ (\mathit{aux}\ (\mathit{phi}\ k))) \\
\mathit{race} :: (D\ a) \to (D\ a) \to (D\ a)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{race}\ (\mathit{Now}\ a) & \_ & = \mathit{Now}\ a \\
\mathit{race}\ (\mathit{Later}\ \_) & (\mathit{Now}\ a) & = \mathit{Now}\ a \\
\mathit{race}\ (\mathit{Later}\ d) & (\mathit{Later}\ d') & = \mathit{Later}\ (\mathit{race}\ d\ d')
\end{array}
$$

From Delay to Partial

- D is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$
- We define

$$P\ a = D\ a / \simeq$$

where $\simeq \subseteq D\ a \times D\ a$ identifies values with different finite delay.

Defining $\simeq$

- $(\downarrow) \subseteq D\ a \times a$ is defined inductively.

$$\frac{}{\mathit{Now}\ a \downarrow a} \qquad \frac{d \downarrow a}{\mathit{Later}\ d \downarrow a}$$

-

$$\begin{aligned} \sqsubseteq &\subseteq D\ a \times D\ a \\ d \sqsubseteq d' &= \forall a. d \downarrow a \implies d' \downarrow a \end{aligned}$$

-

$$\begin{aligned} \simeq &\subseteq D\ a \times D\ a \\ d \simeq d' &= d \sqsubseteq d' \wedge d' \sqsubseteq d \end{aligned}$$

Deja vu ?

- Constructive Domain Theory!
- $P\ a = a_\perp$
- Note that constructively

$$a_\perp \neq a + \{\perp\}$$

because we cannot observe non-termination.

- $P\ a$ and hence $a \to P\ b$ are ωCPOs.
- $rec\ f = \sqcup_{i \in Nat} f^i \perp$ we construct $\sqcup$ in $a \to P\ b$.
- Need that f is ω-continous.

Modalities vs IO

- Different kind of effects:

  Runtime system
  - Stream IO
  - References
  - Concurrency
  - Quantum IO

  Modality
  - Errors (e.g. *Maybe*)
  - Partiality
  - Nondeterminism (*Scheduler* $\rightarrow a$).
  - Probability ($a \rightarrow \mathbb{R}^{+}$)

Effects, foundationally

- We give functional specifications of effects.
- This way effects can be integrated into Type Theory without extending Type Theory.
- Can we do this for *Hoare Type Theory*?

Greg Morrisett's TLCA 07 lecture

TLCA_June_2007.ppt

Defining ST in Coq?

Can try to define:

```
ST P A Q :=
  {i:heap|P h} -> {f:heap; x:A |Q x i h}
```

but then you sacrifice:

- recursive (diverging) computations
- non-deterministic computations
- code that stores computations in the heap

So we will do something different.

27 June 2007 39

Loose ends

- Combine effects using coproducts or monad transformers e.g. Concurrency + Streams.
  see Wouter's paper *Data types á la carte*
- Difference between internal effects (e.g. IORefs) and proper IO (e.g. streams)
  Exploit dependent types to structure effects, e.g. regions.
- Obligation: show that the specified semantics agrees with the actual implementation.
  Translate high level effects into low level effects?
- Interpretation of functions in constructive logic
  lawless sequences because we have access to the real world.