I/O in a Dependently Typed Programming Language

Wouter Swierstra
TYPES 2007
Polymorphic lambda calculus with data types
Polymorphic lambda calculus with data types

Associated types

Rank-n types

Functional dependencies

Multiparameter type classes

Generalized algebraic data types

Impredicativity
Proving with dependent types
Programming with dependent types
How to deal with effects?
Impurity

Implicit effects:

\[
\text{launchMissiles} : \text{Unit} \\
\text{launchMissiles} \rightarrow_{\beta} ()
\]

But this is pretty dangerous!
Conversion rule

\[
\begin{align*}
\Gamma & \vdash t : \sigma \\
\sigma & \equiv_{\beta} \tau \\
\Gamma & \vdash t : \tau
\end{align*}
\]

\[
\begin{align*}
p : T(launchMissiles) & \quad launchMissiles \rightarrow_{\beta} () \\
p & \quad T()
\end{align*}
\]
Purity

• We must avoid triggering effects statically.
• Use primitive functions with no associated operational behaviour.

\[
\text{putChar} : \text{Char} \rightarrow \text{IO ()} \\
\text{getChar} : \text{IO Char}
\]

• Placeholders for the “real” functions
Monadic I/O in Haskell

Combine effects using the usual monadic operations:

return : a -> IO a

>>= : IO a -> (a->IO b) -> IO b
Unsatisfactory

- This may be safe, but is it enough?
- We want to reason about our code.
- We don’t have a definition of `putChar` or `getChar`.
- We can’t prove anything about I/O functions.
Defining Teletype I/O

```haskell
data Teletype a =
    PutChar Char (Teletype a)
  | GetChar (Char -> Teletype a)
  | Return a
```

- Teletype is a monad!
Defined functions

putChar : Char -> Teletype ()
putChar c =
    PutChar c (Return ())
getChar : Teletype Char
getChar = GetChar Return
What does it mean?

data Output a =
  | Finish a
  | Print Char (Output a)
run : Teletype a
  -> Stream Char
  -> Output a
So what?

• We have defined our own version of `getChar` and `putChar`.

• **We have meaningful placeholders.**

• A compiler should replace our definitions with appropriate calls to C functions...

  ... provided we have given an accurate description of how these functions behave.
Reasoning about effects

We can now prove:

```
echo =
  getChar
  >>= putChar
  >>= \( () \rightarrow \text{echo} \\
```

copies the input stream to the output.
A refinement...

• We would like to allow infinite streams of output:

  \begin{verbatim}
  printAs = putChar ‘a’
  >>= \x -> printAs
  \end{verbatim}

• Teletype should be coinductive.

• But what about:

  \begin{verbatim}
  sink = getChar >>= sink
  \end{verbatim}
Eating

• We need a mixed inductive-coinductive definition:

$$\nu X. \mu Y. \text{Char} \times X + Y^{\text{Char}} + A$$

• See recent work by Peter Hancock.
What else?

- We can give similar definitions for many other effects:
  -Mutable state
  -Concurrency
  -Software Transactional Memory
  -...
Mutable state

data State a =
  NewRef Data (Loc -> State a)
  | WriteRef Data Loc (State a)
  | ReadRef Loc (Data -> State a)
  | Return a

data Loc = Nat

data Data = Nat
Semantics

runState :
  State a -> Store -> (a,Store)

data Store =
  Store Loc (Loc -> Data)

What’s the initial store?

Why can references only store integers?
A better definition...

- We can define heterogeneous, well-scoped, well-typed references.
- The definition is a little bit tricky...
Heaps

Postulate a universe $U$...

\begin{verbatim}
data Shape = List U
data Heap : Shape -> Set
  | empty : Heap []
  | alloc : el a -> Heap s
       -> Heap (a :: s)
\end{verbatim}
The State type

```haskell
data State (A : Set) :
  Shape -> Shape -> Set where
  ...
```
Running code

run : Heap s
    -> State A s t
    -> (A, Heap t)
Problems

- Is this still a monad?
- Need explicit “weakening” of references.
- The devil is in the details.
The last slide

• Check out the Haskell library:
  www.cs.nott.ac.uk/~wss/repos/IOSpec

• Submitted ICFP paper on my homepage.

• Future work:
  • Combining different effects
  • Precise and total run functions