A Functional Specification of Effects

Wouter Swierstra
joint work with Thorsten Altenkirch

3/12/2007

How can we write reliable software?

Static typing

- Types can give a "partial proof of correctness."
- For example:

```
div : Int -> Int -> Int
```

- We can prevent certain illegitimate calls to div, such as `div True 2`;
- ... but what about `div 3 0` ?

Dependent types

- If we want to make sure that division never goes wrong, we need stronger types.
- A better type for division would be:

```
Int -> {x : Int | x != 0} -> Int
```

- Now the *value* `x` appears in a *type*.

Status quo

- Dependent types form the basis of many *theorem proving tools*, such as Coq.
- Coq has
  - a rich type theory ~ propositions;

```
forall p : Prop, p -> p
```

  - a simple lambda calculus ~ proofs.

```
fun x => x
```

Curry-Howard Isomorphism

- Coq has
  - a rich type system ~ types;

```
forall a : Set, a -> a
```

  - a simple lambda calculus ~ programs.

```
fun x => x
```

Example

- We can implement *stacks* as a list;
- and add functions to manipulate stacks;
- prove properties of our implementation;
- extract code to Haskell or ML.

Limitations

- Programs in Coq must be pure and total:
  - must terminate on all possible inputs;
  - no mutable state, I/O, etc.
- Great for formalizing constructive mathematics.
- What about programming *queues* using mutable references?

Real programs

- Real programs tend to:
  - diverge;
  - throw exceptions;
  - use concurrency;
  - interact with the user;
  - use mutable state...

How can we incorporate such effects in a dependently typed programming language?

Haskell

- Haskell is a functional language with a careful treatment of I/O.
- All effects are encapsulated in a **monad**;
- This determines a clear evaluation order – Haskell is a *non-strict* language.

Monads: motivation

- What does the following Haskell program do?

```
[print "Hello", print "World"]
```

- In Haskell, the print statements are not immediately evaluated.
- Monads make the evaluation order explicit.

Monads: top-down

- The `main` function that gets executed when you run a Haskell program has type:

  ```
  main :: IO a
  ```

- It does some I/O,
- and returns a value of type `a`.

Monads: bind

- To sequence two effectful computations, we could use a "semi-colon" operation:

  ```
  >> :: IO a -> IO b -> IO b
  ```

- But now the result of the first computation always gets discarded.

Monads: bind

- A better choice is:

```
IO a -> (a -> IO b) -> IO b
```

- This feeds the result of the first computation to the second one.

Monads: return

- Sometimes we want to mix I/O interactions and pure computations.

```
return :: a -> IO a
```

- There's no function going the other way!

Monads: example

- In Haskell, you have built-in functions that perform I/O.

```
getChar :: IO Char
```

```
putChar :: Char -> IO ()
```

- Using the monadic operators you can combine them to form complex computations.

Example: echo

- For example, you can write an echo function:

```
echo :: IO ()
echo = getChar >>= \c ->
       putChar c >>= \() ->
       echo
```

Monads: generally

- Any functor with a bind and return operation (subject to certain laws) is a monad.
- Haskell supports special syntax for programing with monads.
- “Programmable semi-colon”

Will this do?

Reasoning

- Reasoning about **pure** functional programs is really easy:
  - structural induction;
  - expand definitions.
- This is essentially what we can do using proof assistants such as Coq.
- But what about the **impure** ones?

Echo revisited

- We would like to reason about how our echo function behaves...
- But functions like `getChar` don't have a pure definition.
- Instead, it calls a C library that does all the dirty work.

Now what?

- We could break out a semantics textbook, hope to find some useful semantics that correspond to how Haskell behaves, and do a pen and paper proof.

Now what?

- We could break out a semantics textbook, hope to find some useful semantics that correspond to how Haskell behaves, and do a pen and paper proof.

  ... but that's not how we're going to do it.

Idea

- The rest of this talk will outline one idea:

  a purely functional

  specification of effects

  - can help you reason about your code;
  - opens the door to "verified effectful programs".

Outline

- We'll build a monad capturing the operations we want to specify;
- Add functions to build computations in this monad;
- and assign meaning to these computations.

Another monad...

```
data IO a where
  Return :: a -> IO a
| Put :: Char -> IO a -> IO a
| Get ::(Char -> IO a) -> IO a
```

Building computations

We introduce some helper functions to make it easier to write computations:

```
getChar :: IO Char

getChar = Get Return

putChar :: Char -> IO ()

putChar c = Put c (Return ())
```

Result

- What should the result of a computation be?

```
data Output =
      Finish a
    | Print Char (Output a)
    | Read (Output a)
```

Executing computations

```
run :: IO a ->
  Stream Char -> Output a
run input (Return x) = Finish x
run (i:is) (Get g) = run is (g i)
run is (Put c io) =
  Print c (run is io)
```

Why bother?

- So we’ve written quite a bit of code, what does this buy us?
- We can write specifications of impure computations;
- and show that our impure computations meet their spec.

Example: echo

- We can specify the behaviour we expect our echo function to have:

```
copy :: Stream Char -> Output ()
copy (i:is) =
  Read (Print i (copy is))
```

- And we can prove once and for all:

```
run echo is = copy is
```

A coinductive aside

- Our IO data type is actually mixed inductive-coinductive:
  - $\nu X \,.\, \mu Y \,.\, Y^C \;+\; X \times C \;+\; A$
- We can only consume finite input from the user, before producing (potentially infinite) output to the screen.
- Note: run and copy are still total.

So what?

- If we write our specifications in Coq, this proof can be machine-verified
- If we program in Haskell we can port all the debugging and testing technology on pure programs to work on these pure specifications.

Is that all?

- We have written similar semantics for:
  - mutable state;
  - concurrency;
  - STM;
  - non-termination;
  - distributed arrays;

Outline

- We'll build a monad capturing the operations we want to specify;
- Add functions to build computations in this monad;
- and assign meaning to these computations.

Mutable state

```
data Ref = Int

data IO a where

   Return :: a -> IO a

 | Read :: Ref -> (Int -> IO a) -> IO a

 | Write :: Ref -> Int -> IO a -> IO a

 | New :: Int -> (Ref -> IO a) -> IO a
```

Mutable state II

```
new :: Int -> IO Ref

write :: Int -> Ref -> IO ()

read :: Ref -> IO Int
```

Describing memory

```
data Heap = Ref -> Int
type Store = (Heap, Int)

emptyStore = (undefined, 0)
```

Execution

- Our semantics now have the following type:

```
IO a -> Store -> (a,Store)
```

- but not everything in the garden is rosy...

What’s wrong?

- Our run function is not **total**...
- What will happen when we access unallocated memory?
- We have only managed to store natural numbers – what if we want something else?

Solution

- In a richer type theory, such as Coq, or Epigram, or Agda, we can give a total run function...
- ... and even provide heterogeneous references.

Sized heaps

- We record the size of the heap:

```
data Heap : Nat -> * where
  empty : Heap 0
  alloc : Nat -> Heap n
              -> Heap (n + 1)
```

Key points

- We make sure references always point to a valid place in the heap;
- We now write `IO n m a` for a computation that takes a heap of size *n* to a heap of size *m*, returning a value of type *a*.
- Our run function uses this "heap size" information to guarantee totality.

But now...

- Precise types help guarantee total semantics,
- but introduce new problems:
  - when we allocate new memory, the type of valid references changes.
  - it becomes much harder to write compositional programs.

Further work

- Add more powerful logical technology (separation logic, Hoare logic, ...)
- Find good examples!
- Combine effects.
- Make it usable.