Haskell and the Curry-Howard isomorphism
Part 1

Ben Sherman

January 27, 2014

Let's play a game

- I'll give you a Haskell type (e.g., $a -> b -> a$)
- Can you construct a (valid) value of that type?
- No cheating!
  - No exceptions or non-termination
  - (No $\mathrm{undefined}$, error, $\mathrm{unsafeCoerce}$, $\mathrm{unsafePerformIO}$, etc.)

a -> a

a -> a

```
id :: a -> a
id x = x
```

```
a -> b -> (a,b)
```

a -> b -> (a,b)

```
(,) :: a -> b -> (a,b)
(,) x y = (x, y)
```

$$(a,b) -> a$$

(a,b) -> a

```
fst :: (a,b) -> a
fst (x, y) = x
```

a −> (a,b)

a -> (a,b)

Nothing!

(a -> b -> c) -> (b -> a -> c)

$$(a \to b \to c) \to (b \to a \to c)$$

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Recall

```
data Maybe a = Just a | Nothing

data Either a b = Left a | Right b
```

a

a

No way!

Maybe a

Maybe a

```
nothing :: Maybe a
nothing = Nothing
```

a -> Either a b

a -> Either a b

```
left :: a -> Either a b
left x = Left x
```

```
Either a b -> a
```

Either a b -> a

Nope!

```
(a -> c) -> (b -> c) -> Either a b -> c
```

```haskell
(a -> c) -> (b -> c) -> Either a b -> c
```

```haskell
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

Either (a -> c) (b -> c) -> a -> b -> c

Either (a -> c) (b -> c) -> a -> b -> c

```haskell
eelim :: Either (a -> c) (b -> c) -> a -> b -> c
eelim (Left f) x y = f x
eelim (Right g) x y = g y
```

$$(b -> c) -> (a -> b) -> (a -> c)$$

(b -> c) -> (a -> b) -> (a -> c)

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) g f x = g (f x)
```

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"
function types : $\mathrm{a} -> \mathrm{b}$	implications (implies) : $p \to q$

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"
function types : a -> b	implications (implies) : $p \to q$
tuples : (a, b)	conjunctions (and) : $p \land q$

We've been doing logic!

Haskell	Logic
type variables : `a`	proposition variables : p
types : `Bool`	propositions : "Socrates is a man"
function types : `a -> b`	implications (implies) : $p \to q$
tuples : `(a, b)`	conjunctions (and) : $p \wedge q$
either : `Either a b`	disjunctions (or) : $p \vee q$

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"
function types : $\mathrm{a} \ {-}{>} \ \mathrm{b}$	implications (implies) : $p \to q$
tuples : $(\mathrm{a},\ \mathrm{b})$	conjunctions (and) : $p \wedge q$
either : $\mathrm{Either}\ \mathrm{a}\ \mathrm{b}$	disjunctions (or) : $p \vee q$
type inhabitation : $\mathrm{id}\ ::\ \mathrm{a} \ {-}{>} \ \mathrm{a}$	truth : $\vdash p \to p$

We've been doing logic!

Haskell	Logic
type variables : a	proposition variables : p
types : Bool	propositions : "Socrates is a man"
function types : $\mathrm{a} \; -> \; \mathrm{b}$	implications (implies) : $p \to q$
tuples : $(\mathrm{a}, \; \mathrm{b})$	conjunctions (and) : $p \wedge q$
either : $\mathrm{Either} \; \mathrm{a} \; \mathrm{b}$	disjunctions (or) : $p \vee q$
type inhabitation : $\mathrm{id} \; :: \; \mathrm{a} \; -> \; \mathrm{a}$	truth : $\vdash p \to p$

The type is the *what*. The value is the *why*.

Programs as proofs

- A type is inhabited if and only if the proposition that it represents is true.
- Any value of a certain type is a proof that the corresponding proposition is true!
- There is a *dynamics of proof*: We can *run* a proof by computing its corresponding value. We can inspect and play with them.
  - Not possible in classical logic systems

Modus ponens is β reduction

In classical logic, *modus ponens* (or implication elimination) is "handed down from up high":

$$\frac{p, p \to q}{q} \to_{\mathsf{elim}}$$

Modus ponens is β reduction

In classical logic, *modus ponens* (or implication elimination) is "handed down from up high":

$$\frac{p, p \to q}{q} \to_{\text{elim}}$$

In Haskell, it's just a consequence of how function application works:

$$\frac{M :: a, (\lambda x.P) :: a \to b}{P[M/x] :: b} \beta_{\text{red}}$$

Other laws are just Haskell features

- The Hilbert system of logic has additional axioms, while natural deduction has additional rules of deduction.
- Haskell constructors give us introduction rules
- Pattern matching gives us elimination rules
- Lambda abstraction gives us additional Hilbert axioms (like const)

Other laws are just Haskell features

- The Hilbert system of logic has additional axioms, while natural deduction has additional rules of deduction.
- Haskell constructors give us introduction rules
- Pattern matching gives us elimination rules
- Lambda abstraction gives us additional Hilbert axioms (like const)

The computational interpretation explains why we have these rules and gives them meaning.

What about negation?

In classical logic, we can always prove the law of the excluded middle:

$$\vdash p \vee \neg p$$

Suppose we had a negation type function in Haskell:
$\mathrm{Not} :: * -> *$.
Do we expect to be able to find an inhabitant of
$\mathrm{Either}\ a\ (\mathrm{Not}\ a)$?

Law of the excluded middle

Suppose we do always have an inhabitant of $\mathrm{Either}\ a\ (\mathrm{Not}\ a)$:

```
type PequalsNP = ...

explainMe :: Either PequalsNP (Not PequalsNP) -> String
explainMe (Left yes) = "Of course! Here's why: " ++ show yes
explainMe (Right no) = "Of course not, because " ++ show no
```

(Being able to inspect proofs works against us here...)

Negation in constructive logic

Classical negation is too powerful in constructive logic. Let's use a more sensible definition of negation:

```
data Absurdity --no constructors, empty type

type Not a = a -> Absurdity
```

Classical vs. constructive negation

Classical:

$$a \longleftrightarrow \neg(\neg a)$$

Constructive:

```haskell
--forwards :: a -> Not (Not a)
--forwards :: a -> Not a -> Absurdity
forwards :: a -> (a -> Absurdity) -> Absurdity
forwards x f = f x

--backwards :: Not (Not a) -> a
--backwards :: ((a -> Absurdity) -> Absurdity) -> a
```

Unfortunately, we can't make an a with that!

Contrapositives

```haskell
contra :: (a -> b) -> (Not b -> Not a)
--contra :: (a -> b) -> (b -> Absurdity) -> (a -> Absurdity)
contra f g = g . f
```

(Not is a contravariant functor, and contra is its contramap)

Constructive negation

Just because something is not not true, doesn't mean that it is true!

Constructive negation from 30,000 ft.

You: "I've proved that any non-constant polynomial has a root!"
Me: "Great. I'd love to know a root for my polynomial P."
You: "Let's run my proof... Ah indeed, it would be absurd if P had no roots!"
Me: "I think you only proved that it's not not true that any non-constant polynomial has a root."

Warning: Haskell is not sound!

"Bottom" ($\perp$) inhabits all types:

represents absurdity, or an exception.

- exceptions and unsafe functions
- partial functions
- general recursion

Exceptions and unsafe functions

```haskell
undefined :: a
error  :: String -> a
unsafeCoerce :: a -> b
```

Partial functions

```haskell
head :: [a] -> a
head (x : xs) = x

niceTry :: a
niceTry = head []
```

General recursion

When we define some x :: a, can we assume x :: a when we prove x :: a?

```
--unfortunately, this typechecks
x :: a
x = x
```

Just the beginning!

- We'd like a more expressive logic.
- In particular, it would be nice to make types that depend on values:

```
fta :: (p :: Polynomial)
       -> ( x :: Complex Number , Equal (evaluate p x) 0 )
```

- *Dependent types*
- Try out Agda and Idris!
- "Agda safety: we last proved false on April 18th 2012 ."