

Programs are Proofs

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University
Semester 2, 2009

Programs are Proofs

- Every time we write a function we also prove a theorem.*
- The type of the function we wrote is a theorem that we have proved.
- The proof says that we can construct the function's output values from its input values.
- The following are examples use just **function abstraction/application** \equiv **implication introduction/elimination**.

$$id :: p \rightarrow p$$

$$id = \lambda x. x$$

$$fst :: p \rightarrow q \rightarrow p$$

$$fst = \lambda x. \lambda y. x$$

$$mp :: (p \rightarrow q) \rightarrow p \rightarrow q$$

$$mp = \lambda f. \lambda x. f x$$

The Curry-Howard Isomorphism

Typing rule

Prop Calc (Gentzen)

Prop Calc (Sequent)

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau}$$

$$\frac{[p]}{p}$$

$$\Gamma, p \vdash p$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda x. e_2 :: \tau_1 \rightarrow \tau_2}$$

$$\frac{[p]}{p \rightarrow q}$$

$$\frac{\Gamma, p \vdash q}{\Gamma \vdash p \rightarrow q}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}}$$

$$\frac{p \rightarrow q \quad p}{q}$$

$$\frac{\Gamma \vdash p \rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q}$$

* Big Star

- A function used to prove a theorem is called a **proof witness**. Its existence guarantees the fact its outputs can be constructed from its inputs.
- This only works when the language is strongly typed, and we can't use recursion, or exceptions.
- The following examples do **not** prove the property stated by their types.

$$lunch :: p$$

$$lunch = lunch$$

$$sorry :: (p \rightarrow q) \rightarrow p \rightarrow q$$

$$sorry = \lambda x. \lambda y. x \text{ (error "I'm sorry, Dave. I'm afraid I can't do that")}$$

Encoding Conjunction (and)

A conjunction $p \wedge q$ says "both p and q are true" or "we have something of type p and something of type q ".

We represent this in Haskell as the *Pair* data type.

```
data Pair a b = Pair a b
```

Here is a theorem written as a predicate and as a type.

$$(p \wedge (p \rightarrow q)) \rightarrow q \equiv (\text{Pair } p (p \rightarrow q)) \rightarrow q$$

Proved in Haskell:

```
prop :: Pair p (p -> q) -> q
prop = \x -> case x of { Pair a f -> f a }
```

Proved in Haskell:

```
prop :: Either (p -> q) r -> p -> Either r q
prop = \x -> \y ->
    case x of {
        Left f -> Right (f y);
        Right z -> Left z;
    }
```

Proved again using some sugar:

```
prop :: Either (p -> q) r -> p -> Either r q
prop (Left f) y = Right (f y)
prop (Right z) _ = Left z
```

Encoding Disjunction (or)

A disjunction $p \vee q$ says "either p is true or q is true" or "either we have something of type p , or we have something of type q ".

We represent this in Haskell as the *Either* data type.

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

Here is a theorem written as a predicate and as a type.

$$\begin{aligned} ((p \rightarrow q) \vee r) \rightarrow p \rightarrow (r \vee q) \\ \equiv \\ (\text{Either } (p \rightarrow q) r) \rightarrow p \rightarrow (\text{Either } r q) \end{aligned}$$

Comparison with Fitch style proof

$\begin{array}{l} 1 \quad \frac{}{(p \rightarrow q) \vee r} \\ 2 \quad \frac{}{p} \\ 3 \quad \frac{}{p \rightarrow q} \\ 4 \quad \frac{}{q} \\ 5 \quad \frac{}{r \vee q} \\ 6 \quad \frac{}{r} \\ 7 \quad \frac{}{r \vee q} \\ 8 \quad \frac{}{r \vee q} \\ 9 \quad \frac{}{p \rightarrow (r \vee q)} \\ 10 \quad \frac{}{((p \rightarrow q) \vee r) \rightarrow p \rightarrow (r \vee q)} \end{array}$	<pre>prop :: Either (p -> q) r -> p -> Either r q prop = \x -> \y -> case x of { Left f -> Right (f y); Right z -> Left z; }</pre> <p>→-E, 2, 3</p> <p>∨-I, 4</p> <p>∨-I, 6</p> <p>∨-E, 1, 3–5, 6–7</p> <p>→-I, 2–8</p> <p>→-I, 1–9</p>
---	--

Handling Negation

Recall the negation-elimination rule from Natural Deduction.

If we assume that p is false, but this leads to a contradiction, then p must actually be true.

$$\frac{[\neg p] \quad q \wedge \neg q}{p}$$

Negation Elimination

If we have both p and $\neg p$ are true, then q is true, for any q .

We'll wrap up this idea in the data type for *Not*.

```
data Not p = Not (forall q. p -> q)
```

If we have something of type p , and something of type $\text{Not } p$, then we can extract the function from the $\text{Not } p$ and make something of type q .

This lets us write a function to perform something like negation elimination.

```
notElim :: p -> Not p -> q
notElim = \xp -> \xnp -> case xnp of { Not f -> f xp }
```

Example

$$\frac{\frac{[\text{not outside}]^1 \quad \frac{\text{not outside} \rightarrow \text{computer games}}{\text{computer games}} \text{ (Axiom)} \quad (\rightarrow E)}{\text{computer games} \wedge \text{no computer games}} \quad \frac{\text{no computer games}}{(\wedge I)} \text{ (Axiom)}}{\text{outside}} \text{ } (\neg E[1])$$



Contradiction Elimination

In Natural Deduction we write truth as \top or **T**, and falsehood as \perp or **F**.

We'll define these in Haskell as:

```
data TRUTH = TRUTH
type FALSE = Not TRUTH
```

Now we can write functions to perform contradiction intro and elimination.

```
absurd :: FALSE -> p
absurd = \xf -> case xf of { Not tp -> tp TRUTH }

contra :: Pair p (Not p) -> FALSE
contra = \xp -> case xp of { Pair xp xnp -> notElim xp xnp }
```

Negation Introduction

```
notIntro :: (p -> FALSE) -> Not p
notIntro = \xpf -> Not (\xp -> absurd (xpf xp))
```

Example: not-or elimination

$$\frac{\neg(p \vee q)}{\neg p}$$

```
notOrElim :: Not (Either p q) -> Not p
notOrElim = \x -> notIntro (\xp -> notElim (Left xp) x)
```

$$\frac{\neg(p \vee q)}{\neg p}$$

```
notOrElim :: Not (Either p q) -> Not p
notOrElim
= \x1 :: Not (Either p q) ->
  let z2 = \x2 :: p ->
        let x3 :: Either p q           = Left x2
            x4 :: Pair (Either p q) (Not (Either p q)) = Pair x3 x1
            x5 :: FALSE                 = contra x4
        in absurd x5
  in notIntro z2
```

Writing it a different way...

$$\frac{\neg(p \vee q)}{\neg p}$$

1	$\neg(p \vee q)$	
2	p	
3	$p \vee q$	\vee -I, 2
4	$(p \vee q) \wedge \neg(p \vee q)$	\wedge -I, 1, 3
5	\perp	
6	$\neg p$	\neg -I, 2–5
7	$\neg(p \vee q) \rightarrow \neg p$	\rightarrow -I, 1–6

Proof assistants

- A proof assistant is a program that can help you [write a witness with a given type signature](#) \equiv [write a proof for a given theorem](#).
- They can be thought of as inverse type inferencers. You supply the type, they supply the program.
- They aren't entirely automatic. Some will type-check your program/proof, but not help you find it.
- **Examples:**
 - Edinburgh LCF, HOL, Isabelle
 - Coq
 - Agda
 - Twelf