Curry-Howard correspondence (cont’d)

We drew the following connection between several branches of mathematics:

Algebra	Logic	Types	Sets
0	$ `\bot` $ (falsity)	`Void`	$ `\emptyset` $
1	$ `\top` $ (truth)	`empty : Unit`	$ `\{ \bullet \}` $
$ `a \times b` $	$ `A \land B` $ (conjunction)	`pair a b : Prod A B`	$ `A \times B = \{ (a, b) \mid a \in A, b \in B \}` $
$ `a + b` $	$ `A \vee B` $ (disjunction)	`left a, right b : Sum A B`	$ `A \uplus B = \{ (a, 0), (b, 1) \mid a \in A, b \in B \}` $

What was missing from this picture were functions and function types, which required the introduction of the lambda calculus and its type system. Now we can paint the full picture.

Algebra	Logic	Types	Sets
$ `b^a` $	$ `A\rightarrow B` $	`A -> B`	$ `A\Rightarrow B = \{ f : A\to B \}` $

The correspondence between Logic-Types-Sets is perhaps not suprising, but why is it *exponentiation* when it comes to algebra? A hint is this: the number of functions from $`A`$ to $`B`$ is $`b^a`$, where these are the sizes of the two respective sets.

Isomorphisms

The correspondence between algebraic exponent laws, type isomorphisms, and tautologies still hold:

- $`a^{b^c} = a^{b\times c}`$, which corresponds to `C -> B -> A ~ (Prod C B) -> A`
- $`a^b\times a^c = a^{b+c}`$, which corresponds to `Prod (B -> A) (C -> A) ~ Sum B C -> A`
- $`(a\times b)^c = a^c \times b^c`$, which corresponds to `C -> Prod A B ~ Prod (C -> A) (C -> B)`
- $`a ^ 1 = a`$, which corresponds to `Unit -> A ~ A`
- $`a ^ 0 = 1`$, which corresponds to `Void -> A ~ Unit`
    - noting that $`0 ^ 0 = 1`$, which corresponds to `Void -> Void ~ Unit`.

The first isomorphism is particularly famous and it has a name: *Currying*. This also comes from the name of Haskell Curry of Curry-Howard and Haskell fame. It is very useful because it says that you can pass the arguments either as a pair or one at a time.

In order to check that the above truly are isos we need to set up functions between types so that they compose into identities in both directions. But since the types are function types, we will need to *compare functions*. We will compare two functions `f, g : A -> B` in the following way: `f = g` if and only if for all `a : A` we have that `f a = g a`. This principle is called *extensionality*.

Example: (Currying)

We need to define the two function, let us call them `cur`(ry) and `unc`(urry).

```
c : {A B C : Set} -> ((Prod C B) -> A) -> (C -> B -> A)
cur                  f                  = \c b -> f (pair
c b)

u : {A B C : Set} -> (C -> B -> A) -> ((Prod C B) -> A)
unc                  f              = \ { (pair c b) -> f
c b }
```

The definitions above emphasise the function-to-function mapping by using lambda terms. In the case of `u` we use a new Agda feature, pattern-matching lambdas. A more straightforward definition can be written by defining the functions explicitly:

```
c : {A B C : Set} -> ((Prod C B) -> A) -> C -> B -> A
cur                  f                    c    b =  f (pai
r c b)

u : {A B C : Set} -> (C -> B -> A) -> (Prod C B) -> A
```

```
unc                   f                 (pair c b) = f c b
```

Let us now check that `c(u(f)) = f : C -> B -> A` and `u(c(g)) = g : Prod C B -> A`.

To check that

```
cur (unc f) = f : C -> B -> A
```

we need to check that for all `c : C, b : B` we have that

```
cur (unc f) c b = f c b
```

We expand the LHS:

```
cur (unc f) c b
  = (unc f) (pair c b)
  = unc f (pair c b)
  = f c b
```

In the other direction:

```
unc (cur f) (pair c b)
  = (cur f) c b
  = cur f c b
  = f (pair c b)
```

Propositions as types

Remember the STLC rules:

- Axiom

```
--------------[Axi]
Γ, x:A ⊢ x : A
```

- Function introduction

```
Γ ⊢ t : A -> B   Γ ⊢ u : A
---------------------------
Γ ⊢ t u : B
```

- Function elimination

```
Γ, x:A ⊢ t : B
--------------
Γ ⊢ t : A -> B
```

Finally, we extend the STLC with Product and Sum:

- Product Elimination Left:

```
Γ ⊢ t : Prod A B
----------------
Γ ⊢ proj1 t : A
```

- Product Elimination Right:

```
Γ ⊢ t : Prod A B
-----------------
Γ ⊢ proj2 t : B
```

- Product Introduction:

```
Γ ⊢ t : A   Γ ⊢ u : B
-----------------------
Γ ⊢ andI t u : Prod A B
```

- Sum Elimination:

```
Γ ⊢ t : Sum A B   Γ ⊢ u : A -> C   Γ ⊢ v : B -> C
-----------------------------------------------------
Γ ⊢  orE t u v : Sum A B -> (A -> C) -> (B -> C) -> C

```

- Sum Introduction Left:

```
Γ ⊢ t : A
---------------------
Γ ⊢ orIl t : Sum A B
```

- Sum Introduction Right:

```
Γ ⊢ t : B
```

```
-------------------
Γ ⊢ orIr t : Sum A B
```

Although not very important for programming, in a spirit of completism we can add:

- *Ex Falso Quodlibet*

```

-------------------
Γ ⊢ EFQ : Void -> A
```

Comparing these rules to those of natural deduction we can see they are (essentially) the same.

Substitution and execution

The final piece of the CH correspondence we will (briefly) look at is how program execution (term evaluation) itself has a logical counterpart in the *Substitution* rule.

Remember that function application is defined by *substitution*, which can be written in STLC as:

```
Γ ⊢ \x -> t : A -> B    Γ ⊢ u : A
---------------------------------
Γ ⊢ t[x / u] : B
```

where `t[x / u]` is `t` with variable `x` replaced by term `u`.

In natural deduction, *Substitution* (aka *Cut* in other formulations of proof rules known as *sequent calculus*) is a *derived* rule which shows how proofs can be joined [1]:

$$
\frac
{\Gamma \vdash A \quad \Gamma\vdash A\to B}
{\Gamma \vdash B}
$$

This rule plays an important technical role in showing propositional logic is *consistent*, i.e. $`\not\vdash\bot`$.

---

1. The form of this rule is usually slightly different, but this is also admissible. ↩