Isomorphisms

A note on syntax

The definition of a polymorphic type (or a proposition) is usually preceded by some variable declarations enclosed in `{...}`. For example, the projection is

```
proj1 : {A B : Set} -> Prod A B -> A
proj1 (pair a b) = a
```

The declaration states that `A` and `B` are types (which are denoted by `Set` in Agda). The curly brackets mean that these variables are "optional" arguments. We could have written just as well

```
proj1 : {A B : Set} -> Prod A B -> A
proj1 {A} {B} (pair a b) = a
```

but since we don't use `A` and `B` in the definition of the function we can ommit them.

Consider by comparison *modus ponens*:

```
mp : {A B : Set} -> (A -> B) -> A -> B
mp {A} {B} f a = b where
  b : B
```

```
  b = f a
```

Because we actually mention `B` in the defintion we need to provide it explicitly. On the other hand if we *inline* the definition

```
mp : {A B : Set} -> (A -> B) -> A -> B
mp {A} {B} f a = f b
```

then we no longer need to mention these optional arguments, and can write just

```
mp : {A B : Set} -> (A -> B) -> A -> B
mp f a = f b
```

Type isomorphism

In the previous lecture we saw how we can “convert” between two types, for example

```
h : {A : Set} -> Sum A A -> Prod Boolean A
h (left a)  = pair true a
h (right a) = pair true a

i : {A : Set} -> Prod Boolean A -> Sum A A
i (pair true a)  = left a
i (pair false a) = right a
```

The functions `h`, `i` happen to be inverses of each other. How do we know? We can calculate!

First let us define the function `hi` which is the composition of `h` and `i`, i.e.

```
hi : {A : Set} -> Prod Boolean A -> Prod Boolean A
hi x = h (i x)
```

We know that the argument type of `hi` is that of `i` and the result type is that of `h`. Since `x : Prod Boolean A` we can pattern-match `x`, which gives the following equivalent definition:

```
hi : {A : Set} -> Prod Boolean A -> Prod Boolean A
hi (pair true a)  = h (i (pair true a))
hi (pair false a) = h (i (pair false a))
```

Note that we pattern-matched bot `Prod` and `Boolean`.

Now let's calculate:

```
hi (pair true a)  = h (i (pair true a))
                  = h (left a)           ... by def of i
                  = pair true a          ... by def of h

hi (pair false a) = h (i (pair false a))
```

```
                  = h (right a)          ... by def of i
                  = pair false a         ... by def of h
```

This covers all the cases. In both cases we note that

```
hi x = x
```

so

```
h (i x) = x
```

Similarly we can compute that

> ***Exercise**:*
>
> - Show that `ih x` is the identity.
> - Show that all the conversion functions from the previous lectures are inverses of each other.

The existence of invertible functions `f : A -> B`, and `g : B -> A` means that the two types are ***isomorphic*** (“same shape”), written `A ~ B`. Isomorphism is a very important mathematical and programming concept, indicating that data can be converted between the two without loss of precision. Practical examples of isomorphic data types are various date and time formats, or different measurement units (Imperial

vs. metric).

If we write `Void` as `0`, `Unit` as `1`, `Prod A B` as `A x B`, `Sum A B` as `A + B` we can see that many type isomorphisms are consistent with calling these data typs *algebraic*:

```
    A + (B + C) \sim (A + B) + C \\
    A + B \sim B + A\\
    A + B \sim A\\
    A \times (B \times C) \sim (A\times B) \times C\\
    A \times B \sim B \times A\\
    A \times 1 \sim A\\
    A \times (B + C) \sim A \times B + A \times C\\
    (A + B) \times C \sim A \times C + B \times C\\
    A \times 0 \sim 0
```

This mathematical structure even has a name: a *semi-ring*.

> ***Exercise:*** *Show that the types above are indeed isomorphic.*

The Curry-Howard Correspondence

We can now reveal the correspondence between Logic and Types. This correspondence is in a mathematical sense an isomorphism, but showing that is beyond the scope of this module. This is why we will call it a “correspondence”, and deal with it informally. We will see how

propositions correspond to *types* and *proofs* correspond to *programs*.

Remember the Postulates from a few lectures ago:

```
postulate
  andEl  : {P Q : Prop}    -> P and Q -> P
  andEr  : {P Q : Prop}    -> P and Q -> Q
  andI   : {P Q : Prop}    -> P -> Q -> P and Q
  orE    : {P Q R : Prop} -> P or Q -> (P -> R) -> (Q -> R
) -> R
  orIl   : {P Q : Prop}    -> P -> P or Q
  orIr   : {P Q : Prop}    -> Q -> Q or P
  EFQ    : {P : Prop}      -> Falsity -> P
  DNE    : {P : Prop}      -> (not (not P)) -> P
```

The propositions-are-types correspondence is:

- `Falsity ~ Void`
- `Truth ~ Unit`
- `A and B ~ Prod A B`
- `A or B ~ Sum A B`

Note: We left out implication (`->`) for now. We will return to it a bit later.

The important point here is that as we identifiy propositions with types, the postulates should be implementable as functions. Lets see if we can implement functions with those types.

```
andEl' : {P Q : Set} -> Prod P Q -> P
andEl' (pair p _) = p

andEr' : {P Q : Set} -> Prod P Q -> Q
andEr' (pair _ q) = q

andI' : {P Q : Set} -> P -> Q -> Prod P Q
andI' p q = pair p q

orE' : {P Q R : Set} -> Sum P Q -> (P -> R) -> (Q -> R) ->
 R
orE' (left p)  f g = f p
orE' (right q) f g = g q

orIl' : {P Q : Set} -> P -> Sum P Q
orIl' p = left p

orIr' : {P Q : Set} -> Q -> Sum P Q
orIr' q = right q

EFQ' : {P : Set} -> Void -> P
EFQ' ()
```

We left one postulate out:

```
DNE' : {P : Prop} -> ((P -> Void) -> Void) -> P
```

```
DNE' x = ?
```

It turns out, this postulate cannot be implemented. It falls outside of the CH correspondence.

How do we know it cannot be implemented? This is an informal argument. If we could implement `DNE` then we could derive an implemntation of the `LEM` directly from

```
LEM : {P : Set} -> Sum P (P -> Void)
LEM = DNE' goal where
  goal z = z (orIr v) where
     v x = z (orIl x)
```

With some clever encodings, `P` could be the type of “termination checkers for Turing Machines” and `P -> Void`, correspondingly the type of “non-termination checkers for Turing Machines”. The function `LEM` would compute the solution to the Halting Problem – which is of course not possible. This argument can be formalised into an actual proof.

The CH Correspondence is important enough to motivate some mathematicians, logicians, and philosophers, to use a logic without DNE (and every equivalent principle, such as the LEM). This logic is called ***Constructive Logic*** (CL) and it truly is *the logic of computation*. It is the fundamental logic of Computer Science. A lot of mathematics has been shown to be possible based on constructive logic.

Proof relevance

Constructive logic takes the concept of *proof* as essential. In contrast, classical logic (constructive logic + DNE) takes the concept of *truth* as essential. To guid your intuition it is best not to think of a constructive-logic proposition `A` as *true* but, if there is a `p` such that `p : A` you should think of it as *proved by* `p`.

In constructive logic, this helps us distinguish between propositions (as types) more carefully.

We know that $`\top\vee\top \leftrightarrow \top`$, i.e. “true or true is equivalent to true”. This is the case clasically and constructively. We can prove that by constructing the function-proof-term of the corresponding types:

```
suu-u : Sum Unit Unit -> Unit
suu-u (left empty) = empty
suu-u (right empty) = empty

u-suu : Unit -> Sum Unit Unit
u-suu empty = left empty
```

However, these functions do not set up an isomorphism; `u-suu` is obviously not a bijection, since `Sum Unit Unit` has 2 elements and `Unit` only 1. Indeed, it is not the case that (using the algebraic notation) `1 + 1 ~ 1`.

This means that *proofs matter*. Proofs *are relevant*. `Unit` and `Sum Unit Unit` are both provable, but the former has essentially one proof(-term), and the latter two proof(-term)s.

Summary

In conclusion: there is an important correspondence between algebra, logic, types, and sets:

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 \}` $

so that

- the equations of Algebra are reflected in isomorphisms of types
- the introduction and elimination rules of Logic are reflected in functions defined using pattern matching and type constructors