# Types

Agda, which we have introduced as a *proof checker* is actually a fully-blown programming language. Here are some simple functions on natural numbers written in Agda:

```
f : Nat -> Nat
f x = 3 * x + 7

g : Nat -> Nat
g x = 4 * x * x + 9 * x + 10

m : Nat
m = max (f 9) (g 6)
```

Agda is not a practical language, but a conceptual language. Its consistent design and disregard for performance makes it a great teaching language *concepts* though (don't try to write you next Andoid app in it though).

We have read `p : A` as *`p` is a proof of `A`*, and we have read `f : A -> B` as either *`f` is a proof of `A -> B`* or *`f` is a rule mapping proofs of `A` into proofs of `B`*.

Alternatively, with a programming hat on, we can read `p : A` as *program `p` has type `A`*, and `f : A -> B` as *`f` is a function from `A`s to*

`B` *s*.

Indeed, this is exactly what is happening in the simple code above.

The concept of ***type*** is central to Agda. In some sense types appear in all programming languages, but Agda treats them in a logically uniform way. First of all, rather strikingly, it has *no built-in types* [1]. However, what Agda offers us is an unparalleled power in constructing data types.

# Basic types or enums

In this section we will show some important types. These are not built into Agda, but they need to be defined. In the process we will also use them as examples for Agda syntax for defining types.

A type definition consists of the following keywords:

```
data Void : Set where
```

- `data` : introduces the definition (keyword)
- `Void` : the name of the type (identifier, can be almost anything, including unicode)
- `Set` : it confirms that `Void` is a type (Agda, confusingly, calls types “sets”; predefined identifier)
- `where` : introduces the *constructors* (keyword).

In the case of *Void* there are no constructors so this data type cannot be instantiated.

Another simple type is:

```
data Unit : Set where
    empty : Unit
```

In this case the definition says that `empty` is the only possible element of `Unit`.

In a language like Java this would be akin to an `Enum` type:

```
public enum Unit { EMPTY }
```

> *Question: Investigate whether the `Void` type is legal in Java: `public enum Void { }`*

In Python this is akin to:

```
class Unit(Enum):
   EMPTY = 1
```

Other types with finite elements can be similarly defined, for example:

```
data Boolean : Set where
  true false : Boolean
```

This can be done with `enum`s in Java and Python although, for reasons

of performance, you may want to stick to the built-in `boolean`s.

We will return to *infinite* types (natural numbers, lists, etc) in a couple of weeks.

# Composite types

Out of simple types we can form *composite* types.

## Product (or tuple)

This is simply a way to put together two existing data types and form a new one. Most programming languages have this ability. For example, in Java, we can write

```
public class Tuple<X, Y> {
  public final X x;
  public final Y y;
  public Tuple(X x, Y y) {
    this.x = x;
    this.y = y;
  }
}
```

In Python, which is essentially untyped, tuples can be directly created from data, e.g.

```
tup1 = ("hello", 0)
```

In Agda, the syntax for creating a tuple is:

```
data Prod (X Y : Set) : Set where
  pair : X -> Y -> Prod X Y
```

Let us look at all the syntactic elements:

- `data` : we are defining a data type
- `Prod` : the name of the data type
- `(A B : Set)` : the *parameters* of the data types are other types ("type polymorphism")
- `: Set` : we are defining a type indeed
- `where` : definitions to follow
- `pair` : the name of the constructor
- `X -> Y -> Prod X Y` : the type of the constructor. It requires an `X` and `Y` and will return their product `Prod X Y`.

Examples:

```
x : Prod Boolean Unit
x = pair true empty

y : Prod (Prod Unit Unit) (Prod Boolean Boolean)
y = pair (pair empty empty) (pair true false)
```

# Sum or union

A sum (union) is a special data type that allows storing different data types in the same memory location. A sum may have many members, but only one member can contain a value at any given time.

This is consider a more sophisticated type construction. The language C had (minimalistic) direct support for it. For example, an *instance* of union type is

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

> ***Aside:*** *The C union types are so primitive they can hardly be considered as such. For example the programmer needs to know how to interpret the data, the type does not remember which one it actually stores.*

In Java you would need to write something rather complicated, but it works properly.

In Agda the definition of the sum simply means that the type has two constructors:

```
data Sum (A B : Set) : Set where
  left  : A -> Sum A B
  right : B -> Sum A B
```

The tags `left` / `right` are attached to the data so that the correct `A` or `B` type can be recovered from the sum.

The types we have defined so far can be freely combined, for example

```
x : Sum Boolean (Prod Unit Unit)
x = left true

y : Sum Boolean (Prod Unit Unit)
y = right (pair empty empty)
```

> ***In-class exercises:***
>
> - define the type of week-days
> - define the type construction of “triples” (tuples with three components)
> - define a type of sums with three components
> - can we define any element of `Prod A Void` for some `A`?
> - can we define any element of `Sum A Void` for some `A`?

# Algebraic types

Because the operations on types are “sum” and “product” this family of

types is usually called "*algebraic*" types. The names “sum” and “product” are not accidental. If type `A` has `m` elements and type `B` has `n` elements, then:

- How many elements has `Sum A B`?
- How many elements has `Prod A B`?

> *Exercise: Experiment with various particular types, then attempt a mathematical proof.*

---

1. This statement is almost true. ↩