# More on types

# Dealing with errors

In imperative programming runtime errors are ubiquitous. Array out of bounds and null pointer dereferencing are just two of the most common such errors. The way programming languages deal with errors differ, from the extreme carelessness of C (many such errors are considered *undefined behaviours*, i.e. *anything at all can happen*, including wiping out the hard drive and charging £5000 to your credit card) to the more controlled runtime exceptions of languages such as Java, Haskell, or OCaml, to the *prohibition of runtime errors* in languages such as Agda or Coq.

But how are we to deal with errors though? How can we implement functions such as “return the head of the list” without knowing for sure at runtime that the list is non-empty, or “return the _n_th element of a list” without knowing that *n* is less than the length?

We can not just write:

```
hd : {A : Set} -> List A -> A
hd (cons x _) = x
```

because we are not dealing with the *nil* case, which is unacceptable to the Agda type checker.

The solution is to use a *special type* called the *option type*:

```
data Option (A : Set) : Set where
    none : Option A
    some : A -> Option A
```

The type is a wrapper around any type $A$, which has two constructors:

- `none` adds a special error value
- `some` injects `A` into the wrapper `Option A`.

With this type we can now write:

```
hd : {A : Set} -> List A -> Option A
hd (cons x _) = some x
hd nil        = none
```

The use of option types is a much better solution than the use of "special values" like `NULL` or -1. Such values are special only because of sometimes implicit conventions which are not type-checked and may be forgotten by the programmer or lost in the documentation.

**Example:**

```
nth : {A : Set} -> Nat -> List A -> Option A
nth _        nil        = none
nth zero    (cons x _)  = some x
```

```
nth (suc n) (cons _ xs) = nth n xs
```

**Observation:**

Once we are introducing the option type the wrapper cannot be removed. There is no function that can extract a proper value from the wrapper type

```
val : {A : Set} -> Option A -> A
val (some a) = a
val none     = ???
```

However, practical language allow the writing of incomplete pattern matches, at the expense of potential runtime exceptions, in order to allow the removal of the wrapper, which must be carried throughout the code.

Consider writing the program that computes *the sum of the second and third elements of a list of naturals*:

```
sum23 : List Nat -> Nat
sum23 xs = add x2 x3 where
  x2 = nth two xs
  x3 = nth three xs
```

is clearly illegal because `add : Nat -> Nat -> Nat` whereas `x2 x3 :`

`Option Nat`. The solution is to *lift* addition to deal with the special `none` value:

```
add' : Option Nat -> Option Nat -> Option Nat
add' (some m) (some n) = some (add m n)
add' _        _        = none
```

So that the function above is:

```
sum23 : List Nat -> Option Nat
sum23 xs = add' x2 x3 where
  x2 = nth two xs
  x3 = nth three xs
```

The lifting can be also performed as a *generic* operation:

```
lift2 : (Nat -> Nat -> Nat) -> (Option Nat -> Option Nat -
> Option Nat)
lift2 f (some n) (some m) = some (f m n)
lift2 f _        _        = none

add' = lift2 add
```

A much more general approach to “lifting” functions will be seen when we study *monads*.

> ***Exercise:*** *Show that* `Option A ~ Sum A Unit`.

# Other types

## Trees

We can visualise a list as an imbalanced tree. For example the list `[0,1,2] = cons 0 (cons 1 (cons 2 nil))` can be visualised as

```
graph TD;
  cons_a --> 0;
  cons_a --> cons_b;
  cons_b --> 1;
  cons_b --> cons_c;
  cons_c --> 2;
  cons_c --> nil;
```

All recursive types can be visualised as kinds of trees. The following is a definition of a *binary* tree with data stored *at the leaves*:

```
data BTreeL (A : Set) : Set where
  leaf : A -> BTreeL A
  node : BTreeL A -> BTreeL A -> BTreeL A
```

The following is an example of data of this type:

```
graph TD;
  node1 --> node2;
  node1 --> node3;
  node2 --> leaf4;
  node2 --> leaf9;
  leaf9 --> 2;
  node3 --> node5;
  node3 --> node6;
  node5 --> leaf7;
  node5 --> leaf10;
  leaf10 --> 1;
  node6 --> leaf8;
  node6 --> leaf11;
  leaf11 --> 0;
  leaf4 --> 7;;
  leaf7 --> 5;;
  leaf8 --> 3;;
```

## S-expressions

A very general data-type, which is used in *serialisation* of arbitrary data structures is the S-expression. It representes nested paranthetical expressions whose atomic values are (usually) strings. They were the key data structure of the LISP programming language and remain one of the most useful ways to encode arbitrary structures in human-readable format.

The definition is:

```
data Sexp (A : Set) : Set where
  atom : A -> Sexp A
  list : List (Sexp A) -> Sexp A
```

Supposing that we have a type of `String`s defined, the expression

```
(this (is an) (s expression))
```

would be represented as

```
open import Agda.Builtin.String

a-this = atom "this"
a-is   = atom "is"
a-an   = atom "an"
a-s    = atom "s"
a-exp  = atom "expression"

l-is-an = list (cons a-is (cons a-an nil))
l-s-exp = list (cons a-s (cons a-exp nil))
l-ex    = list (cons a-this (cons l-is-an (cons l-s-exp ni
l)))
```

which means

```
list
(cons (atom "this")
 (cons (list (cons (atom "is") (cons (atom "an") nil)))
  (cons (list (cons (atom "s") (cons (atom "expression") n
il)))
   nil)))
```