The Idris Tutorial

Version 0.0

Contents

This is a crash course in Idris 2 (sort of a tutorial, but rather less gentle I'm afraid!). It provides a brief introduction to programming in the Idris Language. It covers the core language features, assuming some experience with an existing functional programming language such as Haskell or OCaml.

This has been revised and updated from the Idris 1 tutorial. For details of changes since Idris 1, see updates-index.

---

1 Introduction

In conventional programming languages, there is a clear distinction between *types* and *values*. For example, in Haskell, the following are types, representing integers, characters, lists of characters, and lists of any value respectively:

- `Int`, `Char`, `[Char]`, `[a]`

Correspondingly, the following values are examples of inhabitants of those types:

- `42`, `'a'`, `"Hello world!"`, `[2,3,4,5,6]`

In a language with *dependent types*, however, the distinction is less clear. Dependent types allow types to "depend" on values — in other words, types are a *first class* language construct and can be manipulated

like any other value. The standard example is the type of lists of a given length[1], `Vect n a`, where `a` is the element type and `n` is the length of the list and can be an arbitrary term.

When types can contain values, and where those values describe properties, for example the length of a list, the type of a function can begin to describe its own properties. Take for example the concatenation of two lists. This operation has the property that the resulting list's length is the sum of the lengths of the two input lists. We can therefore give the following type to the `app` function, which concatenates vectors:

```
app : Vect n a -> Vect m a -> Vect (n + m) a
```

This tutorial introduces Idris, a general purpose functional programming language with dependent types. The goal of the Idris project is to build a dependently typed language suitable for verifiable general purpose programming. To this end, Idris is a compiled language which aims to generate efficient executable code. It also has a lightweight foreign function interface which allows easy interaction with external libraries.

1.1 Intended Audience

This tutorial is intended as a brief introduction to the language, and is aimed at readers already familiar with a functional language such as Haskell or OCaml. In particular, a certain amount of familiarity with Haskell syntax is assumed, although most concepts will at least be explained briefly. The reader is also assumed to have some interest in using dependent types for writing and verifying software.

For a more in-depth introduction to Idris, which proceeds at a much slower pace, covering interactive program development, with many more examples, see Type-Driven Development with Idris by Edwin Brady, available from Manning.

1.2 Example Code

This tutorial includes some example code, which has been tested against Idris 2. These files are available with the Idris 2 distribution, so that you can try them out easily. They can be found under `samples`. It is, however, strongly recommended that you type them in yourself, rather than simply loading and reading them.

2 Getting Started

2.1 Installing from Source

Prerequisites

Idris 2 is implemented in Idris 2 itself, so to bootstrap it you can build from generated Scheme sources. To do this, you need either Chez Scheme (default, and currently preferred since it is the fastest) or Racket. You can get one of these from:

- Chez Scheme
- Racket

Both are also available from MacPorts/Homebrew and all major Linux distributions.

[1] Typically, and perhaps confusingly, referred to in the dependently typed programming literature as "vectors".

Note: If you install Chez Scheme from source files, building it locally, make sure you run `./configure --threads` to build multithreading support in.

Downloading and Installing

You can download the Idris 2 source from the Idris web site or get the latest development version from idris-lang/Idris2 on Github. This includes the Idris 2 source code and the Scheme code generated from that. Once you have unpacked the source, you can install it as follows:

```
make bootstrap SCHEME=chez
```

Where *chez* is the executable name of the Chez Scheme compiler. This will vary from system to system, but is often one of `scheme`, `chezscheme`, or `chezscheme9.5`. If you are building via Racket, you can install it as follows:

```
make bootstrap-racket
```

Once you've successfully bootstrapped with any of the above commands, you can install with the command `make install`. This will, by default, install into `${HOME}/.idris2`. You can change this by editing the options in `config.mk`. For example, to install into `/usr/local`, you can edit the `PREFIX` as follows:

```
PREFIX ?= /usr/local
```

2.2 Installing from a Package Manager

Installing Using Homebrew

If you are Homebrew user you can install Idris 2 together with all the requirements by running following command:

```
brew install idris2
```

2.3 Checking Installation

To check that installation has succeeded, and to write your first Idris program, create a file called `hello.idr` containing the following text:

```
module Main

main : IO ()
main = putStrLn "Hello world"
```

If you are familiar with Haskell, it should be fairly clear what the program is doing and how it works, but if not, we will explain the details later. You can compile the program to an executable by entering `idris2 hello.idr -o hello` at the shell prompt. This will, by default, create an executable called `hello`, which invokes a generated and compiled Chez Schem program, in the destination directory `build/exec` which you can run:

```
$ idris2 hello.idr -o hello
$ ./build/exec/hello
Hello world
```

(On Macos you may first need to install realpath: `brew install coreutils`)

Please note that the dollar sign `$` indicates the shell prompt! Some useful options to the Idris command are:

- `-o prog` to compile to an executable called `prog`.
- `--check` type check the file and its dependencies without starting the interactive environment.
- `--package pkg` add package as dependency, e.g. `--package contrib` to make use of the contrib package.
- `--help` display usage summary and command line options.

You can find out more about compiling to executables in Section sect-execs.

2.4 The Interactive Environment

Entering `idris2` at the shell prompt starts up the interactive environment. You should see something like the following:

```
$ idris2
     ____    __     _         ___
    /  _/___/ /____(_)____   |__ \
    / // __  / ___/ / ___/   __/ /     Version 0.3.0
  _/ // /_/ / /  / (__  )   / __/      https://www.idris-lang.org
 /___/\__,_/_/  /_/____/   /____/      Type :? for help

Welcome to Idris 2.  Enjoy yourself!
Main>
```

This gives a `ghci` style interface which allows evaluation of, as well as type checking of, expressions; theorem proving, compilation; editing; and various other operations. The command `:?` gives a list of supported commands. Below, we see an example run in which `hello.idr` is loaded, the type of `main` is checked and then the program is compiled to the executable file `hello`, available in the destination directory `build/exec/`. Type checking a file, if successful, creates a bytecode version of the file (in this case `build/ttc/hello.ttc`) to speed up loading in future. The bytecode is regenerated if the source file changes.

```
$ idris2 hello.idr
     ____    __     _         ___
    /  _/___/ /____(_)____   |__ \
    / // __  / ___/ / ___/   __/ /     Version 0.3.0
  _/ // /_/ / /  / (__  )   / __/      https://www.idris-lang.org
 /___/\__,_/_/  /_/____/   /____/      Type :? for help

Welcome to Idris 2.  Enjoy yourself!
Main> :t main
Main.main : IO ()
Main> :c hello main
File build/exec/hello written
Main> :q
Bye for now!
```

3 Types and Functions

3.1 Primitive Types

Idris defines several primitive types: `Int`, `Integer` and `Double` for numeric operations, `Char` and `String` for text manipulation, and `Ptr` which represents foreign pointers. There are also several data types declared in the library, including `Bool`, with values `True` and `False`. We can declare some constants with these types. Enter the following into a file `Prims.idr` and load it into the Idris interactive environment by typing `idris2 Prims.idr`:

```
module Prims

x : Int
x = 94

foo : String
foo = "Sausage machine"

bar : Char
bar = 'Z'

quux : Bool
quux = False
```

An Idris file consists of an optional module declaration (here `module Prims`) followed by an optional list of imports and a collection of declarations and definitions. In this example no imports have been specified. However Idris programs can consist of several modules and the definitions in each module each have their own namespace. This is discussed further in Section *Modules and Namespaces* (page 32). When writing Idris programs both the order in which definitions are given and indentation are significant. Functions and data types must be defined before use, incidentally each definition must have a type declaration, for example see `x : Int`, `foo : String`, from the above listing. New declarations must begin at the same level of indentation as the preceding declaration. Alternatively, a semicolon `;` can be used to terminate declarations.

A library module `prelude` is automatically imported by every Idris program, including facilities for IO, arithmetic, data structures and various common functions. The prelude defines several arithmetic and comparison operators, which we can use at the prompt. Evaluating things at the prompt gives an answer, for example:

```
Prims> 13+9*9
94 : Integer
Prims> x == 9*9+13
True
```

All of the usual arithmetic and comparison operators are defined for the primitive types. They are overloaded using interfaces, as we will discuss in Section *Interfaces* (page 22) and can be extended to work on user defined types. Boolean expressions can be tested with the `if...then...else` construct, for example:

```
*prims> if x == 8 * 8 + 30 then "Yes!" else "No!"
"Yes!"
```

3.2 Data Types

Data types are declared in a similar way and with similar syntax to Haskell. Natural numbers and lists, for example, can be declared as follows:

```
data Nat    = Z   | S Nat           -- Natural numbers
                                    -- (zero and successor)
data List a = Nil | (::) a (List a) -- Polymorphic lists
```

The above declarations are taken from the standard library. Unary natural numbers can be either zero (`Z`), or the successor of another natural number (`S k`). Lists can either be empty (`Nil`) or a value added to the front of another list (`x :: xs`). In the declaration for `List`, we used an infix operator `::`. New operators such as this can be added using a fixity declaration, as follows:

```
infixr 10 ::
```

Functions, data constructors and type constructors may all be given infix operators as names. They may be used in prefix form if enclosed in brackets, e.g. `(::)`. Infix operators can use any of the symbols:

```
:+-*\/=.?|&><!@$%^~#
```

Some operators built from these symbols can't be user defined. These are

`%`, `\`, `:`, `=`, `|`, `|||`, `<-`, `->`, `=>`, `?`, `!`, `&`, `**`, `..`

3.3 Functions

Functions are implemented by pattern matching, again using a similar syntax to Haskell. The main difference is that Idris requires type declarations for all functions, using a single colon `:` (rather than Haskell's double colon `::`). Some natural number arithmetic functions can be defined as follows, again taken from the standard library:

```
-- Unary addition
plus : Nat -> Nat -> Nat
plus Z     y = y
plus (S k) y = S (plus k y)

-- Unary multiplication
mult : Nat -> Nat -> Nat
mult Z     y = Z
mult (S k) y = plus y (mult k y)
```

The standard arithmetic operators `+` and `*` are also overloaded for use by `Nat`, and are implemented using the above functions. Unlike Haskell, there is no restriction on whether types and function names must begin with a capital letter or not. Function names (`plus` and `mult` above), data constructors (`Z`, `S`, `Nil` and `::`) and type constructors (`Nat` and `List`) are all part of the same namespace. By convention, however, data types and constructor names typically begin with a capital letter. We can test these functions at the Idris prompt:

```
Main> plus (S (S Z)) (S (S Z))
4
Main> mult (S (S (S Z))) (plus (S (S Z)) (S (S Z)))
12
```

Like arithmetic operations, integer literals are also overloaded using interfaces, meaning that we can also test the functions as follows:

```
Idris> plus 2 2
4
Idris> mult 3 (plus 2 2)
12
```

You may wonder, by the way, why we have unary natural numbers when our computers have perfectly good integer arithmetic built in. The reason is primarily that unary numbers have a very convenient structure which is easy to reason about, and easy to relate to other data structures as we will see later. Nevertheless, we do not want this convenience to be at the expense of efficiency. Fortunately, Idris knows about the relationship between `Nat` (and similarly structured types) and numbers. This means it can

optimise the representation, and functions such as `plus` and `mult`.

`where` clauses

Functions can also be defined *locally* using `where` clauses. For example, to define a function which reverses a list, we can use an auxiliary function which accumulates the new, reversed list, and which does not need to be visible globally:

```
reverse : List a -> List a
reverse xs = revAcc [] xs where
  revAcc : List a -> List a -> List a
  revAcc acc [] = acc
  revAcc acc (x :: xs) = revAcc (x :: acc) xs
```

Indentation is significant — functions in the `where` block must be indented further than the outer function.

Note: Scope

Any names which are visible in the outer scope are also visible in the `where` clause (unless they have been redefined, such as `xs` here). A name which appears in the type will be in scope in the `where` clause.

As well as functions, `where` blocks can include local data declarations, such as the following where `MyLT` is not accessible outside the definition of `foo`:

```
foo : Int -> Int
foo x = case isLT of
            Yes => x*2
            No => x*4
    where
       data MyLT = Yes | No

       isLT : MyLT
       isLT = if x < 20 then Yes else No
```

Functions defined in a `where` clause need a type declaration just like any top level function. Here is another example of how this works in practice:

```
even : Nat -> Bool
even Z = True
even (S k) = odd k where
  odd : Nat -> Bool
  odd Z = False
  odd (S k) = even k

test : List Nat
test = [c (S 1), c Z, d (S Z)]
  where c : Nat -> Nat
        c x = 42 + x

        d : Nat -> Nat
        d y = c (y + 1 + z y)
              where z : Nat -> Nat
                    z w = y + w
```

Totality and Covering

By default, functions in Idris must be `covering`. That is, there must be patterns which cover all possible values of the inputs types. For example, the following definition will give an error:

```
fromMaybe : Maybe a -> a
fromMaybe (Just x) = x
```

This gives an error because `fromMaybe Nothing` is not defined. Idris reports:

```
frommaybe.idr:1:1--2:1:fromMaybe is not covering. Missing cases:
        fromMaybe Nothing
```

You can override this with a `partial` annotation:

```
partial fromMaybe : Maybe a -> a
fromMaybe (Just x) = x
```

However, this is not advisable, and in general you should only do this during the initial development of a function, or during debugging. If you try to evaluate `fromMaybe Nothing` at run time you will get a run time error.

Holes

Idris programs can contain *holes* which stand for incomplete parts of programs. For example, we could leave a hole for the greeting in our "Hello world" program:

```
main : IO ()
main = putStrLn ?greeting
```

The syntax `?greeting` introduces a hole, which stands for a part of a program which is not yet written. This is a valid Idris program, and you can check the type of `greeting`:

```
Main> :t greeting
--------------------------------------
greeting : String
```

Checking the type of a hole also shows the types of any variables in scope. For example, given an incomplete definition of `even`:

```
even : Nat -> Bool
even Z = True
even (S k) = ?even_rhs
```

We can check the type of `even_rhs` and see the expected return type, and the type of the variable `k`:

```
Main> :t even_rhs
   k : Nat
--------------------------------------
even_rhs : Bool
```

Holes are useful because they help us write functions *incrementally*. Rather than writing an entire function in one go, we can leave some parts unwritten and use Idris to tell us what is necessary to complete the definition.

3.4 Dependent Types

First Class Types

In Idris, types are first class, meaning that they can be computed and manipulated (and passed to functions) just like any other language construct. For example, we could write a function which computes a type:

```
isSingleton : Bool -> Type
isSingleton True = Nat
isSingleton False = List Nat
```

This function calculates the appropriate type from a `Bool` which flags whether the type should be a singleton or not. We can use this function to calculate a type anywhere that a type can be used. For example, it can be used to calculate a return type:

```
mkSingle : (x : Bool) -> isSingleton x
mkSingle True = 0
mkSingle False = []
```

Or it can be used to have varying input types. The following function calculates either the sum of a list of `Nat`, or returns the given `Nat`, depending on whether the singleton flag is true:

```
sum : (single : Bool) -> isSingleton single -> Nat
sum True x = x
sum False [] = 0
sum False (x :: xs) = x + sum False xs
```

Vectors

A standard example of a dependent data type is the type of "lists with length", conventionally called vectors in the dependent type literature. They are available as part of the Idris library, by importing `Data.Vect`, or we can declare them as follows:

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

Note that we have used the same constructor names as for `List`. Ad-hoc name overloading such as this is accepted by Idris, provided that the names are declared in different namespaces (in practice, normally in different modules). Ambiguous constructor names can normally be resolved from context.

This declares a family of types, and so the form of the declaration is rather different from the simple type declarations above. We explicitly state the type of the type constructor `Vect` — it takes a `Nat` and a type as an argument, where `Type` stands for the type of types. We say that `Vect` is *indexed* over `Nat` and *parameterised* by `Type`. Each constructor targets a different part of the family of types. `Nil` can only be used to construct vectors with zero length, and `::` to construct vectors with non-zero length. In the type of `::`, we state explicitly that an element of type `a` and a tail of type `Vect k a` (i.e., a vector of length `k`) combine to make a vector of length `S k`.

We can define functions on dependent types such as `Vect` in the same way as on simple types such as `List` and `Nat` above, by pattern matching. The type of a function over `Vect` will describe what happens to the lengths of the vectors involved. For example, `++`, defined as follows, appends two `Vect`:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

The type of `(++)` states that the resulting vector's length will be the sum of the input lengths. If we get the definition wrong in such a way that this does not hold, Idris will not accept the definition. For example:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ xs -- BROKEN
```

When run through the Idris type checker, this results in the following:

```
$ idris2 Vect.idr --check
1/1: Building Vect (Vect.idr)
Vect.idr:7:26--8:1:While processing right hand side of Main.++ at Vect.idr:7:1--8:1:
When unifying plus k k and plus k m
Mismatch between:
        k
and
        m
```

This error message suggests that there is a length mismatch between two vectors — we needed a vector of length `k + m`, but provided a vector of length `k + k`.

The Finite Sets

Finite sets, as the name suggests, are sets with a finite number of elements. They are available as part of the Idris library, by importing `Data.Fin`, or can be declared as follows:

```
data Fin : Nat -> Type where
   FZ : Fin (S k)
   FS : Fin k -> Fin (S k)
```

From the signature, we can see that this is a type constructor that takes a `Nat`, and produces a type. So this is not a set in the sense of a collection that is a container of objects, rather it is the canonical set of unnamed elements, as in "the set of 5 elements," for example. Effectively, it is a type that captures integers that fall into the range of zero to `(n - 1)` where `n` is the argument used to instantiate the `Fin` type. For example, `Fin 5` can be thought of as the type of integers between 0 and 4.

Let us look at the constructors in greater detail.

`FZ` is the zeroth element of a finite set with `S k` elements; `FS n` is the `n+1`th element of a finite set with `S k` elements. `Fin` is indexed by a `Nat`, which represents the number of elements in the set. Since we can't construct an element of an empty set, neither constructor targets `Fin Z`.

As mentioned above, a useful application of the `Fin` family is to represent bounded natural numbers. Since the first `n` natural numbers form a finite set of `n` elements, we can treat `Fin n` as the set of integers greater than or equal to zero and less than `n`.

For example, the following function which looks up an element in a `Vect`, by a bounded index given as a `Fin n`, is defined in the prelude:

```
index : Fin n -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

This function looks up a value at a given location in a vector. The location is bounded by the length of the vector (`n` in each case), so there is no need for a run-time bounds check. The type checker guarantees that the location is no larger than the length of the vector, and of course no less than zero.

Note also that there is no case for `Nil` here. This is because it is impossible. Since there is no element of

`Fin Z`, and the location is a `Fin n`, then `n` can not be `Z`. As a result, attempting to look up an element in an empty vector would give a compile time type error, since it would force `n` to be `Z`.

Implicit Arguments

Let us take a closer look at the type of `index`:

```
index : Fin n -> Vect n a -> a
```

It takes two arguments, an element of the finite set of `n` elements, and a vector with `n` elements of type `a`. But there are also two names, `n` and `a`, which are not declared explicitly. These are *implicit* arguments to `index`. We could also write the type of `index` as:

```
index : forall a, n . Fin n -> Vect n a -> a
```

Implicit arguments, given with the `forall` declaration, are not given in applications of `index`; their values can be inferred from the types of the `Fin n` and `Vect n a` arguments. Any name beginning with a lower case letter which appears as a parameter or index in a type declaration, which is not applied to any arguments, will *always* be automatically bound as an implicit argument. Implicit arguments can still be given explicitly in applications, using `{a=value}` and `{n=value}`, for example:

```
index {a=Int} {n=2} FZ (2 :: 3 :: Nil)
```

In fact, any argument, implicit or explicit, may be given a name. We could have declared the type of `index` as:

```
index : (i : Fin n) -> (xs : Vect n a) -> a
```

It is a matter of taste whether you want to do this — sometimes it can help document a function by making the purpose of an argument more clear.

The names of implicit arguments are in scope in the body of the function, although they cannot be used at run time. There is much more to say about implicit arguments - we will discuss the question of what is available at run time, among other things, in Section *Multiplicities* (page 36)

Note: Declaration Order and `mutual` blocks

In general, functions and data types must be defined before use, since dependent types allow functions to appear as part of types, and type checking can rely on how particular functions are defined (though this is only true of total functions; see Section *Totality Checking* (page 54)). However, this restriction can be relaxed by using a `mutual` block, which allows data types and functions to be defined simultaneously:

```
mutual
  even : Nat -> Bool
  even Z = True
  even (S k) = odd k

  odd : Nat -> Bool
  odd Z = False
  odd (S k) = even k
```

In a `mutual` block, first all of the type declarations are added, then the function bodies. As a result, none of the function types can depend on the reduction behaviour of any of the functions in the block.

3.5 I/O

Computer programs are of little use if they do not interact with the user or the system in some way. The difficulty in a pure language such as Idris — that is, a language where expressions do not have side-effects — is that I/O is inherently side-effecting. So, Idris provides a parameterised type `IO` which *describes* the interactions that the run-time system will perform when executing a function:

```
data IO a -- description of an IO operation returning a value of type a
```

We'll leave the definition of `IO` abstract, but effectively it describes what the I/O operations to be executed are, rather than how to execute them. The resulting operations are executed externally, by the run-time system. We've already seen one I/O program:

```
main : IO ()
main = putStrLn "Hello world"
```

The type of `putStrLn` explains that it takes a string, and returns an I/O action which produces an element of the unit type `()`. There is a variant `putStr` which decribes the output of a string without a newline:

```
putStrLn : String -> IO ()
putStr   : String -> IO ()
```

We can also read strings from user input:

```
getLine : IO String
```

A number of other I/O operations are available. For example, by adding `import System.File` to your program, you get access to functions for reading and writing files, including:

```
data File -- abstract
data Mode = Read | Write | ReadWrite

openFile : (f : String) -> (m : Mode) -> IO (Either FileError File)
closeFile : File -> IO ()

fGetLine : (h : File) -> IO (Either FileError String)
fPutStr : (h : File) -> (str : String) -> IO (Either FileError ())
fEOF : File -> IO Bool
```

Note that several of these return `Either`, since they may fail.

3.6 "do" notation

I/O programs will typically need to sequence actions, feeding the output of one computation into the input of the next. `IO` is an abstract type, however, so we can't access the result of a computation directly. Instead, we sequence operations with `do` notation:

```
greet : IO ()
greet = do putStr "What is your name? "
           name <- getLine
           putStrLn ("Hello " ++ name)
```

The syntax `x <- iovalue` executes the I/O operation `iovalue`, of type `IO a`, and puts the result, of type `a` into the variable `x`. In this case, `getLine` returns an `IO String`, so `name` has type `String`. Indentation is significant — each statement in the do block must begin in the same column. The `pure` operation allows us to inject a value directly into an IO operation:

```
pure : a -> IO a
```

As we will see later, `do` notation is more general than this, and can be overloaded.

You can try executing `greet` at the Idris 2 REPL by running the command `:exec greet`:

3.7 Laziness

Normally, arguments to functions are evaluated before the function itself (that is, Idris uses *eager* evaluation). However, this is not always the best approach. Consider the following function:

```
ifThenElse : Bool -> a -> a -> a
ifThenElse True  t e = t
ifThenElse False t e = e
```

This function uses one of the `t` or `e` arguments, but not both. We would prefer if *only* the argument which was used was evaluated. To achieve this, Idris provides a `Lazy` primitive, which allows evaluation to be suspended. It is a primitive, but conceptually we can think of it as follows:

```
data Lazy : Type -> Type where
     Delay : (val : a) -> Lazy a

Force : Lazy a -> a
```

A value of type `Lazy a` is unevaluated until it is forced by `Force`. The Idris type checker knows about the `Lazy` type, and inserts conversions where necessary between `Lazy a` and `a`, and vice versa. We can therefore write `ifThenElse` as follows, without any explicit use of `Force` or `Delay`:

```
ifThenElse : Bool -> Lazy a -> Lazy a -> a
ifThenElse True  t e = t
ifThenElse False t e = e
```

3.8 Infinite data Types

Infinite data types (codata) allow us to define infinite data structures by marking recursive arguments as potentially infinite. One example of an infinite type is Stream, which is defined as follows.

```
data Stream : Type -> Type where
  (::) : (e : a) -> Inf (Stream a) -> Stream a
```

The following is an example of how the codata type `Stream` can be used to form an infinite data structure. In this case we are creating an infinite stream of ones.

```
ones : Stream Nat
ones = 1 :: ones
```

3.9 Useful Data Types

Idris includes a number of useful data types and library functions (see the `libs/` directory in the distribution, and the documentation). This section describes a few of these, and how to import them.

`List` and `Vect`

We have already seen the `List` and `Vect` data types:

```
data List a = Nil | (::) a (List a)

data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

You can get access to `Vect` with `import Data.Vect`. Note that the constructor names are the same for each — constructor names (in fact, names in general) can be overloaded, provided that they are declared in different namespaces (see Section *Modules and Namespaces* (page 32)), and will typically be resolved according to their type. As syntactic sugar, any type with the constructor names `Nil` and `::` can be written in list form. For example:

- `[]` means `Nil`
- `[1,2,3]` means `1 :: 2 :: 3 :: Nil`

The library also defines a number of functions for manipulating these types. `map` is overloaded both for `List` and `Vect` (we'll see more details of precisely how later when we cover interfaces in Section *Interfaces* (page 22)) and applies a function to every element of the list or vector.

```
map : (a -> b) -> List a -> List b
map f []        = []
map f (x :: xs) = f x :: map f xs

map : (a -> b) -> Vect n a -> Vect n b
map f []        = []
map f (x :: xs) = f x :: map f xs
```

For example, given the following vector of integers, and a function to double an integer:

```
intVec : Vect 5 Int
intVec = [1, 2, 3, 4, 5]

double : Int -> Int
double x = x * 2
```

the function `map` can be used as follows to double every element in the vector:

```
*UsefulTypes> show (map double intVec)
"[2, 4, 6, 8, 10]" : String
```

For more details of the functions available on `List` and `Vect`, look in the library files:

- `libs/base/Data/List.idr`
- `libs/base/Data/Vect.idr`

Functions include filtering, appending, reversing, and so on.

Aside: Anonymous functions and operator sections

There are neater ways to write the above expression. One way would be to use an anonymous function:

```
*UsefulTypes> show (map (\x => x * 2) intVec)
"[2, 4, 6, 8, 10]" : String
```

The notation `\x => val` constructs an anonymous function which takes one argument, `x` and returns the expression `val`. Anonymous functions may take several arguments, separated by commas, e.g. `\x, y, z => val`. Arguments may also be given explicit types, e.g. `\x : Int => x * 2`, and can pattern match, e.g. `\(x, y) => x + y`. We could also use an operator section:

```
*UsefulTypes> show (map (* 2) intVec)
"[2, 4, 6, 8, 10]" : String
```

`(*2)` is shorthand for a function which multiplies a number by 2. It expands to `\x => x * 2`. Similarly, `(2*)` would expand to `\x => 2 * x`.

Maybe

`Maybe`, defined in the Prelude, describes an optional value. Either there is a value of the given type, or there isn't:

```
data Maybe a = Just a | Nothing
```

`Maybe` is one way of giving a type to an operation that may fail. For example, looking something up in a `List` (rather than a vector) may result in an out of bounds error:

```
list_lookup : Nat -> List a -> Maybe a
list_lookup _     Nil         = Nothing
list_lookup Z     (x :: xs) = Just x
list_lookup (S k) (x :: xs) = list_lookup k xs
```

The `maybe` function is used to process values of type `Maybe`, either by applying a function to the value, if there is one, or by providing a default value:

```
maybe : Lazy b -> Lazy (a -> b) -> Maybe a -> b
```

Note that the types of the first two arguments are wrapped in `Lazy`. Since only one of the two arguments will actually be used, we mark them as `Lazy` in case they are large expressions where it would be wasteful to compute and then discard them.

Tuples

Values can be paired with the following built-in data type:

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

As syntactic sugar, we can write `(a, b)` which, according to context, means either `Pair a b` or `MkPair a b`. Tuples can contain an arbitrary number of values, represented as nested pairs:

```
fred : (String, Int)
fred = ("Fred", 42)

jim : (String, Int, String)
jim = ("Jim", 25, "Cambridge")
```

```
*UsefulTypes> fst jim
"Jim" : String
*UsefulTypes> snd jim
(25, "Cambridge") : (Int, String)
*UsefulTypes> jim == ("Jim", (25, "Cambridge"))
True : Bool
```

Dependent Pairs

Dependent pairs allow the type of the second element of a pair to depend on the value of the first element:

```
data DPair : (a : Type) -> (p : a -> Type) -> Type where
   MkDPair : {p : a -> Type} -> (x : a) -> p x -> DPair a p
```

Again, there is syntactic sugar for this. `(x : a ** p)` is the type of a pair of A and P, where the name `x` can occur inside `p`. `( x ** p )` constructs a value of this type. For example, we can pair a number with a `Vect` of a particular length:

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

If you like, you can write it out the long way; the two are equivalent:

```
vec : DPair Nat (\n => Vect n Int)
vec = MkDPair 2 [3, 4]
```

The type checker could infer the value of the first element from the length of the vector. We can write an underscore `_` in place of values which we expect the type checker to fill in, so the above definition could also be written as:

```
vec : (n : Nat ** Vect n Int)
vec = (_ ** [3, 4])
```

We might also prefer to omit the type of the first element of the pair, since, again, it can be inferred:

```
vec : (n ** Vect n Int)
vec = (_ ** [3, 4])
```

One use for dependent pairs is to return values of dependent types where the index is not necessarily known in advance. For example, if we filter elements out of a `Vect` according to some predicate, we will not know in advance what the length of the resulting vector will be:

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
```

If the `Vect` is empty, the result is:

```
filter p Nil = (_ ** [])
```

In the `::` case, we need to inspect the result of a recursive call to `filter` to extract the length and the vector from the result. To do this, we use a `case` expression, which allows pattern matching on intermediate values:

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
filter p Nil = (_ ** [])
filter p (x :: xs)
    = case filter p xs of
```

(continues on next page)

(continued from previous page)

```
          (_ ** xs') => if p x then (_ ** x :: xs')
                                else (_ ** xs')
```

Dependent pairs are sometimes referred to as "Sigma types".

Records

Records are data types which collect several values (the record's *fields*) together. Idris provides syntax for defining records and automatically generating field access and update functions. Unlike the syntax used for data structures, records in Idris follow a different syntax to that seen with Haskell. For example, we can represent a person's name and age in a record:

```
record Person where
    constructor MkPerson
    firstName, middleName, lastName : String
    age : Int

fred : Person
fred = MkPerson "Fred" "Joe" "Bloggs" 30
```

The constructor name is provided using the `constructor` keyword, and the *fields* are then given which are in an indented block following the *where* keyword (here, `firstName`, `middleName`, `lastName`, and `age`). You can declare multiple fields on a single line, provided that they have the same type. The field names can be used to access the field values:

```
*Record> fred.firstName
"Fred" : String
*Record> fred.age
30 : Int
*Record> :t (.firstName)
Main.Person.(.firstName) : Person -> String
```

We can use prefix field projections, like in Haskell:

```
*Record> firstName fred
"Fred" : String
*Record> age fred
30 : Int
*Record> :t firstName
firstName : Person -> String
```

Prefix field projections can be disabled per record definition using pragma `%prefix_record_projections off`, which makes all subsequently defined records generate only dotted projections. This pragma has effect until the end of the module or until the closest occurrence of `%prefix_record_projections on`.

We can also use the field names to update a record (or, more precisely, produce a copy of the record with the given fields updated):

```
*Record> record { firstName = "Jim" } fred
MkPerson "Jim" "Joe" "Bloggs" 30 : Person
*Record> record { firstName = "Jim", age $= (+ 1) } fred
MkPerson "Jim" "Joe" "Bloggs" 31 : Person
```

The syntax `record { field = val, ... }` generates a function which updates the given fields in a record. `=` assigns a new value to a field, and `$=` applies a function to update its value.

Each record is defined in its own namespace, which means that field names can be reused in multiple

records.

Records, and fields within records, can have dependent types. Updates are allowed to change the type of a field, provided that the result is well-typed.

```
record Class where
    constructor ClassInfo
    students : Vect n Person
    className : String
```

It is safe to update the `students` field to a vector of a different length because it will not affect the type of the record:

```
addStudent : Person -> Class -> Class
addStudent p c = record { students = p :: students c } c


*Record> addStudent fred (ClassInfo [] "CS")
ClassInfo [MkPerson "Fred" "Joe" "Bloggs" 30] "CS" : Class
```

We could also use `$=` to define `addStudent` more concisely:

```
addStudent' : Person -> Class -> Class
addStudent' p c = record { students $= (p ::) } c
```

Nested record projection

Nested record fields can be accessed using the dot notation:

```
x.a.b.c
map (.a.b.c) xs
```

For the dot notation, there must be no spaces after the dots but there may be spaces before the dots. The composite projection must be parenthesised, otherwise `map .a.b.c xs` would be understood as `map.a.b.c xs`.

Nested record fields can be accessed using the prefix notation, too:

```
(c . b . a) x
map (c . b . a) xs
```

Dots with spaces around them stand for function composition operators.

Nested record update

Idris also provides a convenient syntax for accessing and updating nested records. For example, if a field is accessible with the expression `x.a.b.c`, it can be updated using the following syntax:

```
record { a.b.c = val } x
```

This returns a new record, with the field accessed by the path `a.b.c` set to `val`. The syntax is first class, i.e. `record { a.b.c = val }` itself has a function type.

The `$=` notation is also valid for nested record updates.

Dependent Records

Records can also be dependent on values. Records have *parameters*, which cannot be updated like the other fields. The parameters appear as arguments to the resulting type, and are written following the record type name. For example, a pair type could be defined as follows:

```
record Prod a b where
    constructor Times
    fst : a
    snd : b
```

Using the `Class` record from earlier, the size of the class can be restricted using a `Vect` and the size included in the type by parameterising the record with the size. For example:

```
record SizedClass (size : Nat) where
    constructor SizedClassInfo
    students : Vect size Person
    className : String
```

In the case of `addStudent` earlier, we can still add a student to a `SizedClass` since the size is implicit, and will be updated when a student is added:

```
addStudent : Person -> SizedClass n -> SizedClass (S n)
addStudent p c = record { students = p :: students c } c
```

In fact, the dependent pair type we have just seen is, in practice, defined as a record, with fields `fst` and `snd` which allow projecting values out of the pair:

```
record DPair a (p : a -> Type) where
  constructor MkDPair
  fst : a
  snd : p fst
```

It is possible to use record update syntax to update dependent fields, provided that all related fields are updated at once. For example:

```
cons : t -> (x : Nat ** Vect x t) -> (x : Nat ** Vect x t)
cons val xs
    = record { fst = S (fst xs),
               snd = (val :: snd xs) } xs
```

Or even:

```
cons' : t -> (x : Nat ** Vect x t) -> (x : Nat ** Vect x t)
cons' val
    = record { fst $= S,
               snd $= (val ::) }
```

3.10 More Expressions

`let` bindings

Intermediate values can be calculated using `let` bindings:

```
mirror : List a -> List a
mirror xs = let xs' = reverse xs in
                xs ++ xs'
```

We can do pattern matching in `let` bindings too. For example, we can extract fields from a record as follows, as well as by pattern matching at the top level:

```
data Person = MkPerson String Int

showPerson : Person -> String
showPerson p = let MkPerson name age = p in
                   name ++ " is " ++ show age ++ " years old"
```

List comprehensions

Idris provides *comprehension* notation as a convenient shorthand for building lists. The general form is:

```
[ expression | qualifiers ]
```

This generates the list of values produced by evaluating the `expression`, according to the conditions given by the comma separated `qualifiers`. For example, we can build a list of Pythagorean triples as follows:

```
pythag : Int -> List (Int, Int, Int)
pythag n = [ (x, y, z) | z <- [1..n], y <- [1..z], x <- [1..y],
                         x*x + y*y == z*z ]
```

The `[a..b]` notation is another shorthand which builds a list of numbers between `a` and `b`. Alternatively `[a,b..c]` builds a list of numbers between `a` and `c` with the increment specified by the difference between `a` and `b`. This works for type `Nat`, `Int` and `Integer`, using the `enumFromTo` and `enumFromThenTo` function from the prelude.

`case` expressions

Another way of inspecting intermediate values is to use a `case` expression. The following function, for example, splits a string into two at a given character:

```
splitAt : Char -> String -> (String, String)
splitAt c x = case break (== c) x of
                  (x, y) => (x, strTail y)
```

`break` is a library function which breaks a string into a pair of strings at the point where the given function returns true. We then deconstruct the pair it returns, and remove the first character of the second string.

A `case` expression can match several cases, for example, to inspect an intermediate value of type `Maybe a`. Recall `list_lookup` which looks up an index in a list, returning `Nothing` if the index is out of bounds. We can use this to write `lookup_default`, which looks up an index and returns a default value if the index is out of bounds:

```
lookup_default : Nat -> List a -> a -> a
lookup_default i xs def = case list_lookup i xs of
                               Nothing => def
                               Just x => x
```

If the index is in bounds, we get the value at that index, otherwise we get a default value:

```
*UsefulTypes> lookup_default 2 [3,4,5,6] (-1)
5 : Integer
```

(continues on next page)

(continued from previous page)

```
*UsefulTypes> lookup_default 4 [3,4,5,6] (-1)
-1 : Integer
```

3.11 Totality

Idris distinguishes between *total* and *partial* functions. A total function is a function that either:

- Terminates for all possible inputs, or
- Produces a non-empty, finite, prefix of a possibly infinite result

If a function is total, we can consider its type a precise description of what that function will do. For example, if we have a function with a return type of `String` we know something different, depending on whether or not it's total:

- If it's total, it will return a value of type `String` in finite time;
- If it's partial, then as long as it doesn't crash or enter an infinite loop, it will return a `String`.

Idris makes this distinction so that it knows which functions are safe to evaluate while type checking (as we've seen with *First Class Types* (page 10)). After all, if it tries to evaluate a function during type checking which doesn't terminate, then type checking won't terminate! Therefore, only total functions will be evaluated during type checking. Partial functions can still be used in types, but will not be evaluated further.

4 Interfaces

We often want to define functions which work across several different data types. For example, we would like arithmetic operators to work on `Int`, `Integer` and `Double` at the very least. We would like `==` to work on the majority of data types. We would like to be able to display different types in a uniform way.

To achieve this, we use *interfaces*, which are similar to type classes in Haskell or traits in Rust. To define an interface, we provide a collection of overloadable functions. A simple example is the `Show` interface, which is defined in the prelude and provides an interface for converting values to `String`:

```
interface Show a where
    show : a -> String
```

This generates a function of the following type (which we call a *method* of the `Show` interface):

```
show : Show a => a -> String
```

We can read this as: "under the constraint that `a` has an implementation of `Show`, take an input `a` and return a `String`." An implementation of an interface is defined by giving definitions of the methods of the interface. For example, the `Show` implementation for `Nat` could be defined as:

```
Show Nat where
    show Z = "Z"
    show (S k) = "s" ++ show k

Main> show (S (S (S Z)))
"sssZ" : String
```

Only one implementation of an interface can be given for a type — implementations may not overlap.

Implementation declarations can themselves have constraints. To help with resolution, the arguments of an implementation must be constructors (either data or type constructors) or variables (i.e. you cannot give an implementation for a function). For example, to define a `Show` implementation for vectors, we need to know that there is a `Show` implementation for the element type, because we are going to use it to convert each element to a `String`:

```
Show a => Show (Vect n a) where
    show xs = "[" ++ show' xs ++ "]" where
        show' : forall n . Vect n a -> String
        show' Nil        = ""
        show' (x :: Nil) = show x
        show' (x :: xs)  = show x ++ ", " ++ show' xs
```

Note that we need the explicit `forall n .` in the `show'` function because otherwise the `n` is already in scope, and fixed to the value of the top level `n`.

4.1 Default Definitions

The Prelude defines an `Eq` interface which provides methods for comparing values for equality or inequality, with implementations for all of the built-in types:

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool
```

To declare an implementation for a type, we have to give definitions of all of the methods. For example, for an implementation of `Eq` for `Nat`:

```
Eq Nat where
    Z     == Z     = True
    (S x) == (S y) = x == y
    Z     == (S y) = False
    (S x) == Z     = False

    x /= y = not (x == y)
```

It is hard to imagine many cases where the `/=` method will be anything other than the negation of the result of applying the `==` method. It is therefore convenient to give a default definition for each method in the interface declaration, in terms of the other method:

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

A minimal complete implementation of `Eq` requires either `==` or `/=` to be defined, but does not require both. If a method definition is missing, and there is a default definition for it, then the default is used instead.

4.2 Extending Interfaces

Interfaces can also be extended. A logical next step from an equality relation `Eq` is to define an ordering relation `Ord`. We can define an `Ord` interface which inherits methods from `Eq` as well as defining some of its own:

```
data Ordering = LT | EQ | GT

interface Eq a => Ord a where
    compare : a -> a -> Ordering

    (<) : a -> a -> Bool
    (>) : a -> a -> Bool
    (<=) : a -> a -> Bool
    (>=) : a -> a -> Bool
    max : a -> a -> a
    min : a -> a -> a
```

The `Ord` interface allows us to compare two values and determine their ordering. Only the `compare` method is required; every other method has a default definition. Using this we can write functions such as `sort`, a function which sorts a list into increasing order, provided that the element type of the list is in the `Ord` interface. We give the constraints on the type variables left of the fat arrow `=>`, and the function type to the right of the fat arrow:

```
sort : Ord a => List a -> List a
```

Functions, interfaces and implementations can have multiple constraints. Multiple constraints are written in brackets in a comma separated list, for example:

```
sortAndShow : (Ord a, Show a) => List a -> String
sortAndShow xs = show (sort xs)
```

Constraints are, like types, first class objects in the language. You can see this at the REPL:

```
Main> :t Ord
Prelude.Ord : Type -> Type
```

So, `(Ord a, Show a)` is an ordinary pair of `Types`, with two constraints as the first and second element of the pair.

Note: Interfaces and `mutual` blocks

Idris is strictly "define before use", except in `mutual` blocks. In a `mutual` block, Idris elaborates in two passes: types on the first pass and definitions on the second. When the mutual block contains an interface declaration, it elaborates the interface header but none of the method types on the first pass, and elaborates the method types and any default definitions on the second pass.

4.3 Quantities for Parameters

By default parameters that are not explicitly ascribed a type in an `interface` declaration are assigned the quantity `0`. This means that the parameter is not available to use at runtime in the methods' definitions.

For instance, `Show a` gives rise to a `0`-quantified type variable `a` in the type of the `show` method:

```
Main> :set showimplicits
Main> :t show
Prelude.show : {0 a : Type} -> Show a => a -> String
```

However some use cases require that some of the parameters are available at runtime. We may for instance want to declare an interface for `Storable` types. The constraint `Storable a size` means that we can store values of type `a` in a `Buffer` in exactly `size` bytes.

If the user provides a method to read a value for such a type `a` at a given offset, then we can read the `k` th element stored in the buffer by computing the appropriate offset from `k` and `size`. This is demonstrated by providing a default implementation for the method `peekElementOff` implemented in terms of `peekByteOff` and the parameter `size`.

```
data ForeignPtr : Type -> Type where
  MkFP : Buffer -> ForeignPtr a

interface Storable (0 a : Type) (size : Nat) | a where
  peekByteOff : HasIO io => ForeignPtr a -> Int -> io a

  peekElemOff : HasIO io => ForeignPtr a -> Int -> io a
  peekElemOff fp k = peekByteOff fp (k * cast size)
```

Note that `a` is explicitly marked as runtime irrelevant so that it is erased by the compiler. Equivalently we could have written `interface Storable a (size : Nat)`. The meaning of `| a` is explained in *Determining Parameters* (page 31).

4.4 Functors and Applicatives

So far, we have seen single parameter interfaces, where the parameter is of type `Type`. In general, there can be any number of parameters (even zero), and the parameters can have *any* type. If the type of the parameter is not `Type`, we need to give an explicit type declaration. For example, the `Functor` interface is defined in the prelude:

```
interface Functor (0 f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

A functor allows a function to be applied across a structure, for example to apply a function to every element in a `List`:

```
Functor List where
  map f []      = []
  map f (x::xs) = f x :: map f xs


Idris> map (*2) [1..10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20] : List Integer
```

Having defined `Functor`, we can define `Applicative` which abstracts the notion of function application:

```
infixl 2 <*>

interface Functor f => Applicative (0 f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b
```

4.5 Monads and `do`-notation

The `Monad` interface allows us to encapsulate binding and computation, and is the basis of `do`-notation introduced in Section *"do" notation* (page 13). It extends `Applicative` as defined above, and is defined as follows:

```
interface Applicative m => Monad (m : Type -> Type) where
    (>>=)  : m a -> (a -> m b) -> m b
```

Inside a `do` block, the following syntactic transformations are applied:

- `x <- v; e` becomes `v >>= (\x => e)`
- `v; e` becomes `v >>= (\_ => e)`
- `let x = v; e` becomes `let x = v in e`

`IO` has an implementation of `Monad`, defined using primitive functions. We can also define an implementation for `Maybe`, as follows:

```
Monad Maybe where
    Nothing  >>= k = Nothing
    (Just x) >>= k = k x
```

Using this we can, for example, define a function which adds two `Maybe Int`, using the monad to encapsulate the error handling:

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = do x' <- x -- Extract value from x
               y' <- y -- Extract value from y
               pure (x' + y') -- Add them
```

This function will extract the values from `x` and `y`, if they are both available, or return `Nothing` if one or both are not ("fail fast"). Managing the `Nothing` cases is achieved by the `>>=` operator, hidden by the `do` notation.

```
Main> m_add (Just 82) (Just 22)
Just 94
Main> m_add (Just 82) Nothing
Nothing
```

The translation of `do` notation is entirely syntactic, so there is no need for the `(>>=)` operator to be the operator defined in the `Monad` interface. Idris will, in general, try to disambiguate which `(>>=)` you mean by type, but you can explicitly choose with qualified do notation, for example:

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = Prelude.do
               x' <- x -- Extract value from x
               y' <- y -- Extract value from y
               pure (x' + y') -- Add them
```

The `Prelude.do` means that Idris will use the `(>>=)` operator defined in the `Prelude`.

Pattern Matching Bind

Sometimes we want to pattern match immediately on the result of a function in `do` notation. For example, let's say we have a function `readNumber` which reads a number from the console, returning a value of the form `Just x` if the number is valid, or `Nothing` otherwise:

```
import Data.Strings

readNumber : IO (Maybe Nat)
readNumber = do
  input <- getLine
  if all isDigit (unpack input)
     then pure (Just (stringToNatOrZ input))
     else pure Nothing
```

If we then use it to write a function to read two numbers, returning `Nothing` if neither are valid, then we would like to pattern match on the result of `readNumber`:

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers =
  do x <- readNumber
     case x of
          Nothing => pure Nothing
          Just x_ok => do y <- readNumber
                          case y of
                               Nothing => pure Nothing
                               Just y_ok => pure (Just (x_ok, y_ok))
```

If there's a lot of error handling, this could get deeply nested very quickly! So instead, we can combine the bind and the pattern match in one line. For example, we could try pattern matching on values of the form `Just x_ok`:

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers
  = do Just x_ok <- readNumber
       Just y_ok <- readNumber
       pure (Just (x_ok, y_ok))
```

There is still a problem, however, because we've now omitted the case for `Nothing` so `readNumbers` is no longer total! We can add the `Nothing` case back as follows:

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers
  = do Just x_ok <- readNumber
            | Nothing => pure Nothing
       Just y_ok <- readNumber
            | Nothing => pure Nothing
       pure (Just (x_ok, y_ok))
```

The effect of this version of `readNumbers` is identical to the first (in fact, it is syntactic sugar for it and directly translated back into that form). The first part of each statement (`Just x_ok <-` and `Just y_ok <-`) gives the preferred binding - if this matches, execution will continue with the rest of the `do` block. The second part gives the alternative bindings, of which there may be more than one.

`!`-notation

In many cases, using `do`-notation can make programs unnecessarily verbose, particularly in cases such as `m_add` above where the value bound is used once, immediately. In these cases, we can use a shorthand version, as follows:

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = pure (!x + !y)
```

The notation `!expr` means that the expression `expr` should be evaluated and then implicitly bound. Conceptually, we can think of `!` as being a prefix function with the following type:

```
(!) : m a -> a
```

Note, however, that it is not really a function, merely syntax! In practice, a subexpression `!expr` will lift `expr` as high as possible within its current scope, bind it to a fresh name `x`, and replace `!expr` with `x`. Expressions are lifted depth first, left to right. In practice, `!`-notation allows us to program in a more direct style, while still giving a notational clue as to which expressions are monadic.

For example, the expression:

```
let y = 94 in f !(g !(print y) !x)
```

is lifted to:

```
let y = 94 in do y' <- print y
                 x' <- x
                 g' <- g y' x'
                 f g'
```

Monad comprehensions

The list comprehension notation we saw in Section *More Expressions* (page 20) is more general, and applies to anything which has an implementation of both `Monad` and `Alternative`:

```
interface Applicative f => Alternative (0 f : Type -> Type) where
    empty : f a
    (<|>) : f a -> f a -> f a
```

In general, a comprehension takes the form `[ exp | qual1, qual2, ..., qualn ]` where `quali` can be one of:

- A generator `x <- e`
- A *guard*, which is an expression of type `Bool`
- A let binding `let x = e`

To translate a comprehension `[exp | qual1, qual2, ..., qualn]`, first any qualifier `qual` which is a *guard* is translated to `guard qual`, using the following function:

```
guard : Alternative f => Bool -> f ()
```

Then the comprehension is converted to `do` notation:

```
do { qual1; qual2; ...; qualn; pure exp; }
```

Using monad comprehensions, an alternative definition for `m_add` would be:

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = [ x' + y' | x' <- x, y' <- y ]
```

4.6 Interfaces and IO

In general, `IO` operations in the libraries aren't written using `IO` directly, but rather via the `HasIO` interface:

```
interface Monad io => HasIO io where
  liftIO : (1 _ : IO a) -> io a
```

`HasIO` explains, via `liftIO`, how to convert a primitive `IO` operation to an operation in some underlying type, as long as that type has a `Monad` implementation. These interface allows a programmer to define some more expressive notion of interactive program, while still giving direct access to `IO` primitives.

4.7 Idiom brackets

While `do` notation gives an alternative meaning to sequencing, idioms give an alternative meaning to *application*. The notation and larger example in this section is inspired by Conor McBride and Ross Paterson's paper "Applicative Programming with Effects"[1].

First, let us revisit `m_add` above. All it is really doing is applying an operator to two values extracted from `Maybe Int`. We could abstract out the application:

```
m_app : Maybe (a -> b) -> Maybe a -> Maybe b
m_app (Just f) (Just a) = Just (f a)
m_app _        _        = Nothing
```

Using this, we can write an alternative `m_add` which uses this alternative notion of function application, with explicit calls to `m_app`:

```
m_add' : Maybe Int -> Maybe Int -> Maybe Int
m_add' x y = m_app (m_app (Just (+)) x) y
```

Rather than having to insert `m_app` everywhere there is an application, we can use idiom brackets to do the job for us. To do this, we can give `Maybe` an implementation of `Applicative` as follows, where `<*>` is defined in the same way as `m_app` above (this is defined in the Idris library):

```
Applicative Maybe where
    pure = Just

    (Just f) <*> (Just a) = Just (f a)
    _        <*> _        = Nothing
```

Using `<*>` we can use this implementation as follows, where a function application `[| f a1 ...an |]` is translated into `pure f <*> a1 <*> ... <*> an`:

```
m_add' : Maybe Int -> Maybe Int -> Maybe Int
m_add' x y = [| x + y |]
```

An error-handling interpreter

Idiom notation is commonly useful when defining evaluators. McBride and Paterson describe such an evaluator[1], for a language similar to the following:

```
data Expr = Var String      -- variables
          | Val Int         -- values
          | Add Expr Expr   -- addition
```

Evaluation will take place relative to a context mapping variables (represented as `String`s) to `Int` values, and can possibly fail. We define a data type `Eval` to wrap an evaluator:

```
data Eval : Type -> Type where
     MkEval : (List (String, Int) -> Maybe a) -> Eval a
```

Wrapping the evaluator in a data type means we will be able to provide implementations of interfaces for it later. We begin by defining a function to retrieve values from the context during evaluation:

[1] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. J. Funct. Program. 18, 1 (January 2008), 1-13. DOI=10.1017/S0956796807006326 https://dx.doi.org/10.1017/S0956796807006326

```
fetch : String -> Eval Int
fetch x = MkEval (\e => fetchVal e) where
    fetchVal : List (String, Int) -> Maybe Int
    fetchVal [] = Nothing
    fetchVal ((v, val) :: xs) = if (x == v)
                                then (Just val)
                                else (fetchVal xs)
```

When defining an evaluator for the language, we will be applying functions in the context of an `Eval`, so it is natural to give `Eval` an implementation of `Applicative`. Before `Eval` can have an implementation of `Applicative` it is necessary for `Eval` to have an implementation of `Functor`:

```
Functor Eval where
    map f (MkEval g) = MkEval (\e => map f (g e))

Applicative Eval where
    pure x = MkEval (\e => Just x)

    (<*>) (MkEval f) (MkEval g) = MkEval (\x => app (f x) (g x)) where
        app : Maybe (a -> b) -> Maybe a -> Maybe b
        app (Just fx) (Just gx) = Just (fx gx)
        app _         _         = Nothing
```

Evaluating an expression can now make use of the idiomatic application to handle errors:

```
eval : Expr -> Eval Int
eval (Var x)   = fetch x
eval (Val x)   = [| x |]
eval (Add x y) = [| eval x + eval y |]

runEval : List (String, Int) -> Expr -> Maybe Int
runEval env e = case eval e of
    MkEval envFn => envFn env
```

For example:

```
InterpE> runEval [("x", 10), ("y",84)] (Add (Var "x") (Var "y"))
Just 94
InterpE> runEval [("x", 10), ("y",84)] (Add (Var "x") (Var "z"))
Nothing
```

4.8 Named Implementations

It can be desirable to have multiple implementations of an interface for the same type, for example to provide alternative methods for sorting or printing values. To achieve this, implementations can be *named* as follows:

```
[myord] Ord Nat where
   compare Z (S n)     = GT
   compare (S n) Z     = LT
   compare Z Z         = EQ
   compare (S x) (S y) = compare @{myord} x y
```

This declares an implementation as normal, but with an explicit name, `myord`. The syntax `compare @{myord}` gives an explicit implementation to `compare`, otherwise it would use the default implementation for `Nat`. We can use this, for example, to sort a list of `Nat` in reverse. Given the following list:

```
testList : List Nat
testList = [3,4,1]
```

We can sort it using the default `Ord` implementation, by using the `sort` function available with `import Data.List`, then we can try with the named implementation `myord` as follows, at the Idris prompt:

```
Main> show (sort testList)
"[1, 3, 4]"
Main> show (sort @{myord} testList)
"[4, 3, 1]"
```

Sometimes, we also need access to a named parent implementation. For example, the prelude defines the following `Semigroup` interface:

```
interface Semigroup ty where
  (<+>) : ty -> ty -> ty
```

Then it defines `Monoid`, which extends `Semigroup` with a "neutral" value:

```
interface Semigroup ty => Monoid ty where
  neutral : ty
```

We can define two different implementations of `Semigroup` and `Monoid` for `Nat`, one based on addition and one on multiplication:

```
[PlusNatSemi] Semigroup Nat where
  (<+>) x y = x + y

[MultNatSemi] Semigroup Nat where
  (<+>) x y = x * y
```

The neutral value for addition is `0`, but the neutral value for multiplication is `1`. It's important, therefore, that when we define implementations of `Monoid` they extend the correct `Semigroup` implementation. We can do this with a `using` clause in the implementation as follows:

```
[PlusNatMonoid] Monoid Nat using PlusNatSemi where
  neutral = 0

[MultNatMonoid] Monoid Nat using MultNatSemi where
  neutral = 1
```

The `using PlusNatSemi` clause indicates that `PlusNatMonoid` should extend `PlusNatSemi` specifically.

4.9 Determining Parameters

When an interface has more than one parameter, it can help resolution if the parameters used to find an implementation are restricted. For example:

```
interface Monad m => MonadState s (0 m : Type -> Type) | m where
  get : m s
  put : s -> m ()
```

In this interface, only `m` needs to be known to find an implementation of this interface, and `s` can then be determined from the implementation. This is declared with the `| m` after the interface declaration. We call `m` a *determining parameter* of the `MonadState` interface, because it is the parameter used to find an implementation. This is similar to the concept of *functional dependencies* in Haskell.

5 Modules and Namespaces

An Idris program consists of a collection of modules. Each module includes an optional `module` declaration giving the name of the module, a list of `import` statements giving the other modules which are to be imported, and a collection of declarations and definitions of types, interfaces and functions. For example, the listing below gives a module which defines a binary tree type `BTree` (in a file `BTree.idr`):

```
module BTree

public export
data BTree a = Leaf
             | Node (BTree a) a (BTree a)

export
insert : Ord a => a -> BTree a -> BTree a
insert x Leaf = Node Leaf x Leaf
insert x (Node l v r) = if (x < v) then (Node (insert x l) v r)
                                   else (Node l v (insert x r))

export
toList : BTree a -> List a
toList Leaf = []
toList (Node l v r) = BTree.toList l ++ (v :: BTree.toList r)

export
toTree : Ord a => List a -> BTree a
toTree [] = Leaf
toTree (x :: xs) = insert x (toTree xs)
```

The modifiers `export` and `public export` say which names are visible from other namespaces. These are explained further below.

Then, this gives a main program (in a file `bmain.idr`) which uses the `BTree` module to sort a list:

```
module Main

import BTree

main : IO ()
main = do let t = toTree [1,8,2,7,9,3]
          print (BTree.toList t)
```

The same names can be defined in multiple modules: names are *qualified* with the name of the module. The names defined in the `BTree` module are, in full:

- `BTree.BTree`
- `BTree.Leaf`
- `BTree.Node`
- `BTree.insert`
- `BTree.toList`
- `BTree.toTree`

If names are otherwise unambiguous, there is no need to give the fully qualified name. Names can be disambiguated either by giving an explicit qualification, using the `with` keyword, or according to their type.

The `with` keyword in expressions comes in two variants:

- `with BTree.insert (insert x empty)` for one name
- `with [BTree.insert, BTree.empty] (insert x empty)` for multiple names

This is particularly useful with `do` notation, where it can often improve error messages: `with MyModule. (>>=) do ...`

There is no formal link between the module name and its filename, although it is generally advisable to use the same name for each. An `import` statement refers to a filename, using dots to separate directories. For example, `import foo.bar` would import the file `foo/bar.idr`, which would conventionally have the module declaration `module foo.bar`. The only requirement for module names is that the main module, with the `main` function, must be called `Main` — although its filename need not be `Main.idr`.

5.1 Export Modifiers

Idris allows for fine-grained control over the visibility of a namespace's contents. By default, all names defined in a namespace are kept private. This aids in specification of a minimal interface and for internal details to be left hidden. Idris allows for functions, types, and interfaces to be marked as: `private`, `export`, or `public export`. Their general meaning is as follows:

- `private` meaning that it is not exported at all. This is the default.
- `export` meaning that its top level type is exported.
- `public export` meaning that the entire definition is exported.

A further restriction in modifying the visibility is that definitions must not refer to anything within a lower level of visibility. For example, `public export` definitions cannot use `private` or `export` names, and `export` types cannot use `private` names. This is to prevent private names leaking into a module's interface.

Meaning for Functions

- `export` the type is exported
- `public export` the type and definition are exported, and the definition can be used after it is imported. In other words, the definition itself is considered part of the module's interface. The long name `public export` is intended to make you think twice about doing this.

Note: Type synonyms in Idris are created by writing a function. When setting the visibility for a module, it is usually a good idea to `public export` all type synonyms if they are to be used outside the module. Otherwise, Idris won't know what the synonym is a synonym for.

Since `public export` means that a function's definition is exported, this effectively makes the function definition part of the module's API. Therefore, it's generally a good idea to avoid using `public export` for functions unless you really mean to export the full definition.

Note: *For beginners*: If the function needs to be accessed only at runtime, use `export`. However, if it's also meant to be used at *compile* time (e.g. to prove a theorem), use `public export`. For example, consider the function `plus : Nat -> Nat -> Nat` discussed previously, and the following theorem: `thm : plus Z m = m`. In order to prove it, the type checker needs to reduce `plus Z m` to `m` (and hence obtain `thm : m = m`). To achieve this, it will need access to the *definition* of `plus`, which includes the

equation `plus Z m = m`. Therefore, in this case, `plus` has to be marked as `public export`.

Meaning for Data Types

For data types, the meanings are:

- `export` the type constructor is exported
- `public export` the type constructor and data constructors are exported

Meaning for Interfaces

For interfaces, the meanings are:

- `export` the interface name is exported
- `public export` the interface name, method names and default definitions are exported

Propagating Inner Module API's

Additionally, a module can re-export a module it has imported, by using the `public` modifier on an `import`. For example:

```
module A

import B
import public C
```

The module `A` will export the name `a`, as well as any public or abstract names in module `C`, but will not re-export anything from module `B`.

Renaming imports

Sometimes it is convenient to be able to access the names in another module via a different namespace (typically, a shorter one). For this, you can use *import... as*. For example:

```
module A

import Data.List as L
```

This module `A` has access to the exported names from module `Data.List`, but can also explicitly access them via the module name `L`. `import...as` can also be combined with `import public` to create a module which exports a larger API from other sub-modules:

```
module Books

import public Books.Hardback as Books
import public Books.Comic as Books
```

Here, any module which imports `Books` will have access to the exported interfaces of `Books.Hardback` and `Books.Comic` both under the namespace `Books`.

5.2 Explicit Namespaces

Defining a module also defines a namespace implicitly. However, namespaces can also be given *explicitly*. This is most useful if you wish to overload names within the same module:

```
module Foo

namespace X
  export
  test : Int -> Int
  test x = x * 2

namespace Y
  export
  test : String -> String
  test x = x ++ x
```

This (admittedly contrived) module defines two functions with fully qualified names `Foo.X.test` and `Foo.Y.test`, which can be disambiguated by their types:

```
*Foo> test 3
6 : Int
*Foo> test "foo"
"foofoo" : String
```

The export rules, `public export` and `export`, are *per namespace*, not *per file*, so the two `test` definitions above need the `export` flag to be visible outside their own namespaces.

5.3 Parameterised blocks

Groups of functions can be parameterised over a number of arguments using a `parameters` declaration, for example:

```
parameters (x : Nat, y : Nat)
  addAll : Nat -> Nat
  addAll z = x + y + z
```

The effect of a `parameters` block is to add the declared parameters to every function, type and data constructor within the block. Specifically, adding the parameters to the front of the argument list. Outside the block, the parameters must be given explicitly. The `addAll` function, when called from the REPL, will thus have the following type signature.

```
*params> :t addAll
addAll : Nat -> Nat -> Nat -> Nat
```

and the following definition.

```
addAll : (x : Nat) -> (y : Nat) -> (z : Nat) -> Nat
addAll x y z = x + y + z
```

Parameters blocks can be nested, and can also include data declarations, in which case the parameters are added explicitly to all type and data constructors. They may also be dependent types with implicit arguments:

```
parameters (y : Nat, xs : Vect x a)
  data Vects : Type -> Type where
    MkVects : Vect y a -> Vects a
```

(continues on next page)

(continued from previous page)

```
  append : Vects a -> Vect (x + y) a
  append (MkVects ys) = xs ++ ys
```

To use `Vects` or `append` outside the block, we must also give the `xs` and `y` arguments. Here, we can use placeholders for the values which can be inferred by the type checker:

```
Main> show (append _ _ (MkVects _ [1,2,3] [4,5,6]))
"[1, 2, 3, 4, 5, 6]"
```

6 Multiplicities

Idris 2 is based on Quantitative Type Theory (QTT), a core language developed by Bob Atkey and Conor McBride. In practice, this means that every variable in Idris 2 has a *quantity* associated with it. A quantity is one of:

- `0`, meaning that the variable is *erased* at run time
- `1`, meaning that the variable is used *exactly once* at run time
- *Unrestricted*, which is the same behaviour as Idris 1

We can see the multiplicities of variables by inspecting holes. For example, if we have the following skeleton definition of `append` on vectors...

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append xs ys = ?append_rhs
```

...we can look at the hole `append_rhs`:

```
Main> :t append_rhs
 0 m : Nat
 0 a : Type
 0 n : Nat
   ys : Vect m a
   xs : Vect n a
------------------------------------
append_rhs : Vect (plus n m) a
```

The `0` next to `m`, `a` and `n` mean that they are in scope, but there will be `0` occurrences at run-time. That is, it is **guaranteed** that they will be erased at run-time.

Multiplicities can be explicitly written in function types as follows:

- `ignoreN : (0 n : Nat) -> Vect n a -> Nat` - this function cannot look at `n` at run time
- `duplicate : (1 x : a) -> (a, a)` - this function must use `x` exactly once (so good luck implementing it, by the way. There is no implementation because it would need to use `x` twice!)

If there is no multiplicity annotation, the argument is unrestricted. If, on the other hand, a name is implicitly bound (like `a` in both examples above) the argument is erased. So, the above types could also be written as:

- `ignoreN : {0 a : _} -> (0 n : Nat) -> Vect n a -> Nat`
- `duplicate : {0 a : _} -> (1 x : a) -> (a, a)`

This section describes what this means for your Idris 2 programs in practice, with several examples. In particular, it describes:

- *Erasure* (page 41) - how to know what is relevant at run time and what is erased
- *Linearity* (page 37) - using the type system to encode *resource usage protocols*
- *Pattern Matching on Types* (page 42) - truly first class types

The most important of these for most programs, and the thing you'll need to know about if converting Idris 1 programs to work with Idris 2, is *erasure* (page 41). The most interesting, however, and the thing which gives Idris 2 much more expressivity, is *linearity* (page 37), so we'll start there.

6.1 Linearity

The `1` multiplicity expresses that a variable must be used exactly once. By "used" we mean either:

- if the variable is a data type or primitive value, it is pattern matched against, ex. by being the subject of a *case* statement, or a function argument that is pattern matched against, etc.,
- if the variable is a function, that function is applied (i.e. ran with an argument)

First, we'll see how this works on some small examples of functions and data types, then see how it can be used to encode *resource protocols* (page 39).

Above, we saw the type of `duplicate`. Let's try to write it interactively, and see what goes wrong. We'll start by giving the type and a skeleton definition with a hole

```
duplicate : (1 x : a) -> (a, a)
duplicate x = ?help
```

Checking the type of a hole tells us the multiplicity of each variable in scope. If we check the type of `?help` we'll see that we can't use `a` at run time, and we have to use `x` exactly once:

```
Main> :t help
 0 a : Type
 1 x : a
------------------------------------
help : (a, a)
```

If we use `x` for one part of the pair...

```
duplicate : (1 x : a) -> (a, a)
duplicate x = (x, ?help)
```

...then the type of the remaining hole tells us we can't use it for the other:

```
Main> :t help
 0 a : Type
 0 x : a
------------------------------------
help : a
```

The same happens if we try defining `duplicate x = (?help, x)` (try it!).

In order to avoid parsing ambiguities, if you give an explicit multiplicity for a variable as with the argument to `duplicate`, you need to give it a name too. But, if the name isn't used in the scope of the type, you can use `_` instead of a name, as follows:

```
duplicate : (1 _ : a) -> (a, a)
```

The intution behind multiplicity `1` is that if we have a function with a type of the following form...

```
f : (1 x : a) -> b
```

...then the guarantee given by the type system is that *if* `f x` *is used exactly once, then* `x` *is used exactly once*. So, if we insist on trying to define `duplicate`...:

```
duplicate x = (x, x)
```

...then Idris will complain:

```
pmtype.idr:2:15--8:1:While processing right hand side of Main.duplicate at pmtype.idr:2:1--8:1:
There are 2 uses of linear name x
```

A similar intuition applies for data types. Consider the following types, `Lin` which wraps an argument that must be used once, and `Unr` which wraps an argument with unrestricted use

```
data Lin : Type -> Type where
     MkLin : (1 _ : a) -> Lin a

data Unr : Type -> Type where
     MkUnr : a -> Unr a
```

If `MkLin x` is used once, then `x` is used once. But if `MkUnr x` is used once, there is no guarantee on how often `x` is used. We can see this a bit more clearly by starting to write projection functions for `Lin` and `Unr` to extract the argument

```
getLin : (1 _ : Lin a) -> a
getLin (MkLin x) = ?howmanyLin

getUnr : (1 _ : Unr a) -> a
getUnr (MkUnr x) = ?howmanyUnr
```

Checking the types of the holes shows us that, for `getLin`, we must use `x` exactly once (Because the `val` argument is used once, by pattern matching on it as `MkLin x`, and if `MkLin x` is used once, `x` must be used once):

```
Main> :t howmanyLin
 0 a : Type
 1 x : a
------------------------------------
howmanyLin : a
```

For `getUnr`, however, we still have to use `val` once, again by pattern matching on it, but using `MkUnr x` once doesn't place any restrictions on `x`. So, `x` has unrestricted use in the body of `getUnr`:

```
Main> :t howmanyUnr
 0 a : Type
   x : a
------------------------------------
howmanyUnr : a
```

If `getLin` has an unrestricted argument...

```
getLin : Lin a -> a
getLin (MkLin x) = ?howmanyLin
```

. . . then `x` is unrestricted in `howmanyLin`:

```
Main> :t howmanyLin
 0 a : Type
   x : a
-----------------------------------
howmanyLin : a
```

Remember the intuition from the type of `MkLin` is that if `MkLin x` is used exactly once, `x` is used exactly once. But, we didn't say that `MkLin x` would be used exactly once, so there is no restriction on `x`.

Resource protocols

One way to take advantage of being able to express linear usage of an argument is in defining resource usage protocols, where we can use linearity to ensure that any unique external resource has only one instance, and we can use functions which are linear in their arguments to represent state transitions on that resource. A door, for example, can be in one of two states, `Open` or `Closed`

```
data DoorState = Open | Closed

data Door : DoorState -> Type where
     MkDoor : (doorId : Int) -> Door st
```

(Okay, we're just pretending here - imagine the `doorId` is a reference to an external resource!)

We can define functions for opening and closing the door which explicitly describe how they change the state of a door, and that they are linear in the door

```
openDoor : (1 d : Door Closed) -> Door Open
closeDoor : (1 d : Door Open) -> Door Closed
```

Remember, the intuition is that if `openDoor d` is used exactly once, then `d` is used exactly once. So, provided that a door `d` has multiplicity `1` when it's created, we *know* that once we call `openDoor` on it, we won't be able to use `d` again. Given that `d` is an external resource, and `openDoor` has changed it's state, this is a good thing!

We can ensure that any door we create has multiplicity `1` by creating them with a `newDoor` function with the following type

```
newDoor : (1 p : (1 d : Door Closed) -> IO ()) -> IO ()
```

That is, `newDoor` takes a function, which it runs exactly once. That function takes a door, which is used exactly once. We'll run it in `IO` to suggest that there is some interaction with the outside world going on when we create the door. Since the multiplicity `1` means the door has to be used exactly once, we need to be able to delete the door when we're finished

```
deleteDoor : (1 d : Door Closed) -> IO ()
```

So an example correct door protocol usage would be

```
doorProg : IO ()
doorProg
    = newDoor $ \d =>
          let d' = openDoor d
              d'' = closeDoor d' in
              deleteDoor d''
```

It's instructive to build this program interactively, with holes along the way, and see how the multiplicities

of `d`, `d'` etc change. For example

```
doorProg : IO ()
doorProg
    = newDoor $ \d =>
          let d' = openDoor d in
              ?whatnow
```

Checking the type of `?whatnow` shows that `d` is now spent, but we still have to use `d'` exactly once:

```
Main> :t whatnow
 0 d : Door Closed
 1 d' : Door Open
------------------------------------
whatnow : IO ()
```

Note that the `0` multiplicity for `d` means that we can still *talk* about it - in particular, we can still reason about it in types - but we can't use it again in a relevant position in the rest of the program. It's also fine to shadow the name `d` throughout

```
doorProg : IO ()
doorProg
    = newDoor $ \d =>
          let d = openDoor d
              d = closeDoor d in
              deleteDoor d
```

If we don't follow the protocol correctly - create the door, open it, close it, then delete it - then the program won't type check. For example, we can try not to delete the door before finishing

```
doorProg : IO ()
doorProg
    = newDoor $ \d =>
          let d' = openDoor d
              d'' = closeDoor d' in
              putStrLn "What could possibly go wrong?"
```

This gives the following error:

```
Door.idr:15:19--15:38:While processing right hand side of Main.doorProg at Door.idr:13:1--17:1:
There are 0 uses of linear name d''
```

There's a lot more to be said about the details here! But, this shows at a high level how we can use linearity to capture resource usage protocols at the type level. If we have an external resource which is guaranteed to be used linearly, like `Door`, we don't need to run operations on that resource in an `IO` monad, since we're already enforcing an ordering on operations and don't have access to any out of date resource states. This is similar to the way interactive programs work in the Clean programming language, and in fact is how `IO` is implemented internally in Idris 2, with a special `%World` type for representing the state of the outside world that is always used linearly

```
public export
data IORes : Type -> Type where
     MkIORes : (result : a) -> (1 x : %World) -> IORes a

export
data IO : Type -> Type where
     MkIO : (1 fn : (1 x : %World) -> IORes a) -> IO a
```

Having multiplicities in the type system raises several interesting questions, such as:

- Can we use linearity information to inform memory management and, for example, have type level guarantees about functions which will not need to perform garbage collection?
- How should multiplicities be incorporated into interfaces such as `Functor`, `Applicative` and `Monad`?
- If we have `0`, and `1` as multiplicities, why stop there? Why not have `2`, `3` and more (like Granule)
- What about multiplicity polymorphism, as in the Linear Haskell proposal?
- Even without all of that, what can we do *now*?

6.2 Erasure

The `1` multiplicity give us many possibilities in the kinds of properties we can express. But, the `0` multiplicity is perhaps more important in that it allows us to be precise about which values are relevant at run time, and which are compile time only (that is, which are erased). Using the `0` multiplicity means a function's type now tells us exactly what it needs at run time.

For example, in Idris 1 you could get the length of a vector as follows

```
vlen : Vect n a -> Nat
vlen {n} xs = n
```

This is fine, since it runs in constant time, but the trade off is that `n` has to be available at run time, so at run time we always need the length of the vector to be available if we ever call `vlen`. Idris 1 can infer whether the length is needed, but there's no easy way for a programmer to be sure.

In Idris 2, we need to state explicitly that `n` is needed at run time

```
vlen : {n : Nat} -> Vect n a -> Nat
vlen xs = n
```

(Incidentally, also note that in Idris 2, names bound in types are also available in the definition without explicitly rebinding them.)

This also means that when you call `vlen`, you need the length available. For example, this will give an error

```
sumLengths : Vect m a -> Vect n a -> Nat
sumLengths xs ys = vlen xs + vlen ys
```

Idris 2 reports:

```
vlen.idr:7:20--7:28:While processing right hand side of Main.sumLengths at vlen.idr:7:1--10:1:
m is not accessible in this context
```

This means that it needs to use `m` as an argument to pass to `vlen xs`, where it needs to be available at run time, but `m` is not available in `sumLengths` because it has multiplicity `0`.

We can see this more clearly by replacing the right hand side of `sumLengths` with a hole...

```
sumLengths : Vect m a -> Vect n a -> Nat
sumLengths xs ys = ?sumLengths_rhs
```

...then checking the hole's type at the REPL:

```
Main> :t sumLengths_rhs
 0 n : Nat
```

(continues on next page)

(continued from previous page)

```
 0 a : Type
 0 m : Nat
   ys : Vect n a
   xs : Vect m a
------------------------------------
sumLengths_rhs : Nat
```

Instead, we need to give bindings for `m` and `n` with unrestricted multiplicity

```
sumLengths : {m, n : _} -> Vect m a -> Vect n a -> Nat
sumLengths xs ys = vlen xs + vlen xs
```

Remember that giving no multiplicity on a binder, as with `m` and `n` here, means that the variable has unrestricted usage.

If you're converting Idris 1 programs to work with Idris 2, this is probably the biggest thing you need to think about. It is important to note, though, that if you have bound implicits, such as...

```
excitingFn : {t : _} -> Coffee t -> Moonbase t
```

...then it's a good idea to make sure `t` really is needed, or performance might suffer due to the run time building the instance of `t` unnecessarily!

One final note on erasure: it is an error to try to pattern match on an argument with multiplicity `0`, unless its value is inferrable from elsewhere. So, the following definition is rejected

```
badNot : (0 x : Bool) -> Bool
badNot False = True
badNot True = False
```

This is rejected with the error:

```
badnot.idr:2:1--3:1:Attempt to match on erased argument False in
Main.badNot
```

The following, however, is fine, because in `sNot`, even though we appear to match on the erased argument `x`, its value is uniquely inferrable from the type of the second argument

```
data SBool : Bool -> Type where
     SFalse : SBool False
     STrue  : SBool True

sNot : (0 x : Bool) -> SBool x -> Bool
sNot False SFalse = True
sNot True  STrue  = False
```

Experience with Idris 2 so far suggests that, most of the time, as long as you're using unbound implicits in your Idris 1 programs, they will work without much modification in Idris 2. The Idris 2 type checker will point out where you require an unbound implicit argument at run time - sometimes this is both surprising and enlightening!

6.3 Pattern Matching on Types

One way to think about dependent types is to think of them as "first class" objects in the language, in that they can be assigned to variables, passed around and returned from functions, just like any other construct. But, if they're truly first class, we should be able to pattern match on them too! Idris 2 allows us to do this. For example

```
showType : Type -> String
showType Int = "Int"
showType (List a) = "List of " ++ showType a
showType _ = "something else"
```

We can try this as follows:

```
Main> showType Int
"Int"
Main> showType (List Int)
"List of Int"
Main> showType (List Bool)
"List of something else"
```

Pattern matching on function types is interesting, because the return type may depend on the input value. For example, let's add a case to `showType`

```
showType (Nat -> a) = ?help
```

Inspecting the type of `help` tells us:

```
Main> :t help
   a : Nat -> Type
------------------------------------
help : String
```

So, the return type `a` depends on the input value of type `Nat`, and we'll need to come up with a value to use `a`, for example

```
showType (Nat -> a) = "Function from Nat to " ++ showType (a Z)
```

Note that multiplicities on the binders, and the ability to pattern match on *non-erased* types mean that the following two types are distinct

```
id : a -> a
notId : {a : Type} -> a -> a
```

In the case of `notId`, we can match on `a` and get a function which is certainly not the identity function

```
notId {a = Int} x = x + 1
notId x = x
```

```
Main> notId 93
94
Main> notId "???"
"???"
```

There is an important consequence of being able to distinguish between relevant and irrelevant type arguments, which is that a function is *only* parametric in `a` if `a` has multiplicity `0`. So, in the case of `notId`, `a` is *not* a parameter, and so we can't draw any conclusions about the way the function will behave because it is polymorphic, because the type tells us it might pattern match on `a`.

On the other hand, it is merely a coincidence that, in non-dependently typed languages, types are *irrelevant* and get erased, and values are *relevant* and remain at run time. Idris 2, being based on QTT, allows us to make the distinction between relevant and irrelevant arguments precise. Types can be relevant, values (such as the `n` index to vectors) can be irrelevant.

For more details on multiplicities, see Idris 2: Quantitative Type Theory in Action.

7 Packages

Idris includes a simple build system for building packages and executables from a named package description file. These files can be used with the Idris compiler to manage the development process.

7.1 Package Descriptions

A package description includes the following:

- A header, consisting of the keyword `package` followed by a package name. Package names can be any valid Idris identifier. The iPKG format also takes a quoted version that accepts any valid filename.
- Fields describing package contents, `<field> = <value>`.

At least one field must be the modules field, where the value is a comma separated list of modules. For example, given an idris package `maths` that has modules `Maths.idr`, `Maths.NumOps.idr`, `Maths.BinOps.idr`, and `Maths.HexOps.idr`, the corresponding package file would be:

```
package maths

modules = Maths
        , Maths.NumOps
        , Maths.BinOps
        , Maths.HexOps
```

Other examples of package files can be found in the `libs` directory of the main Idris repository, and in third-party libraries.

7.2 Using Package files

Idris itself is aware about packages, and special commands are available to help with, for example, building packages, installing packages, and cleaning packages. For instance, given the `maths` package from earlier we can use Idris as follows:

- `idris2 --build maths.ipkg` will build all modules in the package
- `idris2 --install maths.ipkg` will install the package, making it accessible by other Idris libraries and programs.
- `idris2 --clean maths.ipkg` will delete all intermediate code and executable files generated when building.

Once the maths package has been installed, the command line option `--package maths` makes it accessible (abbreviated to `-p maths`). For example:

```
idris2 -p maths Main.idr
```

7.3 Package Dependencies Using Atom

If you are using the Atom editor and have a dependency on another package, corresponding to for instance `import Lightyear` or `import Pruviloj`, you need to let Atom know that it should be loaded. The easiest way to accomplish that is with a .ipkg file. The general contents of an ipkg file will be described in the next section of the tutorial, but for now here is a simple recipe for this trivial case:

- Create a folder myProject.
- Add a file myProject.ipkg containing just a couple of lines:

```
package myProject

depends = pruviloj, lightyear
```

- In Atom, use the File menu to Open Folder myProject.

8 Example: The Well-Typed Interpreter

In this section, we'll use the features we've seen so far to write a larger example, an interpreter for a simple functional programming language, with variables, function application, binary operators and an `if...then...else` construct. We will use the dependent type system to ensure that any programs which can be represented are well-typed.

8.1 Representing Languages

First, let us define the types in the language. We have integers, booleans, and functions, represented by `Ty`:

```
data Ty = TyInt | TyBool | TyFun Ty Ty
```

We can write a function to translate these representations to a concrete Idris type — remember that types are first class, so can be calculated just like any other value:

```
interpTy : Ty -> Type
interpTy TyInt       = Integer
interpTy TyBool      = Bool
interpTy (TyFun a t) = interpTy a -> interpTy t
```

We're going to define a representation of our language in such a way that only well-typed programs can be represented. We'll index the representations of expressions by their type, **and** the types of local variables (the context). The context can be represented using the `Vect` data type, so we'll need to import `Data.Vect` at the top of our source file:

```
import Data.Vect
```

Expressions are indexed by the types of the local variables, and the type of the expression itself:

```
data Expr : Vect n Ty -> Ty -> Type
```

The full representation of expressions is:

```
data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: ctxt) t
    Pop  : HasType k ctxt t -> HasType (FS k) (u :: ctxt) t

data Expr : Vect n Ty -> Ty -> Type where
    Var : HasType i ctxt t -> Expr ctxt t
    Val : (x : Integer) -> Expr ctxt TyInt
    Lam : Expr (a :: ctxt) t -> Expr ctxt (TyFun a t)
    App : Expr ctxt (TyFun a t) -> Expr ctxt a -> Expr ctxt t
    Op  : (interpTy a -> interpTy b -> interpTy c) ->
```

(continues on next page)

(continued from previous page)

```
          Expr ctxt a -> Expr ctxt b -> Expr ctxt c
    If  : Expr ctxt TyBool ->
          Lazy (Expr ctxt a) ->
          Lazy (Expr ctxt a) -> Expr ctxt a
```

The code above makes use of the `Vect` and `Fin` types from the base libraries. `Fin` is available as part of `Data.Vect`. Throughout, `ctxt` refers to the local variable context.

Since expressions are indexed by their type, we can read the typing rules of the language from the definitions of the constructors. Let us look at each constructor in turn.

We use a nameless representation for variables — they are *de Bruijn indexed*. Variables are represented by a proof of their membership in the context, `HasType i ctxt T`, which is a proof that variable `i` in context `ctxt` has type `T`. This is defined as follows:

```
data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: ctxt) t
    Pop  : HasType k ctxt t -> HasType (FS k) (u :: ctxt) t
```

We can treat *Stop* as a proof that the most recently defined variable is well-typed, and *Pop n* as a proof that, if the `n`th most recently defined variable is well-typed, so is the `n+1`th. In practice, this means we use `Stop` to refer to the most recently defined variable, `Pop Stop` to refer to the next, and so on, via the `Var` constructor:

```
Var : HasType i ctxt t -> Expr ctxt t
```

So, in an expression `\x. \y. x y`, the variable `x` would have a de Bruijn index of 1, represented as `Pop Stop`, and `y 0`, represented as `Stop`. We find these by counting the number of lambdas between the definition and the use.

A value carries a concrete representation of an integer:

```
Val : (x : Integer) -> Expr ctxt TyInt
```

A lambda creates a function. In the scope of a function of type `a -> t`, there is a new local variable of type `a`, which is expressed by the context index:

```
Lam : Expr (a :: ctxt) t -> Expr ctxt (TyFun a t)
```

Function application produces a value of type `t` given a function from `a` to `t` and a value of type `a`:

```
App : Expr ctxt (TyFun a t) -> Expr ctxt a -> Expr ctxt t
```

We allow arbitrary binary operators, where the type of the operator informs what the types of the arguments must be:

```
Op : (interpTy a -> interpTy b -> interpTy c) ->
     Expr ctxt a -> Expr ctxt b -> Expr ctxt c
```

Finally, `If` expressions make a choice given a boolean. Each branch must have the same type, and we will evaluate the branches lazily so that only the branch which is taken need be evaluated:

```
If : Expr ctxt TyBool ->
     Lazy (Expr ctxt a) ->
     Lazy (Expr ctxt a) ->
     Expr ctxt a
```

8.2 Writing the Interpreter

When we evaluate an `Expr`, we'll need to know the values in scope, as well as their types. `Env` is an environment, indexed over the types in scope. Since an environment is just another form of list, albeit with a strongly specified connection to the vector of local variable types, we use the usual `::` and `Nil` constructors so that we can use the usual list syntax. Given a proof that a variable is defined in the context, we can then produce a value from the environment:

```
data Env : Vect n Ty -> Type where
    Nil  : Env Nil
    (::) : interpTy a -> Env ctxt -> Env (a :: ctxt)

lookup : HasType i ctxt t -> Env ctxt -> interpTy t
lookup Stop    (x :: xs) = x
lookup (Pop k) (x :: xs) = lookup k xs
```

Given this, an interpreter is a function which translates an `Expr` into a concrete Idris value with respect to a specific environment:

```
interp : Env ctxt -> Expr ctxt t -> interpTy t
```

The complete interpreter is defined as follows, for reference. For each constructor, we translate it into the corresponding Idris value:

```
interp env (Var i)     = lookup i env
interp env (Val x)     = x
interp env (Lam sc)    = \x => interp (x :: env) sc
interp env (App f s)   = interp env f (interp env s)
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)  = if interp env x then interp env t
                                         else interp env e
```

Let us look at each case in turn. To translate a variable, we simply look it up in the environment:

```
interp env (Var i) = lookup i env
```

To translate a value, we just return the concrete representation of the value:

```
interp env (Val x) = x
```

Lambdas are more interesting. In this case, we construct a function which interprets the scope of the lambda with a new value in the environment. So, a function in the object language is translated to an Idris function:

```
interp env (Lam sc) = \x => interp (x :: env) sc
```

For an application, we interpret the function and its argument and apply it directly. We know that interpreting `f` must produce a function, because of its type:

```
interp env (App f s) = interp env f (interp env s)
```

Operators and conditionals are, again, direct translations into the equivalent Idris constructs. For operators, we apply the function to its operands directly, and for `If`, we apply the Idris `if...then...else` construct directly.

```
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)  = if interp env x then interp env t
                                         else interp env e
```

8.3 Testing

We can make some simple test functions. Firstly, adding two inputs `\x. \y. y + x` is written as follows:

```
add : Expr ctxt (TyFun TyInt (TyFun TyInt TyInt))
add = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))
```

More interestingly, a factorial function `fact` (e.g. `\x. if (x == 0) then 1 else (fact (x-1) * x)`), can be written as:

```
fact : Expr ctxt (TyFun TyInt TyInt)
fact = Lam (If (Op (==) (Var Stop) (Val 0))
               (Val 1)
               (Op (*) (App fact (Op (-) (Var Stop) (Val 1)))
                       (Var Stop)))
```

8.4 Running

To finish, we write a `main` program which interprets the factorial function on user input:

```
main : IO ()
main = do putStr "Enter a number: "
          x <- getLine
          printLn (interp [] fact (cast x))
```

Here, `cast` is an overloaded function which converts a value from one type to another if possible. Here, it converts a string to an integer, giving 0 if the input is invalid. An example run of this program at the Idris interactive environment is:

```
$ idris2 interp.idr
     ____    __     _         ___
    /  _/___/ /____(_)____   |__ \
    / // __  / ___/ / ___/   __/ /     Version 0.3.0
  _/ // /_/ / /  / (__  )   / __/      https://www.idris-lang.org
 /___/\__,_/_/  /_/____/   /____/      Type :? for help

Welcome to Idris 2.  Enjoy yourself!
Main> :exec main
Enter a number: 6
720
```

Aside: `cast`

The prelude defines an interface `Cast` which allows conversion between types:

```
interface Cast from to where
    cast : from -> to
```

It is a *multi-parameter* interface, defining the source type and object type of the cast. It must be possible for the type checker to infer *both* parameters at the point where the cast is applied. There are casts defined between all of the primitive types, as far as they make sense.

9 Views and the "`with`" rule

Warning: NOT UPDATED FOR IDRIS 2 YET

9.1 Dependent pattern matching

Since types can depend on values, the form of some arguments can be determined by the value of others. For example, if we were to write down the implicit length arguments to `(++)`, we'd see that the form of the length argument was determined by whether the vector was empty or not:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) {n=Z}   []        ys = ys
(++) {n=S k} (x :: xs) ys = x :: xs ++ ys
```

If `n` was a successor in the `[]` case, or zero in the `::` case, the definition would not be well typed.

9.2 The `with` rule — matching intermediate values

Very often, we need to match on the result of an intermediate computation. Idris provides a construct for this, the `with` rule, inspired by views in `Epigram`[1], which takes account of the fact that matching on a value in a dependently typed language can affect what we know about the forms of other values. In its simplest form, the `with` rule adds another argument to the function being defined.

We have already seen a vector filter function. This time, we define it using `with` as follows:

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
filter p [] = ( _ ** [] )
filter p (x :: xs) with (filter p xs)
  filter p (x :: xs) | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

Here, the `with` clause allows us to deconstruct the result of `filter p xs`. The view refined argument pattern `filter p (x :: xs)` goes beneath the `with` clause, followed by a vertical bar `|`, followed by the deconstructed intermediate result `( _ ** xs' )`. If the view refined argument pattern is unchanged from the original function argument pattern, then the left side of `|` is extraneous and may be omitted:

```
filter p (x :: xs) with (filter p xs)
  | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

`with` clauses can also be nested:

```
foo : Int -> Int -> Bool
foo n m with (n + 1)
  foo _ m | 2 with (m + 1)
    foo _ _ | 2 | 3 = True
    foo _ _ | 2 | _ = False
  foo _ _ | _ = False
```

If the intermediate computation itself has a dependent type, then the result can affect the forms of other arguments — we can learn the form of one value by testing another. In these cases, view refined argument patterns must be explicit. For example, a `Nat` is either even or odd. If it is even it will be the sum of two equal `Nat`. Otherwise, it is the sum of two equal `Nat` plus one:

[1] Conor McBride and James McKinna. 2004. The view from the left. J. Funct. Program. 14, 1 (January 2004), 69-111. https://doi.org/10.1017/S0956796803004829

```
data Parity : Nat -> Type where
   Even : {n : _} -> Parity (n + n)
   Odd  : {n : _} -> Parity (S (n + n))
```

We say `Parity` is a *view* of `Nat`. It has a *covering function* which tests whether it is even or odd and constructs the predicate accordingly. Note that we're going to need access to `n` at run time, so although it's an implicit argument, it has unrestricted multiplicity.

```
parity : (n:Nat) -> Parity n
```

We'll come back to the definition of `parity` shortly. We can use it to write a function which converts a natural number to a list of binary digits (least significant first) as follows, using the `with` rule:

```
natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | Even = False :: natToBin j
   natToBin (S (j + j)) | Odd  = True  :: natToBin j
```

The value of `parity k` affects the form of `k`, because the result of `parity k` depends on `k`. So, as well as the patterns for the result of the intermediate computation (`Even` and `Odd`) right of the `|`, we also write how the results affect the other patterns left of the `|`. That is:

- When `parity k` evaluates to `Even`, we can refine the original argument `k` to a refined pattern `(j + j)` according to `Parity (n + n)` from the `Even` constructor definition. So `(j + j)` replaces `k` on the left side of `|`, and the `Even` constructor appears on the right side. The natural number `j` in the refined pattern can be used on the right side of the `=` sign.
- Otherwise, when `parity k` evaluates to `Odd`, the original argument `k` is refined to `S (j + j)` according to `Parity (S (n + n))` from the `Odd` constructor definition, and `Odd` now appears on the right side of `|`, again with the natural number `j` used on the right side of the `=` sign.

Note that there is a function in the patterns (`+`) and repeated occurrences of `j` - this is allowed because another argument has determined the form of these patterns.

9.3 Defining `parity`

The definition of `parity` is a little tricky, and requires some knowledge of theorem proving (see Section *Theorem Proving* (page 50)), but for completeness, here it is:

```
parity : (n : Nat) -> Parity n
parity Z = Even {n = Z}
parity (S Z) = Odd {n = Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j))) | Even
      = rewrite plusSuccRightSucc j j in Even {n = S j}
  parity (S (S (S (j + j)))) | Odd
      = rewrite plusSuccRightSucc j j in Odd {n = S j}
```

For full details on `rewrite` in particular, please refer to the theorem proving tutorial, in Section proofs-index.

10 Theorem Proving

10.1 Equality

Idris allows propositional equalities to be declared, allowing theorems about programs to be stated and proved. An equality type is defined as follows in the Prelude:

```
data Equal : a -> b -> Type where
     Refl : Equal x x
```

As a notational convenience, `Equal x y` can be written as `x = y`. Equalities can be proposed between any values of any types, but the only way to construct a proof of equality is if values actually are equal. For example:

```
fiveIsFive : 5 = 5
fiveIsFive = Refl

twoPlusTwo : 2 + 2 = 4
twoPlusTwo = Refl
```

If we try...

```
twoPlusTwoBad : 2 + 2 = 5
twoPlusTwoBad = Refl
```

...then we'll get an error:

```
Proofs.idr:8:17--10:1:While processing right hand side of Main.twoPlusTwoBad at Proofs.idr:8:1-
↪-10:1:
When unifying 4 = 4 and (fromInteger 2 + fromInteger 2) = (fromInteger 5)
Mismatch between:
        4
and
        5
```

10.2 The Empty Type

There is an empty type, `Void`, which has no constructors. It is therefore impossible to construct a canonical element of the empty type. We can therefore use the empty type to prove that something is impossible, for example zero is never equal to a successor:

```
disjoint : (n : Nat) -> Z = S n -> Void
disjoint n prf = replace {p = disjointTy} prf ()
  where
    disjointTy : Nat -> Type
    disjointTy Z = ()
    disjointTy (S k) = Void
```

Don't worry if you don't get all the details of how this works just yet - essentially, it applies the library function `replace`, which uses an equality proof to transform a predicate. Here we use it to transform a value of a type which can exist, the empty tuple, to a value of a type which can't, by using a proof of something which can't exist.

Once we have an element of the empty type, we can prove anything. `void` is defined in the library, to assist with proofs by contradiction.

```
void : Void -> a
```

10.3 Proving Theorems

When type checking dependent types, the type itself gets *normalised*. So imagine we want to prove the following theorem about the reduction behaviour of `plus`:

```
plusReduces : (n:Nat) -> plus Z n = n
```

We've written down the statement of the theorem as a type, in just the same way as we would write the type of a program. In fact there is no real distinction between proofs and programs. A proof, as far as we are concerned here, is merely a program with a precise enough type to guarantee a particular property of interest.

We won't go into details here, but the Curry-Howard correspondence[1] explains this relationship. The proof itself is immediate, because `plus Z n` normalises to `n` by the definition of `plus`:

```
plusReduces n = Refl
```

It is slightly harder if we try the arguments the other way, because plus is defined by recursion on its first argument. The proof also works by recursion on the first argument to `plus`, namely `n`.

```
plusReducesZ : (n:Nat) -> n = plus n Z
plusReducesZ Z = Refl
plusReducesZ (S k) = cong S (plusReducesZ k)
```

`cong` is a function defined in the library which states that equality respects function application:

```
cong : (f : t -> u) -> a = b -> f a = f b
```

To see more detail on what's going on, we can replace the recursive call to `plusReducesZ` with a hole:

```
plusReducesZ (S k) = cong S ?help
```

Then inspecting the type of the hole at the REPL shows us:

```
Main> :t help
   k : Nat
--------------------------------------
help : k = (plus k Z)
```

We can do the same for the reduction behaviour of plus on successors:

```
plusReducesS : (n:Nat) -> (m:Nat) -> S (plus n m) = plus n (S m)
plusReducesS Z m = Refl
plusReducesS (S k) m = cong S (plusReducesS k m)
```

Even for small theorems like these, the proofs are a little tricky to construct in one go. When things get even slightly more complicated, it becomes too much to think about to construct proofs in this "batch mode".

Idris provides interactive editing capabilities, which can help with building proofs. For more details on building proofs interactively in an editor, see proofs-index.

[1] Timothy G. Griffin. 1989. A formulae-as-type notion of control. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '90). ACM, New York, NY, USA, 47-58. DOI=10.1145/96709.96714 https://doi.acm.org/10.1145/96709.96714

10.4 Theorems in Practice

The need to prove theorems can arise naturally in practice. For example, previously (*Views and the "with" rule* (page 49)) we implemented `natToBin` using a function `parity`:

```
parity : (n:Nat) -> Parity n
```

We provided a definition for `parity`, but without explanation. We might have hoped that it would look something like the following:

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = Even {n=S j}
  parity (S (S (S (j + j)))) | Odd  = Odd {n=S j}
```

Unfortunately, this fails with a type error:

```
With.idr:26:17--27:3:While processing right hand side of Main.with block in 2419 at With.
↪idr:24:3--27:3:
Can't solve constraint between:
        plus j (S j)
and
        S (plus j j)
```

The problem is that normalising `S j + S j`, in the type of `Even` doesn't result in what we need for the type of the right hand side of `Parity`. We know that `S (S (plus j j))` is going to be equal to `S j + S j`, but we need to explain it to Idris with a proof. We can begin by adding some *holes* (see *Totality and Covering* (page 9)) to the definition:

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = let result = Even {n=S j} in
                                      ?helpEven
  parity (S (S (S (j + j)))) | Odd  = let result = Odd {n=S j} in
                                      ?helpOdd
```

Checking the type of `helpEven` shows us what we need to prove for the `Even` case:

```
  j : Nat
  result : Parity (S (plus j (S j)))
--------------------------------------
helpEven : Parity (S (S (plus j j)))
```

We can therefore write a helper function to *rewrite* the type to the form we need:

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in p
```

The `rewrite ... in` syntax allows you to change the required type of an expression by rewriting it according to an equality proof. Here, we have used `plusSuccRightSucc`, which has the following type:

```
plusSuccRightSucc : (left : Nat) -> (right : Nat) -> S (left + right) = left + S right
```

We can see the effect of `rewrite` by replacing the right hand side of `helpEven` with a hole, and working step by step. Beginning with the following:

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = ?helpEven_rhs
```

We can look at the type of `helpEven_rhs`:

```
  j : Nat
  p : Parity (S (plus j (S j)))
--------------------------------------
helpEven_rhs : Parity (S (S (plus j j)))
```

Then we can `rewrite` by applying `plusSuccRightSucc j j`, which gives an equation `S (j + j) = j + S j`, thus replacing `S (j + j)` (or, in this case, `S (plus j j)` since `S (j + j)` reduces to that) in the type with `j + S j`:

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in ?helpEven_rhs
```

Checking the type of `helpEven_rhs` now shows what has happened, including the type of the equation we just used (as the type of `_rewrite_rule`):

```
Main> :t helpEven_rhs
   j : Nat
   p : Parity (S (plus j (S j)))
--------------------------------------
helpEven_rhs : Parity (S (plus j (S j)))
```

Using `rewrite` and another helper for the `Odd` case, we can complete `parity` as follows:

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in p

helpOdd : (j : Nat) -> Parity (S (S (j + S j))) -> Parity (S (S (S (j + j))))
helpOdd j p = rewrite plusSuccRightSucc j j in p

parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = helpEven j (Even {n = S j})
  parity (S (S (S (j + j)))) | Odd  = helpOdd j (Odd {n = S j})
```

Full details of `rewrite` are beyond the scope of this introductory tutorial, but it is covered in the theorem proving tutorial (see proofs-index).

10.5 Totality Checking

If we really want to trust our proofs, it is important that they are defined by *total* functions — that is, a function which is defined for all possible inputs and is guaranteed to terminate. Otherwise we could construct an element of the empty type, from which we could prove anything:

```
-- making use of 'hd' being partially defined
empty1 : Void
empty1 = hd [] where
    hd : List a -> a
    hd (x :: xs) = x

-- not terminating
```

(continues on next page)

(continued from previous page)

```
empty2 : Void
empty2 = empty2
```

Internally, Idris checks every definition for totality, and we can check at the prompt with the `:total` command. We see that neither of the above definitions is total:

```
Void> :total empty1
Void.empty1 is not covering due to call to function empty1:hd
Void> :total empty2
Void.empty2 is possibly not terminating due to recursive path Void.empty2
```

Note the use of the word "possibly" — a totality check can never be certain due to the undecidability of the halting problem. The check is, therefore, conservative. It is also possible (and indeed advisable, in the case of proofs) to mark functions as total so that it will be a compile time error for the totality check to fail:

```
total empty2 : Void
empty2 = empty2
```

Reassuringly, our proof in Section *The Empty Type* (page 51) that the zero and successor constructors are disjoint is total:

```
Main> :total disjoint
Main.disjoint is Total
```

The totality check is, necessarily, conservative. To be recorded as total, a function `f` must:

- Cover all possible inputs
- Be *well-founded* — i.e. by the time a sequence of (possibly mutually) recursive calls reaches `f` again, it must be possible to show that one of its arguments has decreased.
- Not use any data types which are not *strictly positive*
- Not call any non-total functions

Directives and Compiler Flags for Totality

> **Warning:** Not all of this is implemented yet for Idris 2

By default, Idris allows all well-typed definitions, whether total or not. However, it is desirable for functions to be total as far as possible, as this provides a guarantee that they provide a result for all possible inputs, in finite time. It is possible to make total functions a requirement, either:

- By using the `--total` compiler flag.
- By adding a `%default total` directive to a source file. All definitions after this will be required to be total, unless explicitly flagged as `partial`.

All functions *after* a `%default total` declaration are required to be total. Correspondingly, after a `%default partial` declaration, the requirement is relaxed.

Finally, the compiler flag `--warnpartial` causes to print a warning for any undeclared partial function.

Totality checking issues

Please note that the totality checker is not perfect! Firstly, it is necessarily conservative due to the undecidability of the halting problem, so many programs which *are* total will not be detected as such. Secondly, the current implementation has had limited effort put into it so far, so there may still be cases where it believes a function is total which is not. Do not rely on it for your proofs yet!

Hints for totality

In cases where you believe a program is total, but Idris does not agree, it is possible to give hints to the checker to give more detail for a termination argument. The checker works by ensuring that all chains of recursive calls eventually lead to one of the arguments decreasing towards a base case, but sometimes this is hard to spot. For example, the following definition cannot be checked as `total` because the checker cannot decide that `filter (< x) xs` will always be smaller than `(x :: xs)`:

```
qsort : Ord a => List a -> List a
qsort [] = []
qsort (x :: xs)
   = qsort (filter (< x) xs) ++
      (x :: qsort (filter (>= x) xs))
```

The function `assert_smaller`, defined in the prelude, is intended to address this problem:

```
assert_smaller : a -> a -> a
assert_smaller x y = y
```

It simply evaluates to its second argument, but also asserts to the totality checker that `y` is structurally smaller than `x`. This can be used to explain the reasoning for totality if the checker cannot work it out itself. The above example can now be written as:

```
total
qsort : Ord a => List a -> List a
qsort [] = []
qsort (x :: xs)
   = qsort (assert_smaller (x :: xs) (filter (< x) xs)) ++
      (x :: qsort (assert_smaller (x :: xs) (filter (>= x) xs)))
```

The expression `assert_smaller (x :: xs) (filter (<= x) xs)` asserts that the result of the filter will always be smaller than the pattern `(x :: xs)`.

In more extreme cases, the function `assert_total` marks a subexpression as always being total:

```
assert_total : a -> a
assert_total x = x
```

In general, this function should be avoided, but it can be very useful when reasoning about primitives or externally defined functions (for example from a C library) where totality can be shown by an external argument.

11 Interactive Editing

By now, we have seen several examples of how Idris' dependent type system can give extra confidence in a function's correctness by giving a more precise description of its intended behaviour in its *type*. We have also seen an example of how the type system can help with embedded DSL development by allowing a programmer to describe the type system of an object language. However, precise types give

us more than verification of programs — we can also use the type system to help write programs which are *correct by construction*, interactively.

The Idris REPL provides several commands for inspecting and modifying parts of programs, based on their types, such as case splitting on a pattern variable, inspecting the type of a hole, and even a basic proof search mechanism. In this section, we explain how these features can be exploited by a text editor, and specifically how to do so in Vim. An interactive mode for Emacs is also available (though not yet updated for Idris 2).

11.1 Editing at the REPL

Note: The Idris2 repl does not support readline in the interest of keeping dependencies minimal. Unfortunately this precludes some niceties such as line editing, persistent history and completion. A useful work around is to install rlwrap, this utility provides all the aforementioned features simply by invoking the Idris2 repl as an argument to the utility `rlwrap idris2`

The REPL provides a number of commands, which we will describe shortly, which generate new program fragments based on the currently loaded module. These take the general form:

```
:command [line number] [name]
```

That is, each command acts on a specific source line, at a specific name, and outputs a new program fragment. Each command has an alternative form, which *updates* the source file in-place:

```
:command! [line number] [name]
```

It is also possible to invoke Idris in a mode which runs a REPL command, displays the result, then exits, using `idris2 --client`. For example:

```
$ idris2 --client ':t plus'
Prelude.plus : Nat -> Nat -> Nat
$ idris2 --client '2+2'
4
```

A text editor can take advantage of this, along with the editing commands, in order to provide interactive editing support.

11.2 Editing Commands

:addclause

The `:addclause n f` command, abbreviated `:ac n f`, creates a template definition for the function named `f` declared on line `n`. For example, if the code beginning on line 94 contains:

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
```

then `:ac 94 vzipWith` will give:

```
vzipWith f xs ys = ?vzipWith_rhs
```

The names are chosen according to hints which may be given by a programmer, and then made unique by the machine by adding a digit if necessary. Hints can be given as follows:

```
%name Vect xs, ys, zs, ws
```

This declares that any names generated for types in the `Vect` family should be chosen in the order `xs`, `ys`, `zs`, `ws`.

:casesplit

The `:casesplit n x` command, abbreviated `:cs n x`, splits the pattern variable `x` on line `n` into the various pattern forms it may take, removing any cases which are impossible due to unification errors. For example, if the code beginning on line 94 is:

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f xs ys = ?vzipWith_rhs
```

then `:cs 96 xs` will give:

```
vzipWith f [] ys = ?vzipWith_rhs_1
vzipWith f (x :: xs) ys = ?vzipWith_rhs_2
```

That is, the pattern variable `xs` has been split into the two possible cases `[]` and `x :: xs`. Again, the names are chosen according to the same heuristic. If we update the file (using `:cs!`) then case split on `ys` on the same line, we get:

```
vzipWith f [] [] = ?vzipWith_rhs_3
```

That is, the pattern variable `ys` has been split into one case `[]`, Idris having noticed that the other possible case `y :: ys` would lead to a unification error.

:addmissing

The `:addmissing n f` command, abbreviated `:am n f`, adds the clauses which are required to make the function `f` on line `n` cover all inputs. For example, if the code beginning on line 94 is:

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f [] [] = ?vzipWith_rhs_1
```

then `:am 96 vzipWith` gives:

```
vzipWith f (x :: xs) (y :: ys) = ?vzipWith_rhs_2
```

That is, it notices that there are no cases for empty vectors, generates the required clauses, and eliminates the clauses which would lead to unification errors.

:proofsearch

The `:proofsearch n f` command, abbreviated `:ps n f`, attempts to find a value for the hole `f` on line `n` by proof search, trying values of local variables, recursive calls and constructors of the required family. Optionally, it can take a list of *hints*, which are functions it can try applying to solve the hole. For example, if the code beginning on line 94 is:

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f [] [] = ?vzipWith_rhs_1
vzipWith f (x :: xs) (y :: ys) = ?vzipWith_rhs_2
```

then `:ps 96 vzipWith_rhs_1` will give

```
[]
```

This works because it is searching for a `Vect` of length 0, of which the empty vector is the only possibility. Similarly, and perhaps surprisingly, there is only one possibility if we try to solve `:ps 97 vzipWith_rhs_2`:

```
f x y :: vzipWith f xs ys
```

This works because `vzipWith` has a precise enough type: The resulting vector has to be non-empty (a `::`); the first element must have type `c` and the only way to get this is to apply `f` to `x` and `y`; finally, the tail of the vector can only be built recursively.

:makewith

The `:makewith n f` command, abbreviated `:mw n f`, adds a `with` to a pattern clause. For example, recall `parity`. If line 10 is:

```
parity (S k) = ?parity_rhs
```

then `:mw 10 parity` will give:

```
parity (S k) with (_)
  parity (S k) | with_pat = ?parity_rhs
```

If we then fill in the placeholder `_` with `parity k` and case split on `with_pat` using `:cs 11 with_pat` we get the following patterns:

```
parity (S (plus n n)) | even = ?parity_rhs_1
parity (S (S (plus n n))) | odd = ?parity_rhs_2
```

Note that case splitting has normalised the patterns here (giving `plus` rather than `+`). In any case, we see that using interactive editing significantly simplifies the implementation of dependent pattern matching by showing a programmer exactly what the valid patterns are.

11.3 Interactive Editing in Vim

The editor mode for Vim provides syntax highlighting, indentation and interactive editing support using the commands described above. Interactive editing is achieved using the following editor commands, each of which update the buffer directly:

- **`\a` adds a template definition for the name declared on the** current line (using `:addclause`).
- **`\c` case splits the variable at the cursor (using** `:casesplit`).
- **`\m` adds the missing cases for the name at the cursor (using** `:addmissing`).
- `\w` adds a `with` clause (using `:makewith`).

- **`\s` invokes a proof search to solve the hole under the** cursor (using `:proofsearch`).

There are also commands to invoke the type checker and evaluator:

- **`\t` displays the type of the (globally visible) name under the** cursor. In the case of a hole, this displays the context and the expected type.
- `\e` prompts for an expression to evaluate.
- `\r` reloads and type checks the buffer.

Corresponding commands are also available in the Emacs mode. Support for other editors can be added in a relatively straightforward manner by using `idris2 --client`. More sophisticated support can be added by using the IDE protocol (yet to be documented for Idris 2, but which mostly extends to protocol documented for Idris 1.

12 Miscellany

In this section we discuss a variety of additional features:

- auto, implicit, and default arguments;
- literate programming; and
- the universe hierarchy.

12.1 Implicit arguments

We have already seen implicit arguments, which allows arguments to be omitted when they can be inferred by the type checker[1], e.g.

```
index : forall a, n . Fin n -> Vect n a -> a
```

Auto implicit arguments

In other situations, it may be possible to infer arguments not by type checking but by searching the context for an appropriate value, or constructing a proof. For example, the following definition of `head` which requires a proof that the list is non-empty:

```
isCons : List a -> Bool
isCons [] = False
isCons (x :: xs) = True

head : (xs : List a) -> (isCons xs = True) -> a
head (x :: xs) _ = x
```

If the list is statically known to be non-empty, either because its value is known or because a proof already exists in the context, the proof can be constructed automatically. Auto implicit arguments allow this to happen silently. We define `head` as follows:

```
head : (xs : List a) -> {auto p : isCons xs = True} -> a
head (x :: xs) = x
```

[1] https://github.com/david-christiansen/idris-type-providers

The `auto` annotation on the implicit argument means that Idris will attempt to fill in the implicit argument by searching for a value of the appropriate type. In fact, internally, this is exactly how interface resolution works. It will try the following, in order:

- Local variables, i.e. names bound in pattern matches or `let` bindings, with exactly the right type.
- The constructors of the required type. If they have arguments, it will search recursively up to a maximum depth of 100.
- Local variables with function types, searching recursively for the arguments.
- Any function with the appropriate return type which is marked with the `%hint` annotation.

In the case that a proof is not found, it can be provided explicitly as normal:

```
head xs {p = ?headProof}
```

Default implicit arguments

Besides having Idris automatically find a value of a given type, sometimes we want to have an implicit argument with a specific default value. In Idris, we can do this using the `default` annotation. While this is primarily intended to assist in automatically constructing a proof where auto fails, or finds an unhelpful value, it might be easier to first consider a simpler case, not involving proofs.

If we want to compute the n'th fibonacci number (and defining the 0th fibonacci number as 0), we could write:

```
fibonacci : {default 0 lag : Nat} -> {default 1 lead : Nat} -> (n : Nat) -> Nat
fibonacci {lag} Z = lag
fibonacci {lag} {lead} (S n) = fibonacci {lag=lead} {lead=lag+lead} n
```

After this definition, `fibonacci 5` is equivalent to `fibonacci {lag=0} {lead=1} 5`, and will return the 5th fibonacci number. Note that while this works, this is not the intended use of the `default` annotation. It is included here for illustrative purposes only. Usually, `default` is used to provide things like a custom proof search script.

12.2 Literate programming

Like Haskell, Idris supports *literate* programming. If a file has an extension of `.lidr` then it is assumed to be a literate file. In literate programs, everything is assumed to be a comment unless the line begins with a greater than sign `>`, for example:

```
> module literate

This is a comment. The main program is below

> main : IO ()
> main = putStrLn "Hello literate world!\n"
```

An additional restriction is that there must be a blank line between a program line (beginning with `>`) and a comment line (beginning with any other character).

12.3 Cumulativity

Warning: NOT YET IN IDRIS 2

Since values can appear in types and *vice versa*, it is natural that types themselves have types. For example:

```
*universe> :t Nat
Nat : Type
*universe> :t Vect
Vect : Nat -> Type -> Type
```

But what about the type of `Type`? If we ask Idris it reports:

```
*universe> :t Type
Type : Type 1
```

If `Type` were its own type, it would lead to an inconsistency due to Girard's paradox, so internally there is a *hierarchy* of types (or *universes*):

```
Type : Type 1 : Type 2 : Type 3 : ...
```

Universes are *cumulative*, that is, if `x : Type n` we can also have that `x : Type m`, as long as `n < m`. The typechecker generates such universe constraints and reports an error if any inconsistencies are found. Ordinarily, a programmer does not need to worry about this, but it does prevent (contrived) programs such as the following:

```
myid : (a : Type) -> a -> a
myid _ x = x

idid :  (a : Type) -> a -> a
idid = myid _ myid
```

The application of `myid` to itself leads to a cycle in the universe hierarchy — `myid`'s first argument is a `Type`, which cannot be at a lower level than required if it is applied to itself.

13 Further Reading

Further information about Idris programming, and programming with dependent types in general, can be obtained from various sources:

- Type-Driven Development with Idris by Edwin Brady, available from Manning.
- The Idris web site (https://www.idris-lang.org/) and by asking questions on the mailing list.
- The IRC channel `#idris`, on webchat.freenode.net.
- The wiki (https://github.com/idris-lang/Idris-dev/wiki/) has further user provided information, in particular:
  - https://github.com/idris-lang/Idris-dev/wiki/Manual
  - https://github.com/idris-lang/Idris-dev/wiki/Language-Features
- Examining the prelude and exploring the `samples` in the distribution. The Idris 2 source can be

found online at:

  - https://github.com/edwinb/Idris2.

- Existing projects on the `Idris Hackers` web space:

  - https://idris-hackers.github.io.

- Various papers (e.g.[1],[2], and[3]). Although these mostly describe older versions of Idris.

[1] Edwin Brady and Kevin Hammond. 2012. Resource-Safe systems programming with embedded domain specific languages. In Proceedings of the 14th international conference on Practical Aspects of Declarative Languages (PADL'12), Claudio Russo and Neng-Fa Zhou (Eds.). Springer-Verlag, Berlin, Heidelberg, 242-257. DOI=10.1007/978-3-642-27694-1_18 https://dx.doi.org/10.1007/978-3-642-27694-1_18

[2] Edwin C. Brady. 2011. IDRIS —: systems programming meets full dependent types. In Proceedings of the 5th ACM workshop on Programming languages meets program verification (PLPV '11). ACM, New York, NY, USA, 43-54. DOI=10.1145/1929529.1929536 https://doi.acm.org/10.1145/1929529.1929536

[3] Edwin C. Brady and Kevin Hammond. 2010. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10). ACM, New York, NY, USA, 297-308. DOI=10.1145/1863543.1863587 https://doi.acm.org/10.1145/1863543.1863587