The Idris Tutorial

Version 1.3.2

Contents

This is the Idris Tutorial. It provides a brief introduction to programming in the Idris Language. It covers the core language features, and assumes some familiarity with an existing functional programming language such as Haskell or OCaml.

---

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 `C` 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 systems 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 with against Idris. These files are available with the Idris 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 Prerequisites

Before installing Idris, you will need to make sure you have all of the necessary libraries and tools. You will need:

- A fairly recent version of GHC. The earliest version we currently test with is 7.10.3.
- The *GNU Multiple Precision Arithmetic Library* (GMP) is available from MacPorts/Homebrew and all major Linux distributions.

2.2 Downloading and Installing

The easiest way to install Idris, if you have all of the prerequisites, is to type:

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

```
cabal update; cabal install idris
```

This will install the latest version released on Hackage, along with any dependencies. If, however, you would like the most up to date development version you can find it, as well as build instructions, on GitHub at: https://github.com/idris-lang/Idris-dev.

If you haven't previously installed anything using Cabal, then Idris may not be on your path. Should the Idris executable not be found please ensure that you have added `~/.cabal/bin` to your `$PATH` environment variable. Mac OS X users may find they need to add `~/Library/Haskell/bin` instead, and Windows users will typically find that Cabal installs programs in `%HOME%\AppData\Roaming\cabal\bin`.

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 `idris hello.idr -o hello` at the shell prompt. This will create an executable called `hello`, which you can run:

```
$ idris hello.idr -o hello
$ ./hello
Hello world
```

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.

2.3 The Interactive Environment

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

```
$ idris
    ____    __     _
   /  _/___/ /____(_)____
   / // __  / ___/ / ___/     Version 1.3.2
 _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
/___/\__,_/_/  /_/____/       Type :? for help

Idris>
```

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 `hello`. Type checking a file, if successful,

creates a bytecode version of the file (in this case `hello.ibc`) to speed up loading in future. The bytecode is regenerated if the source file changes.

```
$ idris hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.3.2
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Type checking ./hello.idr
*hello> :t main
Main.main : IO ()
*hello> :c hello
*hello> :q
Bye bye
$ ./hello
Hello world
```

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 `idris Prims.idr`:

```
module Prims

x : Int
x = 42

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 30). 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, and the type of the answer. For example:

```
*prims> 6*6+6
42 : Integer
*prims> x == 6*6+6
True : Bool
```

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 == 6 * 6 + 6 then "The answer!" else "Not the answer"
"The answer!" : String
```

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

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:

```
Idris> plus (S (S Z)) (S (S Z))
4 : Nat
Idris> mult (S (S (S Z))) (plus (S (S Z)) (S (S Z)))
12 : Nat
```

Note: When displaying an element of `Nat` such as `(S (S (S (S Z))))`, Idris displays it as `4`. The result of `plus (S (S Z)) (S (S Z))` is actually `(S (S (S (S Z))))` which is the natural number `4`. This can be checked at the Idris prompt:

```
Idris> (S (S (S (S Z))))
4 : Nat
```

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 : Nat
Idris> mult 3 (plus 2 2)
12 : Nat
```

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 only in the type will be in scope in the `where` clause if it is a *parameter* to one of the types, i.e. it is fixed across the entire structure.

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
```

In general, functions defined in a `where` clause need a type declaration just like any top level function. However, the type declaration for a function `f` *can* be omitted if:

- `f` appears in the right hand side of the top level definition
- The type of `f` can be completely determined from its first application

So, for example, the following definitions are legal:

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

test : List Nat
test = [c (S 1), c Z, d (S Z)]
  where c x = 42 + x
        d y = c (y + 1 + z y)
              where z w = y + w
```

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`:

```
*Hello> :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`:

```
*Even> :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:

```
$ idris VBroken.idr --check
VBroken.idr:9:23-25:
When checking right hand side of Vect.++ with expected type
        Vect (S k + m) a

When checking an application of constructor Vect.::::
        Type mismatch between
                Vect (k + k) a (Type of xs ++ xs)
        and
                Vect (plus k m) a (Expected type)

        Specifically:
                Type mismatch between
                        plus k k
                and
                        plus k 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 : {a:Type} -> {n:Nat} -> Fin n -> Vect n a -> a
```

Implicit arguments, given in braces `{}` in the type 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.

Furthermore, `{}` can be used to pattern match on the left hand side, i.e. `{var = pat}` gets an implicit variable and attempts to pattern match on "pat"; For example:

```
isEmpty : Vect n a -> Bool
isEmpty {n = Z} _   = True
isEmpty {n = S k} _ = False
```

"using" notation

Sometimes it is useful to provide types of implicit arguments, particularly where there is a dependency ordering, or where the implicit arguments themselves have dependencies. For example, we may wish to state the types of the implicit arguments in the following definition, which defines a predicate on vectors (this is also defined in `Data.Vect`, under the name `Elem`):

```
data IsElem : a -> Vect n a -> Type where
   Here :  {x:a} ->   {xs:Vect n a} -> IsElem x (x :: xs)
   There : {x,y:a} -> {xs:Vect n a} -> IsElem x xs -> IsElem x (y :: xs)
```

An instance of `IsElem x xs` states that `x` is an element of `xs`. We can construct such a predicate if the required element is `Here`, at the head of the vector, or `There`, in the tail of the vector. For example:

```
testVec : Vect 4 Int
testVec = 3 :: 4 :: 5 :: 6 :: Nil

inVect : IsElem 5 Main.testVec
inVect = There (There Here)
```

Important: Implicit Arguments and Scope

Within the type signature the typechecker will treat all variables that start with an lowercase letter **and** are not applied to something else as an implicit variable. To get the above code example to compile you will need to provide a qualified name for `testVec`. In the example above, we have assumed that the code lives within the `Main` module.

If the same implicit arguments are being used a lot, it can make a definition difficult to read. To avoid this problem, a `using` block gives the types and ordering of any implicit arguments which can appear within the block:

```
using (x:a, y:a, xs:Vect n a)
  data IsElem : a -> Vect n a -> Type where
     Here  : IsElem x (x :: xs)
     There : IsElem x xs -> IsElem x (y :: xs)
```

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 46)). 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. Therefore in Idris, such interactions are encapsulated in the type `IO`:

```
data IO a -- 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 IO program:

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

The type of `putStrLn` explains that it takes a string, and returns an element of the unit type `()` via an I/O action. There is a variant `putStr` which outputs 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 defined in the prelude, for example 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.

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 (in fact, this is used to implement the `if...then...else` construct as we will see later). We would prefer if *only* the argument which was used was evaluated. To achieve this, Idris provides a `Lazy` data type, which allows evaluation to be suspended:

```
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 Codata Types

Codata types allow us to define infinite data structures by marking recursive arguments as potentially infinite. For a codata type `T`, each of its constructor arguments of type `T` are transformed into an argument of type `Inf T`. This makes each of the `T` arguments lazy, and allows infinite data structures of type `T` to be built. One example of a codata type is Stream, which is defined as follows.

```
codata Stream : Type -> Type where
  (::) : (e : a) -> Stream a -> Stream a
```

This gets translated into the following by the compiler.

```
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
```

It is important to note that codata does not allow the creation of infinite mutually recursive data structures. For example the following will create an infinite loop and cause a stack overflow.

```
mutual
  codata Blue a = B a (Red a)
  codata Red a = R a (Blue a)

mutual
  blue : Blue Nat
  blue = B 1 red

  red : Red Nat
  red = R 1 blue

mutual
  findB : (a -> Bool) -> Blue a -> a
  findB f (B x r) = if f x then x else findR f r

  findR : (a -> Bool) -> Red a -> a
  findR f (R x b) = if f x then x else findB f b

main : IO ()
main = do printLn $ findB (== 1) blue
```

To fix this we must add explicit `Inf` declarations to the constructor parameter types, since codata will not add it to constructor parameters of a **different** type from the one being defined. For example, the following outputs 1.

```
mutual
  data Blue : Type -> Type where
   B : a -> Inf (Red a) -> Blue a

  data Red : Type -> Type where
   R : a -> Inf (Blue a) -> Red a

mutual
  blue : Blue Nat
  blue = B 1 red

  red : Red Nat
  red = R 1 blue

mutual
  findB : (a -> Bool) -> Blue a -> a
  findB f (B x r) = if f x then x else findR f r

  findR : (a -> Bool) -> Red a -> a
  findR f (R x b) = if f x then x else findB f b

main : IO ()
main = do printLn $ findB (== 1) blue
```

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. The functions described here are imported automatically by every Idris program, as part of `Prelude.idr`.

`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
```

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 30)), 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` 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/prelude/Prelude/List.idr`
- `libs/base/Data/List.idr`
- `libs/base/Data/Vect.idr`
- `libs/base/Data/VectType.idr`

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

Aside: Anonymous functions and operator sections

There are actually 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` 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. `(a : A ** P)` is the type of a pair of A and P, where the name `a` can occur inside `P`. `( a ** 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 precisely equivalent:

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

The type checker could of course 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 easy:

```
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 `with` notation, which allows pattern matching on intermediate values:

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

We will see more on `with` notation later.

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

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 update

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

```
record { a->b->c } x
```

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
```

Note that it is no longer possible to use the `addStudent` function from earlier, since that would change the size of the class. A function to add a student must now specify in the type that the size of the class has been increased by one. As the size is specified using natural numbers, the new value can be incremented using the `S` constructor:

```
addStudent : Person -> SizedClass n -> SizedClass (S n)
addStudent p c =  SizedClassInfo (p :: students c) (className c)
```

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

Restrictions: The `case` construct is intended for simple analysis of intermediate expressions to avoid

the need to write auxiliary functions, and is also used internally to implement pattern matching `let` and lambda bindings. It will *only* work if:

- Each branch *matches* a value of the same type, and *returns* a value of the same type.
- The type of the result is "known". i.e. the type of the expression can be determined *without* type checking the `case`-expression itself.

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 9)). 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
```

```
Idris> 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' : Vect n a -> String
        show' Nil        = ""
        show' (x :: Nil) = show x
        show' (x :: xs)  = show x ++ ", " ++ show' xs
```

4.1 Default Definitions

The library 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)
```

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 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 (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 (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b
```

4.4 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.

```
*Interfaces> m_add (Just 20) (Just 22)
Just 42 : Maybe Int
*Interfaces> m_add (Just 20) Nothing
Nothing : Maybe Int
```

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:

```
readNumber : IO (Maybe Nat)
readNumber = do
```

(continues on next page)

(continued from previous page)

```
  input <- getLine
  if all isDigit (unpack input)
     then pure (Just (cast 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 = 42 in f !(g !(print y) !x)
```

is lifted to:

```
let y = 42 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 (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 ]
```

4.5 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].

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

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:

```
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
```

4.6 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, then the named implementation `myord` as follows, at the Idris prompt:

```
*named_impl> show (sort testList)
"[sO, ssO, ssssO]" : String
*named_impl> show (sort @{myord} testList)
"[ssssO, ssO, sO]" : String
```

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.

4.7 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 (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.

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
```

(continues on next page)

(continued from previous page)

```
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 modules. 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, or according to their type.

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 module's contents. By default, all names defined in a module are kept private. This aides 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's 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 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 might be 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.

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

%access Directive

The default export mode can be changed with the `%access` directive, for example:

```
module Btree

%access export

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

(continues on next page)

(continued from previous page)

```
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))

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

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

In this case, any function with no access modifier will be exported as `export`, rather than left `private`.

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`.

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
  test : Int -> Int
  test x = x * 2

namespace y
  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
```

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

  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:

```
*params> show (append _ _ (MkVects _ [1,2,3] [4,5,6]))
"[1, 2, 3, 4, 5, 6]" : String
```

6 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.

6.1 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.

6.2 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:

- `idris --build maths.ipkg` will build all modules in the package
- `idris --install maths.ipkg` will install the package, making it accessible by other Idris libraries and programs.
- `idris --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:

```
idris -p maths Main.idr
```

6.3 Testing Idris Packages

The integrated build system includes a simple testing framework. This framework collects functions listed in the `ipkg` file under `tests`. All test functions must return `IO ()`.

When you enter `idris --testpkg yourmodule.ipkg`, the build system creates a temporary file in a fresh environment on your machine by listing the `tests` functions under a single `main` function. It compiles this temporary file to an executable and then executes it.

The tests themselves are responsible for reporting their success or failure. Test functions commonly use `putStrLn` to report test results. The test framework does not impose any standards for reporting and consequently does not aggregate test results.

For example, lets take the following list of functions that are defined in a module called `NumOps` for a sample package `maths`:

```
module Maths.NumOps

%access export -- to make functions under test visible

double : Num a => a -> a
double a = a + a

triple : Num a => a -> a
triple a = a + double a
```

A simple test module, with a qualified name of `Test.NumOps` can be declared as:

```
module Test.NumOps

import Maths.NumOps

%access export  -- to make the test functions visible

assertEq : Eq a => (given : a) -> (expected : a) -> IO ()
assertEq g e = if g == e
    then putStrLn "Test Passed"
    else putStrLn "Test Failed"

assertNotEq : Eq a => (given : a) -> (expected : a) -> IO ()
assertNotEq g e = if not (g == e)
    then putStrLn "Test Passed"
    else putStrLn "Test Failed"

testDouble : IO ()
testDouble = assertEq (double 2) 4

testTriple : IO ()
testTriple = assertNotEq (triple 2) 5
```

The functions `assertEq` and `assertNotEq` are used to run expected passing, and failing, equality tests. The actual tests are `testDouble` and `testTriple`, and are declared in the `maths.ipkg` file as follows:

```
package maths

modules = Maths.NumOps
        , Test.NumOps

tests = Test.NumOps.testDouble
      , Test.NumOps.testTriple
```

The testing framework can then be invoked using `idris --testpkg maths.ipkg`:

```
> idris --testpkg maths.ipkg
Type checking ./Maths/NumOps.idr
Type checking ./Test/NumOps.idr
Type checking /var/folders/63/np5g0d5j54x1s0z12rf41wxm0000gp/T/idristests144128232716531729.idr
Test Passed
Test Passed
```

Note how both tests have reported success by printing `Test Passed` as we arranged for with the `assertEq` and `assertNoEq` functions.

6.4 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

pkgs = pruviloj, lightyear
```

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

6.5 More information

More details, including a complete listing of available fields, can be found in the reference manual in ref-sect-packages.

7 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.

7.1 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, and as it will be used regularly it will be represented as an implicit argument. To do so we define everything in a `using` block (keep in mind that everything after this point needs to be indented so as to be inside the `using` block):

```
using (G:Vect n Ty)
```

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 :: G) t
    Pop  : HasType k G t -> HasType (FS k) (u :: G) t

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

(continues on next page)

(continued from previous page)

```
    Var : HasType i G t -> Expr G t
    Val : (x : Integer) -> Expr G TyInt
    Lam : Expr (a :: G) t -> Expr G (TyFun a t)
    App : Expr G (TyFun a t) -> Expr G a -> Expr G t
    Op  : (interpTy a -> interpTy b -> interpTy c) ->
          Expr G a -> Expr G b -> Expr G c
    If  : Expr G TyBool ->
          Lazy (Expr G a) ->
          Lazy (Expr G a) -> Expr G a
```

The code above makes use of the `Vect` and `Fin` types from the Idris standard library. We import them because they are not provided in the prelude:

```
import Data.Vect
import Data.Fin
```

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 G T`, which is a proof that variable `i` in context `G` has type `T`. This is defined as follows:

```
data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: G) t
    Pop  : HasType k G t -> HasType (FS k) (u :: G) 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 G t -> Expr G 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 G 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 :: G) t -> Expr G (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 G (TyFun a t) -> Expr G a -> Expr G 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 G a -> Expr G b -> Expr G 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 G TyBool ->
     Lazy (Expr G a) ->
     Lazy (Expr G a) ->
     Expr G a
```

7.2 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 G -> Env (a :: G)

lookup : HasType i G t -> Env G -> 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 G -> Expr G 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
```

7.3 Testing

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

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

7.4 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:

```
$ idris interp.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.3.2
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Type checking ./interp.idr
*interp> :exec
Enter a number: 6
720
*interp>
```

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.

8 Views and the "`with`" rule

8.1 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.

8.2 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 (succ n)
```

(continues on next page)

[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

(continued from previous page)

```
  foo _ m | 2 with (succ m)
    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:

```
data Parity : Nat -> Type where
   Even : Parity (n + n)
   Odd  : 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.

```
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.

We will return to this function in the next section *Theorems in Practice* (page 44) to complete the definition of `parity`.

8.3 With and proofs

To use a dependent pattern match for theorem proving, it is sometimes necessary to explicitly construct the proof resulting from the pattern match. To do this, you can postfix the with clause with `proof p` and the proof generated by the pattern match will be in scope and named `p`. For example:

```
data Foo = FInt Int | FBool Bool

optional : Foo -> Maybe Int
```

(continues on next page)

(continued from previous page)

```
optional (FInt x) = Just x
optional (FBool b) = Nothing

isFInt : (foo:Foo) -> Maybe (x : Int ** (optional foo = Just x))
isFInt foo with (optional foo) proof p
  isFInt foo | Nothing = Nothing           -- here, p : Nothing = optional foo
  isFInt foo | (Just x) = Just (x ** Refl) -- here, p : Just x = optional foo
```

9 Theorem Proving

9.1 Equality

Idris allows propositional equalities to be declared, allowing theorems about programs to be stated and proved. Equality is built in, but conceptually has the following definition:

```
data (=) : a -> b -> Type where
   Refl : x = x
```

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
```

9.2 The Empty Type

There is an empty type, $\bot$, which has no constructors. It is therefore impossible to construct an element of the empty type, at least without using a partially defined or general recursive function (see Section *Totality Checking* (page 46) for more details). 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 p = replace {P = disjointTy} p ()
  where
    disjointTy : Nat -> Type
    disjointTy Z = ()
    disjointTy (S k) = Void
```

There is no need to worry too much about how this function works — 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
```

9.3 Simple 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 trivial, 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 (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
```

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 (plusReducesS k m)
```

Even for trivial 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.

9.4 Theorems in Practice

The need to prove theorems can arise naturally in practice. For example, previously (*Views and the "with" rule* (page 41)) we implemented `natToBin` using a function `parity`:

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

However, we didn't provide a definition for `parity`. We might expect it to look something like the following:

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
```

(continues on next page)

[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 http://doi.acm.org/10.1145/96709.96714

(continued from previous page)

```
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:

```
When checking right hand side of with block in views.parity with expected type
        Parity (S (S (j + j)))

Type mismatch between
        Parity (S j + S j) (Type of Even)
and
        Parity (S (S (plus j j))) (Expected type)
```

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 *Holes* (page 8)) 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`):

```
  j : Nat
  p : Parity (S (plus j (S j)))
  _rewrite_rule : S (plus j j) = 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).

9.5 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
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:

```
*Theorems> :total empty1
possibly not total due to: empty1#hd
    not total as there are missing cases
*Theorems> :total empty2
possibly not total due to recursive path empty2
```

Note the use of the word "possibly" — a totality check can, of course, 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


Type checking ./theorems.idr
theorems.idr:25:empty2 is possibly not total due to recursive path empty2
```

Reassuringly, our proof in Section *The Empty Type* (page 43) that the zero and successor constructors are disjoint is total:

```
*theorems> :total disjoint
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

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.

10 Provisional Definitions

Sometimes when programming with dependent types, the type required by the type checker and the type of the program we have written will be different (in that they do not have the same normal form), but nevertheless provably equal. For example, recall the `parity` function:

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

We'd like to implement this as follows:

```
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}
```

This simply states that zero is even, one is odd, and recursively, the parity of `k+2` is the same as the parity of `k`. Explicitly marking the value of `n` is even and odd is necessary to help type inference. Unfortunately, the type checker rejects this:

```
viewsbroken.idr:12:10:When elaborating right hand side of ViewsBroken.parity:
Type mismatch between
    Parity (plus (S j) (S j))
and
    Parity (S (S (plus j j)))

Specifically:
    Type mismatch between
        plus (S j) (S j)
    and
        S (S (plus j j))
```

The type checker is telling us that `(j+1)+(j+1)` and `2+j+j` do not normalise to the same value. This is because `plus` is defined by recursion on its first argument, and in the second value, there is a successor symbol on the second argument, so this will not help with reduction. These values are obviously equal — how can we rewrite the program to fix this problem?

10.1 Provisional definitions

Provisional definitions help with this problem by allowing us to defer the proof details until a later point. There are two main reasons why they are useful.

- When *prototyping*, it is useful to be able to test programs before finishing all the details of proofs.
- When *reading* a program, it is often much clearer to defer the proof details so that they do not distract the reader from the underlying algorithm.

Provisional definitions are written in the same way as ordinary definitions, except that they introduce the right hand side with a `?=` rather than `=`. We define `parity` as follows:

```
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}
```

When written in this form, instead of reporting a type error, Idris will insert a hole standing for a theorem which will correct the type error. Idris tells us we have two proof obligations, with names generated from the module and function names:

```
*views> :m
Global holes:
        [views.parity_lemma_2,views.parity_lemma_1]
```

The first of these has the following type:

```
*views> :p views.parity_lemma_1

---------------------------------- (views.parity_lemma_1) --------
{hole0} : (j : Nat) -> (Parity (plus (S j) (S j))) -> Parity (S (S (plus j j)))

-views.parity_lemma_1>
```

The two arguments are `j`, the variable in scope from the pattern match, and `value`, which is the value

we gave in the right hand side of the provisional definition. Our goal is to rewrite the type so that we can use this value. We can achieve this using the following theorem from the prelude:

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

We need to use `compute` again to unfold the definition of `plus`:

```
-views.parity_lemma_1> compute


--------------------------------- (views.parity_lemma_1) --------
{hole0} : (j : Nat) -> (Parity (S (plus j (S j)))) -> Parity (S (S (plus j j)))
```

After applying `intros` we have:

```
-views.parity_lemma_1> intros

  j : Nat
  value : Parity (S (plus j (S j)))
--------------------------------- (views.parity_lemma_1) --------
{hole2} : Parity (S (S (plus j j)))
```

Then we apply the `plusSuccRightSucc` rewrite rule, symmetrically, to `j` and `j`, giving:

```
-views.parity_lemma_1> rewrite sym (plusSuccRightSucc j j)

  j : Nat
  value : Parity (S (plus j (S j)))
--------------------------------- (views.parity_lemma_1) --------
{hole3} : Parity (S (plus j (S j)))
```

`sym` is a function, defined in the library, which reverses the order of the rewrite:

```
sym : l = r -> r = l
sym Refl = Refl
```

We can complete this proof using the `trivial` tactic, which finds `value` in the premises. The proof of the second lemma proceeds in exactly the same way.

We can now test the `natToBin` function from Section *The with rule — matching intermediate values* (page 41) at the prompt. The number 42 is 101010 in binary. The binary digits are reversed:

```
*views> show (natToBin 42)
"[False, True, False, True, False, True]" : String
```

10.2 Suspension of Disbelief

Idris requires that proofs be complete before compiling programs (although evaluation at the prompt is possible without proof details). Sometimes, especially when prototyping, it is easier not to have to do this. It might even be beneficial to test programs before attempting to prove things about them — if testing finds an error, you know you had better not waste your time proving something!

Therefore, Idris provides a built-in coercion function, which allows you to use a value of the incorrect types:

```
believe_me : a -> b
```

Obviously, this should be used with extreme caution. It is useful when prototyping, and can also be appropriate when asserting properties of external code (perhaps in an external C library). The "proof" of `views.parity_lemma_1` using this is:

```
views.parity_lemma_2 = proof {
    intro;
    intro;
    exact believe_me value;
}
```

The `exact` tactic allows us to provide an exact value for the proof. In this case, we assert that the value we gave was correct.

10.3 Example: Binary numbers

Previously, we implemented conversion to binary numbers using the `Parity` view. Here, we show how to use the same view to implement a verified conversion to binary. We begin by indexing binary numbers over their `Nat` equivalent. This is a common pattern, linking a representation (in this case `Binary`) with a meaning (in this case `Nat`):

```
data Binary : Nat -> Type where
   BEnd : Binary Z
   B0 : Binary n -> Binary (n + n)
   BI : Binary n -> Binary (S (n + n))
```

`B0` and `BI` take a binary number as an argument and effectively shift it one bit left, adding either a zero or one as the new least significant bit. The index, `n + n` or `S (n + n)` states the result that this left shift then add will have to the meaning of the number. This will result in a representation with the least significant bit at the front.

Now a function which converts a Nat to binary will state, in the type, that the resulting binary number is a faithful representation of the original Nat:

```
natToBin : (n:Nat) -> Binary n
```

The `Parity` view makes the definition fairly simple — halving the number is effectively a right shift after all — although we need to use a provisional definition in the Odd case:

```
natToBin : (n:Nat) -> Binary n
natToBin Z = BEnd
natToBin (S k) with (parity k)
   natToBin (S (j + j))     | Even  = BI (natToBin j)
   natToBin (S (S (j + j))) | Odd  ?= B0 (natToBin (S j))
```

The problem with the Odd case is the same as in the definition of `parity`, and the proof proceeds in the same way:

```
natToBin_lemma_1 = proof {
    intro;
    intro;
    rewrite sym (plusSuccRightSucc j j);
    trivial;
}
```

To finish, we'll implement a main program which reads an integer from the user and outputs it in binary.

```
main : IO ()
main = do putStr "Enter a number: "
```

(continues on next page)

(continued from previous page)

```
    x <- getLine
    print (natToBin (fromInteger (cast x)))
```

For this to work, of course, we need a `Show` implementation for `Binary n`:

```
Show (Binary n) where
    show (B0 x) = show x ++ "0"
    show (B1 x) = show x ++ "1"
    show BEnd = ""
```

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 EDSL 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 exploit types to help write programs which are *correct by construction*.

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.

11.1 Editing at the REPL

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]
```

When the REPL is loaded, it also starts a background process which accepts and responds to REPL commands, using `idris --client`. For example, if we have a REPL running elsewhere, we can execute commands such as:

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

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:

- `\d` **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`).
- `\o` **invokes a proof search to solve the hole under the** cursor (using `:proofsearch`).
- `\p` **invokes a proof search with additional hints 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 `idris -client`.

12 Syntax Extensions

Idris supports the implementation of *Embedded Domain Specific Languages* (EDSLs) in several ways[1]. One way, as we have already seen, is through extending `do` notation. Another important way is to allow extension of the core syntax. In this section we describe two ways of extending the syntax: `syntax` rules and `dsl` notation.

12.1 `syntax` rules

We have seen `if...then...else` expressions, but these are not built in. Instead, we can define a function in the prelude as follows (we have already seen this function in Section *Laziness* (page 14)):

```
ifThenElse : (x:Bool) -> Lazy a -> Lazy a -> a;
ifThenElse True  t e = t;
ifThenElse False t e = e;
```

and then extend the core syntax with a `syntax` declaration:

```
syntax if [test] then [t] else [e] = ifThenElse test t e;
```

[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 http://dx.doi.org/10.1007/978-3-642-27694-1_18

The left hand side of a `syntax` declaration describes the syntax rule, and the right hand side describes its expansion. The syntax rule itself consists of:

- **Keywords** — here, `if`, `then` and `else`, which must be valid identifiers.
- **Non-terminals** — included in square brackets, `[test]`, `[t]` and `[e]` here, which stand for arbitrary expressions. To avoid parsing ambiguities, these expressions cannot use syntax extensions at the top level (though they can be used in parentheses).
- **Names** — included in braces, which stand for names which may be bound on the right hand side.
- **Symbols** — included in quotations marks, e.g. `":="`. This can also be used to include reserved words in syntax rules, such as `"let"` or `"in"`.

The limitations on the form of a syntax rule are that it must include at least one symbol or keyword, and there must be no repeated variables standing for non-terminals. Any expression can be used, but if there are two non-terminals in a row in a rule, only simple expressions may be used (that is, variables, constants, or bracketed expressions). Rules can use previously defined rules, but may not be recursive. The following syntax extensions would therefore be valid:

```
syntax [var] ":=" [val]                = Assign var val;
syntax [test] "?" [t] ":" [e]          = if test then t else e;
syntax select [x] from [t] "where" [w] = SelectWhere x t w;
syntax select [x] from [t]             = Select x t;
```

Syntax macros can be further restricted to apply only in patterns (i.e. only on the left hand side of a pattern match clause) or only in terms (i.e. everywhere but the left hand side of a pattern match clause) by being marked as `pattern` or `term` syntax rules. For example, we might define an interval as follows, with a static check that the lower bound is below the upper bound using `so`:

```
data Interval : Type where
   MkInterval : (lower : Double) -> (upper : Double) ->
                So (lower < upper) -> Interval
```

We can define a syntax which, in patterns, always matches `Oh` for the proof argument, and in terms requires a proof term to be provided:

```
pattern syntax "[" [x] "..." [y] "]" = MkInterval x y Oh
term    syntax "[" [x] "..." [y] "]" = MkInterval x y ?bounds_lemma
```

In terms, the syntax `[x...y]` will generate a proof obligation `bounds_lemma` (possibly renamed).

Finally, syntax rules may be used to introduce alternative binding forms. For example, a `for` loop binds a variable on each iteration:

```
syntax for {x} "in" [xs] ":" [body] = forLoop xs (\x => body)

main : IO ()
main = do for x in [1..10]:
              putStrLn ("Number " ++ show x)
          putStrLn "Done!"
```

Note that we have used the `{x}` form to state that `x` represents a bound variable, substituted on the right hand side. We have also put `in` in quotation marks since it is already a reserved word.

12.2 `dsl` notation

The well-typed interpreter in Section *Example: The Well-Typed Interpreter* (page 37) is a simple example of a common programming pattern with dependent types. Namely: describe an *object language* and its

type system with dependent types to guarantee that only well-typed programs can be represented, then program using that representation. Using this approach we can, for example, write programs for serialising binary data[2] or running concurrent processes safely[3].

Unfortunately, the form of object language programs makes it rather hard to program this way in practice. Recall the factorial program in `Expr` for example:

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

Since this is a particularly useful pattern, Idris provides syntax overloading[1] to make it easier to program in such object languages:

```
mkLam : TTName -> Expr (t::g) t' -> Expr g (TyFun t t')
mkLam _ body = Lam body

dsl expr
    variable    = Var
    index_first = Stop
    index_next  = Pop
    lambda      = mkLam
```

A `dsl` block describes how each syntactic construct is represented in an object language. Here, in the `expr` language, any variable is translated to the `Var` constructor, using `Pop` and `Stop` to construct the de Bruijn index (i.e., to count how many bindings since the variable itself was bound); and any lambda is translated to a `Lam` constructor. The `mkLam` function simply ignores its first argument, which is the name that the user chose for the variable. It is also possible to overload `let` and dependent function syntax (`pi`) in this way. We can now write `fact` as follows:

```
fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => If (Op (==) x (Val 0))
                      (Val 1) (Op (*) (app fact (Op (-) x (Val 1))) x))
```

In this new version, `expr` declares that the next expression will be overloaded. We can take this further, using idiom brackets, by declaring:

```
(<*>) : (f : Lazy (Expr G (TyFun a t))) -> Expr G a -> Expr G t
(<*>) f a = App f a

pure : Expr G a -> Expr G a
pure = id
```

Note that there is no need for these to be part of an implementation of `Applicative`, since idiom bracket notation translates directly to the names `<*>` and `pure`, and ad-hoc type-directed overloading is allowed. We can now say:

```
fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => If (Op (==) x (Val 0))
                      (Val 1) (Op (*) [| fact (Op (-) x (Val 1)) |] x))
```

With some more ad-hoc overloading and use of interfaces, and a new syntax rule, we can even go as far as:

[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 http://doi.acm.org/10.1145/1929529.1929536

[3] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. Fundam. Inf. 102, 2 (April 2010), 145-176. http://dl.acm.org/citation.cfm?id=1883636

```
syntax "IF" [x] "THEN" [t] "ELSE" [e] = If x t e

fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => IF x == 0 THEN 1 ELSE [| fact (x - 1) |] * x)
```

13 Miscellany

In this section we discuss a variety of additional features:

- auto, implicit, and default arguments;
- literate programming;
- interfacing with external libraries through the foreign function;
- interface;
- type providers;
- code generation; and
- the universe hierarchy.

13.1 Implicit arguments

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

```
index : {a:Type} -> {n:Nat} -> 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
```

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. 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.

13.2 Implicit conversions

Idris supports the creation of *implicit conversions*, which allow automatic conversion of values from one type to another when required to make a term type correct. This is intended to increase convenience and reduce verbosity. A contrived but simple example is the following:

```
implicit intString : Int -> String
intString = show

test : Int -> String
test x = "Number " ++ x
```

In general, we cannot append an `Int` to a `String`, but the implicit conversion function `intString` can convert `x` to a `String`, so the definition of `test` is type correct. An implicit conversion is implemented just like any other function, but given the `implicit` modifier, and restricted to one explicit argument.

Only one implicit conversion will be applied at a time. That is, implicit conversions cannot be chained. Implicit conversions of simple types, as above, are however discouraged! More commonly, an implicit conversion would be used to reduce verbosity in an embedded domain specific language, or to hide details of a proof. Such examples are beyond the scope of this tutorial.

13.3 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).

13.4 Foreign function calls

For practical programming, it is often necessary to be able to use external libraries, particularly for interfacing with the operating system, file system, networking, *et cetera*. Idris provides a lightweight foreign function interface for achieving this, as part of the prelude. For this, we assume a certain amount of knowledge of C and the `gcc` compiler. First, we define a datatype which describes the external types we can handle:

```
data FTy = FInt | FFloat | FChar | FString | FPtr | FUnit
```

Each of these corresponds directly to a C type. Respectively: `int`, `double`, `char`, `char*`, `void*` and `void`. There is also a translation to a concrete Idris type, described by the following function:

```
interpFTy : FTy -> Type
interpFTy FInt    = Int
interpFTy FFloat  = Double
interpFTy FChar   = Char
interpFTy FString = String
interpFTy FPtr    = Ptr
interpFTy FUnit   = ()
```

A foreign function is described by a list of input types and a return type, which can then be converted to an Idris type:

```
ForeignTy : (xs:List FTy) -> (t:FTy) -> Type
```

A foreign function is assumed to be impure, so `ForeignTy` builds an `IO` type, for example:

```
Idris> ForeignTy [FInt, FString] FString
Int -> String -> IO String : Type

Idris> ForeignTy [FInt, FString] FUnit
Int -> String -> IO () : Type
```

We build a call to a foreign function by giving the name of the function, a list of argument types and the return type. The built in construct `mkForeign` converts this description to a function callable by Idris:

```
data Foreign : Type -> Type where
    FFun : String -> (xs:List FTy) -> (t:FTy) ->
           Foreign (ForeignTy xs t)

mkForeign : Foreign x -> x
```

Note that the compiler expects `mkForeign` to be fully applied to build a complete foreign function call. For example, the `putStr` function is implemented as follows, as a call to an external function `putStr` defined in the run-time system:

```
putStr : String -> IO ()
putStr x = mkForeign (FFun "putStr" [FString] FUnit) x
```

Include and linker directives

Foreign function calls are translated directly to calls to C functions, with appropriate conversion between the Idris representation of a value and the C representation. Often this will require extra libraries to be linked in, or extra header and object files. This is made possible through the following directives:

- `%lib target x` — include the `libx` library. If the target is `C` this is equivalent to passing the `-lx` option to `gcc`. If the target is Java the library will be interpreted as a `groupId:artifactId:packaging:version` dependency coordinate for maven.
- `%include target x` — use the header file or import `x` for the given back end target.
- `%link target x.o` — link with the object file `x.o` when using the given back end target.
- `%dynamic x.so` — dynamically link the interpreter with the shared object `x.so`.

Testing foreign function calls

Normally, the Idris interpreter (used for typechecking and at the REPL) will not perform IO actions. Additionally, as it neither generates C code nor compiles to machine code, the `%lib`, `%include` and `%link` directives have no effect. IO actions and FFI calls can be tested using the special REPL command `:x EXPR`, and C libraries can be dynamically loaded in the interpreter by using the `:dynamic` command or the `%dynamic` directive. For example:

```
Idris> :dynamic libm.so
Idris> :x unsafePerformIO ((mkForeign (FFun "sin" [FFloat] FFloat)) 1.6)
0.9995736030415051 : Double
```

13.5 Type Providers

Idris type providers, inspired by F#'s type providers, are a means of making our types be "about" something in the world outside of Idris. For example, given a type that represents a database schema and a query that is checked against it, a type provider could read the schema of a real database during type checking.

Idris type providers use the ordinary execution semantics of Idris to run an IO action and extract the result. This result is then saved as a constant in the compiled code. It can be a type, in which case it is used like any other type, or it can be a value, in which case it can be used as any other value, including as an index in types.

Type providers are still an experimental extension. To enable the extension, use the `%language` directive:

```
%language TypeProviders
```

A provider `p` for some type `t` is simply an expression of type `IO (Provider t)`. The `%provide` directive causes the type checker to execute the action and bind the result to a name. This is perhaps best illustrated with a simple example. The type provider `fromFile` reads a text file. If the file consists of the string `Int`, then the type `Int` will be provided. Otherwise, it will provide the type `Nat`.

```
strToType : String -> Type
strToType "Int" = Int
strToType _ = Nat

fromFile : String -> IO (Provider Type)
fromFile fname = do Right str <- readFile fname
                      | Left err => pure (Provide Void)
                    pure (Provide (strToType (trim str)))
```

We then use the `%provide` directive:

```
%provide (T1 : Type) with fromFile "theType"

foo : T1
foo = 2
```

If the file named `theType` consists of the word `Int`, then `foo` will be an `Int`. Otherwise, it will be a `Nat`. When Idris encounters the directive, it first checks that the provider expression `fromFile theType` has type `IO (Provider Type)`. Next, it executes the provider. If the result is `Provide t`, then `T1` is defined as `t`. Otherwise, the result is an error.

Our datatype `Provider t` has the following definition:

```
data Provider a = Error String
                | Provide a
```

We have already seen the `Provide` constructor. The `Error` constructor allows type providers to return useful error messages. The example in this section was purposefully simple. More complex type provider implementations, including a statically-checked SQLite binding, are available in an external collection[1].

13.6 C Target

The default target of Idris is C. Compiling via:

```
$ idris hello.idr -o hello
```

is equivalent to:

```
$ idris --codegen C hello.idr -o hello
```

When the command above is used, a temporary C source is generated, which is then compiled into an executable named `hello`.

In order to view the generated C code, compile via:

```
$ idris hello.idr -S -o hello.c
```

To turn optimisations on, use the `%flag C` pragma within the code, as is shown below:

```
module Main
%flag C "-O3"

factorial : Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

(continues on next page)

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

(continued from previous page)

```
main : IO ()
main = do
     putStrLn $ show $ factorial 3
```

To compile the generated C with debugging information e.g. to use `gdb` to debug segmentation faults in Idris programs, use the `%flag C` pragma to include debugging symbols, as is shown below:

```
%flag C "-g"
```

13.7 JavaScript Target

Idris is capable of producing *JavaScript* code that can be run in a browser as well as in the *NodeJS* environment or alike. One can use the FFI to communicate with the *JavaScript* ecosystem.

Code Generation

Code generation is split into two separate targets. To generate code that is tailored for running in the browser issue the following command:

```
$ idris --codegen javascript hello.idr -o hello.js
```

The resulting file can be embedded into your HTML just like any other *JavaScript* code.

Generating code for *NodeJS* is slightly different. Idris outputs a *JavaScript* file that can be directly executed via `node`.

```
$ idris --codegen node hello.idr -o hello
$ ./hello
Hello world
```

Take into consideration that the *JavaScript* code generator is using `console.log` to write text to `stdout`, this means that it will automatically add a newline to the end of each string. This behaviour does not show up in the *NodeJS* code generator.

Using the FFI

To write a useful application we need to communicate with the outside world. Maybe we want to manipulate the DOM or send an Ajax request. For this task we can use the FFI. Since most *JavaScript* APIs demand callbacks we need to extend the FFI so we can pass functions as arguments.

The *JavaScript* FFI works a little bit differently than the regular FFI. It uses positional arguments to directly insert our arguments into a piece of *JavaScript* code.

One could use the primitive addition of *JavaScript* like so:

```
module Main

primPlus : Int -> Int -> IO Int
primPlus a b = mkForeign (FFun "%0 + %1" [FInt, FInt] FInt) a b

main : IO ()
main = do
```

(continues on next page)

(continued from previous page)

```
  a <- primPlus 1 1
  b <- primPlus 1 2
  print (a, b)
```

Notice that the `%n` notation qualifies the position of the `n`-th argument given to our foreign function starting from 0. When you need a percent sign rather than a position simply use `%%` instead.

Passing functions to a foreign function is very similar. Let's assume that we want to call the following function from the *JavaScript* world:

```
function twice(f, x) {
  return f(f(x));
}
```

We obviously need to pass a function `f` here (we can infer it from the way we use `f` in `twice`, it would be more obvious if *JavaScript* had types).

The *JavaScript* FFI is able to understand functions as arguments when you give it something of type `FFunction`. The following example code calls `twice` in *JavaScript* and returns the result to our Idris program:

```
module Main

twice : (Int -> Int) -> Int -> IO Int
twice f x = mkForeign (
  FFun "twice(%0,%1)" [FFunction FInt FInt, FInt] FInt
) f x

main : IO ()
main = do
  a <- twice (+1) 1
  print a
```

The program outputs `3`, just like we expected.

Including external *JavaScript* files

Whenever one is working with *JavaScript* one might want to include external libraries or just some functions that she or he wants to call via FFI which are stored in external files. The *JavaScript* and *NodeJS* code generators understand the `%include` directive. Keep in mind that *JavaScript* and *NodeJS* are handled as different code generators, therefore you will have to state which one you want to target. This means that you can include different files for *JavaScript* and *NodeJS* in the same Idris source file.

So whenever you want to add an external *JavaScript* file you can do this like so:

For *NodeJS*:

```
%include Node "path/to/external.js"
```

And for use in the browser:

```
%include JavaScript "path/to/external.js"
```

The given files will be added to the top of the generated code. For library packages you can also use the ipkg objs option to include the js file in the installation, and use:

```
%include Node "package/external.js"
```

The *JavaScript* and *NodeJS* backends of Idris will also lookup for the file on that location.

Including *NodeJS* modules

The *NodeJS* code generator can also include modules with the `%lib` directive.

```
%lib Node "fs"
```

This directive compiles into the following *JavaScript*

```
var fs = require("fs");
```

Shrinking down generated *JavaScript*

Idris can produce very big chunks of *JavaScript* code. However, the generated code can be minified using the `closure-compiler` from Google. Any other minifier is also suitable but `closure-compiler` offers advanced compilation that does some aggressive inlining and code elimination. Idris can take full advantage of this compilation mode and it's highly recommended to use it when shipping a *JavaScript* application written in Idris.

13.8 Cumulativity

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.

14 Further Reading

Further information about Idris programming, and programming with dependent types in general, can be obtained from various sources:

- The Idris web site (http://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 source can be found online at: https://github.com/idris-lang/Idris-dev.
- Existing projects on the `Idris Hackers` web space: http://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 http://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 http://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 http://doi.acm.org/10.1145/1863543.1863587