Documentation for the Idris Language

Version 1.0

Contents

CHAPTER 1

The Idris Tutorial

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.

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

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

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.

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.

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.

Getting Started

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 7.6.3.
- The *GNU Multiple Precision Arithmetic Library* (GMP) is available from MacPorts/Homebrew and all major Linux distributions.

Downloading and Installing

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

```
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.

The Interactive Environment

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

```
$ idris
    ____    __     _
   /  _/___/ /____(_)____
   / // __  / ___/ / ___/     Version 1.0
 _/ // /_/ / /  / (__  )      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 1.0
```

```
  _/ // /_/ / /  / (__  )      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
```

Types and Functions

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
```

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 `!`.

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.

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 45))). 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.

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.

"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.

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
```

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
```

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)
```

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.

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.

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), variables or constants (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
```

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.

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.

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
```

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
```

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
  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.

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'
```

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 ]
```

Idiom brackets

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

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

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

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

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

Rather than having to insert `m_app` everywhere there is an application, we can use idiom brackets to do

[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

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 |]
```

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
```

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, sssO, ssssO]" : String
*named_impl> show (sort @{myord} testList)
"[ssssO, sssO, 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.

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.

Modules and Namespaces

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

```
module Btree

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

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

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

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

The modifiers `export` and `public export` say which names are visible from other 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`.

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)

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

public a : AType a = ...
```

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`.

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
```

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
```

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 .

Package Descriptions

A package description includes the following:

- A header, consisting of the keyword package followed by the 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.

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
```

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.

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.

More information

More details, including a complete listing of available fields, can be found in the reference manual in *Packages* (page 151).

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.

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
    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
```

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
```

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)))
```

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 1.0
  _/ // /_/ / /  / (__  )      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.

Views and the "`with`" rule

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.

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, e.g. we have already seen a vector filter function, defined as follows:

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

Here, the `with` clause allows us to deconstruct the result of `filter p xs`. Effectively, it adds this value as an extra argument, which we place after the vertical bar.

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. For example, a `Nat` is either even or odd. If it's 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 the result 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 `|`. 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 43) to complete the definition of `parity`.

[1] Conor McBride and James McKinna. 2004. The view from the left. J. Funct. Program. 14, 1 (January 2004), 69-111. DOI=10.1017/S0956796803004829 http://dx.doi.org/10.1017/S0956796803004829ñ

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
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
```

Theorem Proving

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
```

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 45) 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
```

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 *Theorem Proving* (page 136).

Theorems in Practice

The need to prove theorems can arise naturally in practice. For example, previously (*Views and the "with" rule* (page 40)) we implemented `natToBin` using a function `parity`:

[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

```
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}
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 *Theorem Proving* (page 136)).

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 42) 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.

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.

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.

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 non-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.

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`.

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.

`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;
```

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
```

[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

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.

`dsl` notation

The well-typed interpreter in Section *Example: The Well-Typed Interpreter* (page 36) 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

[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

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:

```
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)
```

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.

Auto 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
```

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}
```

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.

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).

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
```

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].

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))
```

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

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

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
  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.

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.

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.

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 chat.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

CHAPTER 2

Frequently Asked Questions

What are the differences between Agda and Idris?

Like Idris, Agda is a functional language with dependent types, supporting dependent pattern matching. Both can be used for writing programs and proofs. However, Idris has been designed from the start to emphasise general purpose programming rather than theorem proving. As such, it supports interoperability with systems libraries and C programs, and language constructs for domain specific language implementation. It also includes higher level programming constructs such as interfaces (similar to type classes) and do notation.

Idris supports multiple back ends (C and JavaScript by default, with the ability to add more via plugins) and has a reference run time system, written in C, with a garbage collector and built-in message passing concurrency.

Is Idris production ready?

Idris is primarily a research tool for exploring the possibilities of software development with dependent types, meaning that the primary goal is not (yet) to make a system which could be used in production. As such, there are a few rough corners, and lots of missing libraries. Nobody is working on Idris full time, and we don't have the resources at the moment to polish the system on our own. Therefore, we don't recommend building your business around it!

Having said that, contributions which help towards making Idris suitable for use in production would be very welcome - this includes (but is not limited to) extra library support, polishing the run-time system (and ensuring it is robust), providing and maintaining a JVM back end, etc.

Why does Idris use eager evaluation rather than lazy?

Idris uses eager evaluation for more predictable performance, in particular because one of the longer term goals is to be able to write efficient and verified low level code such as device drivers and network

infrastructure. Furthermore, the Idris type system allows us to state precisely the type of each value, and therefore the run-time form of each value. In a lazy language, consider a value of type `Int`:

```
thing : Int
```

What is the representation of `thing` at run-time? Is it a bit pattern representing an integer, or is it a pointer to some code which will compute an integer? In Idris, we have decided that we would like to make this distinction precise, in the type:

```
thing_val : Int
thing_comp : Lazy Int
```

Here, it is clear from the type that `thing_val` is guaranteed to be a concrete `Int`, whereas `thing_comp` is a computation which will produce an `Int`.

How can I make lazy control structures?

You can make control structures using the special Lazy type. For example, `if...then...else...` in Idris expands to an application of a function named `ifThenElse`. The default implementation for Booleans is defined as follows in the library:

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

The type `Lazy a` for `t` and `e` indicates that those arguments will only be evaluated if they are used, that is, they are evaluated lazily.

Evaluation at the REPL doesn't behave as I expect. What's going on?

Being a fully dependently typed language, Idris has two phases where it evaluates things, compile-time and run-time. At compile-time it will only evaluate things which it knows to be total (i.e. terminating and covering all possible inputs) in order to keep type checking decidable. The compile-time evaluator is part of the Idris kernel, and is implemented in Haskell using a HOAS (higher order abstract syntax) style representation of values. Since everything is known to have a normal form here, the evaluation strategy doesn't actually matter because either way it will get the same answer, and in practice it will do whatever the Haskell run-time system chooses to do.

The REPL, for convenience, uses the compile-time notion of evaluation. As well as being easier to implement (because we have the evaluator available) this can be very useful to show how terms evaluate in the type checker. So you can see the difference between:

```
Idris> \n, m => (S n) + m
\n => \m => S (plus n m) : Nat -> Nat -> Nat

Idris> \n, m => n + (S m)
\n => \m => plus n (S m) : Nat -> Nat -> Nat
```

Why can't I use a function with no arguments in a type?

If you use a name in a type which begins with a lower case letter, and which is not applied to any arguments, then Idris will treat it as an implicitly bound argument. For example:

```
append : Vect n ty -> Vect m ty -> Vect (n + m) ty
```

Here, `n`, `m`, and `ty` are implicitly bound. This rule applies even if there are functions defined elsewhere with any of these names. For example, you may also have:

```
ty : Type
ty = String
```

Even in this case, `ty` is still considered implicitly bound in the definition of `append`, rather than making the type of `append` equivalent to...

```
append : Vect n String -> Vect m String -> Vect (n + m) String
```

...which is probably not what was intended! The reason for this rule is so that it is clear just from looking at the type of `append`, and no other context, what the implicitly bound names are.

If you want to use an unapplied name in a type, you have two options. You can either explicitly qualify it, for example, if `ty` is defined in the namespace `Main` you can do the following:

```
append : Vect n Main.ty -> Vect m Main.ty -> Vect (n + m) Main.ty
```

Alternatively, you can use a name which does not begin with a lower case letter, which will never be implicitly bound:

```
Ty : Type
Ty = String

append : Vect n Ty -> Vect m Ty -> Vect (n + m) Ty
```

As a convention, if a name is intended to be used as a type synonym, it is best for it to begin with a capital letter to avoid this restriction.

I have an obviously terminating program, but Idris says it possibly isn't total. Why is that?

Idris can't decide in general whether a program is terminating due to the undecidability of the Halting Problem. It is possible, however, to identify some programs which are definitely terminating. Idris does this using "size change termination" which looks for recursive paths from a function back to itself. On such a path, there must be at least one argument which converges to a base case.

- Mutually recursive functions are supported
- However, all functions on the path must be fully applied. In particular, higher order applications are not supported
- Idris identifies arguments which converge to a base case by looking for recursive calls to syntactically smaller arguments of inputs. e.g. `k` is syntactically smaller than `S (S k)` because `k` is a subterm of `S (S k)`, but `(k, k)` is not syntactically smaller than `(S k, S k)`.

If you have a function which you believe to be terminating, but Idris does not, you can either restructure the program, or use the `assert_total` function.

When will Idris be self-hosting?

It's not a priority, though not a bad idea in the long run. It would be a worthwhile effort in the short term to implement libraries to support self-hosting, such as a good parsing library.

Does Idris have universe polymorphism? What is the type of `Type`?

Rather than universe polymorphism, Idris has a cumulative hierarchy of universes; `Type : Type 1`, `Type 1 : Type 2`, etc. Cumulativity means that if `x : Type n` and `n <= m`, then `x : Type m`. Universe levels are always inferred by Idris, and cannot be specified explicitly. The REPL command `:type Type 1` will result in an error, as will attempting to specify the universe level of any type.

Why does Idris use `Double` instead of `Float64`?

Historically the C language and many other languages have used the names `Float` and `Double` to represent floating point numbers of size 32 and 64 respectively. Newer languages such as Rust and Julia have begun to follow the naming scheme described in IEEE Standard for Floating-Point Arithmetic (IEEE 754). This describes single and double precision numbers as `Float32` and `Float64`; the size is described in the type name.

Due to developer familiarity with the older naming convention, and choice by the developers of Idris, Idris uses the C style convention. That is, the name `Double` is used to describe double precision numbers, and Idris does not support 32 bit floats at present.

What is -ffreestanding?

The freestanding flag is used to build Idris binaries which have their libs and compiler in a relative path. This is useful for building binaries where the install directory is unknown at build time. When passing this flag, the IDRIS_LIB_DIR environment variable needs to be set to the path where the Idris libs reside relative to the idris executable. The IDRIS_TOOLCHAIN_DIR environment variable is optional, if that is set, Idris will use that path to find the C compiler.

Example:

```
IDRIS_LIB_DIR="./libs" IDRIS_TOOLCHAIN_DIR="./mingw/bin" CABALFLAGS="-fffi -ffreestanding -
↪frelease" make
```

What does the name 'Idris' mean?

British people of a certain age may be familiar with this singing dragon. If that doesn't help, maybe you can invent a suitable acronym :-) .

Will there be support for Unicode characters for operators?

There are several reasons why we should not support Unicode operators:

- It's hard to type (this is important if you're using someone else's code, for example). Various editors have their own input methods, but you have to know what they are.
- Not every piece of software easily supports it. Rendering issues have been noted on some mobile email clients, terminal-based IRC clients, web browsers, etc. There are ways to resolve these rendering issues but they provide a barrier to entry to using Idris.
- Even if we leave it out of the standard library (which we will in any case!) as soon as people start using it in their library code, others have to deal with it.
- Too many characters look too similar. We had enough trouble with confusion between 0 and O without worrying about all the different kinds of colons and brackets.
- There seems to be a tendency to go over the top with use of Unicode. For example, using sharp and flat for delay and force (or is it the other way around?) in Agda seems gratuitous. We don't want to encourage this sort of thing, when words are often better.

With care, Unicode operators can make things look pretty but so can `lhs2TeX`. Perhaps in a few years time things will be different and software will cope better and it will make sense to revisit this. For now, however, Idris will not be offering arbitrary Unicode symbols in operators.

This seems like an instance of Wadler's Law in action.

This answer is based on Edwin Brady's response in the following pull request.

Where can I find more answers?

There is an Unofficial FAQ on the wiki on GitHub which answers more technical questions and may be updated more often.

CHAPTER 3

Implementing State-aware Systems in Idris: The ST Tutorial

A tutorial on implementing state-aware systems using the *Control.ST* library in *Idris*.

Overview

Pure functional languages with dependent types such as Idris support reasoning about programs directly in the type system, promising that we can *know* a program will run correctly (i.e. according to the specification in its type) simply because it compiles.

Realistically, though, software relies on state, and many components rely on state machines. For example, they describe network transport protocols like TCP, and implement event-driven systems and regular expression matching. Furthermore, many fundamental resources like network sockets and files are, implicitly, managed by state machines, in that certain operations are only valid on resources in certain states, and those operations can change the states of the underlying resource. For example, it only makes sense to send a message on a connected network socket, and closing a socket changes its state from "open" to "closed". State machines can also encode important security properties. For example, in the software which implements an ATM, it's important that the ATM dispenses cash only when the machine is in a state where a card has been inserted and the PIN verified.

In this tutorial we will introduce the `Control.ST` library, which is included with the Idris distribution (currently as part of the `contrib` package) and supports programming and reasoning with state and side effects. This tutorial assumes familiarity with pure programming in Idris, as described in *The Idris Tutorial* (page 2). For further background information, the `ST` library is based on ideas discussed in Chapter 13 (available as a free sample chapter) and Chapter 14 of Type-Driven Development with Idris.

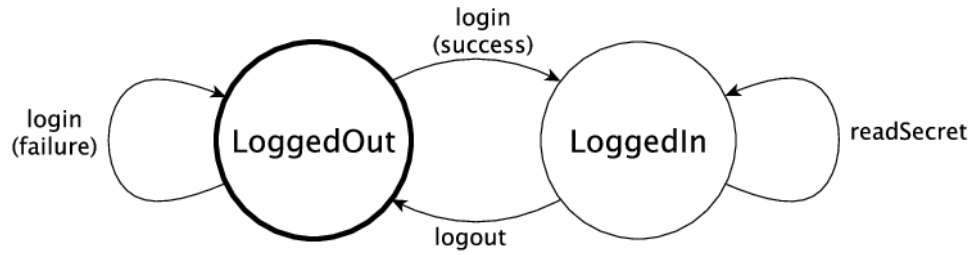
The `ST` library allows us to write programs which are composed of multiple state transition systems. It supports composition in two ways: firstly, we can use several independently implemented state transition systems at once; secondly, we can implement one state transition system in terms of others.

Introductory example: a data store requiring a login

Many software components rely on some form of state, and there may be operations which are only valid in specific states. For example, consider a secure data store in which a user must log in before getting access to some secret data. This system can be in one of two states:

- `LoggedIn`, in which the user is allowed to read the secret
- `LoggedOut`, in which the user has no access to the secret

We can provide commands to log in, log out, and read the data, as illustrated in the following diagram:



The `login` command, if it succeeds, moves the overall system state from `LoggedOut` to `LoggedIn`. The `logout` command moves the state from `LoggedIn` to `LoggedOut`. Most importantly, the `readSecret` command is only valid when the system is in the `LoggedIn` state.

We routinely use type checkers to ensure that variables and arguments are used consistently. However, statically checking that operations are performed only on resources in an appropriate state is not well supported by mainstream type systems. In the data store example, for example, it's important to check that the user is successfully logged in before using `readSecret`. The `ST` library allows us to represent this kind of *protocol* in the type system, and ensure at *compile-time* that the secret is only read when the user is logged in.

Outline

This tutorial starts (*Introducing ST: Working with State* (page 69)) by describing how to manipulate individual states, introduce a data type `STrans` for describing stateful functions, and `ST` which describes top level state transitions. Next (*State Machines in Types* (page 77)) it describes how to represent state machines in types, and how to define *interfaces* for describing stateful systems. Then (*Composing State Machines* (page 84)) it describes how to compose systems of multiple state machines. It explains how to implement systems which use several state machines at once, and how to implement a high level stateful system in terms of lower level systems. Finally (*Example: Network Socket Programming* (page 95)) we'll see a specific example of a stateful API in practice, implementing the POSIX network sockets API.

The `Control.ST` library is also described in a draft paper by Edwin Brady, "State Machines All The Way Down", available here. This paper presents many of the examples from this tutorial, and describes the motivation, design and implementation of the library in more depth.

Introducing ST: Working with State

The `Control.ST` library provides facilities for creating, reading, writing and destroying state in Idris functions, and tracking changes of state in a function's type. It is based around the concept of *resources*, which are, essentially, mutable variables, and a dependent type, `STrans` which tracks how those resources change when a function runs:

```
STrans : (m : Type -> Type) ->
         (resultType : Type) ->
         (in_res : Resources) ->
         (out_res : resultType -> Resources) ->
         Type
```

A value of type `STrans m resultType in_res out_res_fn` represents a sequence of actions which can manipulate state. The arguments are:

- `m`, which is an underlying *computation context* in which the actions will be executed. Usually, this will be a generic type with a `Monad` implementation, but it isn't necessarily so. In particular, there is no need to understand monads to be able to use `ST` effectively!
- `resultType`, which is the type of the value the sequence will produce
- `in_res`, which is a list of *resources* available *before* executing the actions.
- `out_res`, which is a list of resources available *after* executing the actions, and may differ depending on the result of the actions.

We can use `STrans` to describe *state transition systems* in a function's type. We'll come to the definition of `Resources` shortly, but for the moment you can consider it an abstract representation of the "state of the world". By giving the input resources (`in_res`) and the output resources (`out_res`) we are describing the *preconditions* under which a function is allowed to execute, and *postconditions* which describe how a function affects the overall state of the world.

We'll begin in this section by looking at some small examples of `STrans` functions, and see how to execute them. We'll also introduce `ST`, a type-level function which allows us to describe the state transitions of a stateful function concisely.

Type checking the examples

For the examples in this section, and throughout this tutorial, you'll need to `import Control.ST` and add the `contrib` package by passing the `-p contrib` flag to `idris`.

Introductory examples: manipulating `State`

An `STrans` function explains, in its type, how it affects a collection of `Resources`. A resource has a *label* (of type `Var`), which we use to refer to the resource throughout the function, and we write the state of a resource, in the `Resources` list, in the form `label ::: type`.

For example, the following function has a resource `x` available on input, of type `State Integer`, and that resource is still a `State Integer` on output:

```
increment : (x : Var) -> STrans m () [x ::: State Integer]
                                     (const [x ::: State Integer])
increment x = do num <- read x
                 write x (num + 1)
```

Verbosity of the type of `increment`

The type of `increment` may seem somewhat verbose, in that the *input* and *output* resources are repeated, even though they are the same. We'll introduce a much more concise way of writing this type at the end of this section (*ST: Representing state transitions directly* (page 76)), when we describe the `ST` type itself.

This function reads the value stored at the resource `x` with `read`, increments it then writes the result back into the resource `x` with `write`. We'll see the types of `read` and `write` shortly (see *STrans Primitive operations* (page 74)). We can also create and delete resources:

```
makeAndIncrement : Integer -> STrans m Integer [] (const [])
makeAndIncrement init = do var <- new init
                           increment var
                           x <- read var
                           delete var
                           pure x
```

The type of `makeAndIncrement` states that it has *no* resources available on entry (`[]`) or exit (`const []`). It creates a new `State` resource with `new` (which takes an initial value for the resource), increments the value, reads it back, then deletes it using `delete`, returning the final value of the resource. Again, we'll see the types of `new` and `delete` shortly.

The `m` argument to `STrans` (of type `Type -> Type`) is the *computation context* in which the function can be run. Here, the type level variable indicates that we can run it in *any* context. We can run it in the identity context with `runPure`. For example, try entering the above definitions in a file `Intro.idr` then running the following at the REPL:

```
*Intro> runPure (makeAndIncrement 93)
94 : Integer
```

It's a good idea to take an interactive, type-driven approach to implementing `STrans` programs. For example, after creating the resource with `new init`, you can leave a *hole* for the rest of the program to see how creating the resource has affected the type:

```
makeAndIncrement : Integer -> STrans m Integer [] (const [])
makeAndIncrement init = do var <- new init
                           ?whatNext
```

If you check the type of `?whatNext`, you'll see that there is now a resource available, `var`, and that by the end of the function there should be no resource available:

```
  init : Integer
  m : Type -> Type
  var : Var
--------------------------------------
whatNext : STrans m Integer [var ::: State Integer] (\value => [])
```

These small examples work in any computation context `m`. However, usually, we are working in a more restricted context. For example, we might want to write programs which only work in a context that supports interactive programs. For this, we'll need to see how to *lift* operations from the underlying context.

Lifting: Using the computation context

Let's say that, instead of passing an initial integer to `makeAndIncrement`, we want to read it in from the console. Then, instead of working in a generic context `m`, we can work in the specific context `IO`:

```
ioMakeAndIncrement : STrans IO () [] (const [])
```

This gives us access to `IO` operations, via the `lift` function. We can define `ioMakeAndIncrement` as follows:

```
ioMakeAndIncrement : STrans IO () [] (const [])
ioMakeAndIncrement
   = do lift $ putStr "Enter a number: "
        init <- lift $ getLine
        var <- new (cast init)
        lift $ putStrLn ("var = " ++ show !(read var))
        increment var
        lift $ putStrLn ("var = " ++ show !(read var))
        delete var
```

The `lift` function allows us to use funtions from the underlying computation context (`IO` here) directly. Again, we'll see the exact type of `lift` shortly.

!-notation

In `ioMakeAndIncrement` we've used `!(read var)` to read from the resource. You can read about this `!`-notation in the main Idris tutorial (see *Monads and do-notation* (page 25)). In short, it allows us to use an `STrans` function inline, rather than having to bind the result to a variable first.

Conceptually, at least, you can think of it as having the following type:

```
(!) : STrans m a state_in state_out -> a
```

It is syntactic sugar for binding a variable immediately before the current action in a `do` block, then using that variable in place of the `!`-expression.

In general, though, it's bad practice to use a *specific* context like `IO`. Firstly, it requires us to sprinkle `lift` liberally throughout our code, which hinders readability. Secondly, and more importantly, it will limit the safety of our functions, as we'll see in the next section (*State Machines in Types* (page 77)).

So, instead, we define *interfaces* to restrict the computation context. For example, `Control.ST` defines a `ConsoleIO` interface which provides the necessary methods for performing basic console interaction:

```
interface ConsoleIO (m : Type -> Type) where
  putStr : String -> STrans m () res (const res)
  getStr : STrans m String res (const res)
```

That is, we can write to and read from the console with any available resources `res`, and neither will affect the available resources. This has the following implementation for `IO`:

```
ConsoleIO IO where
  putStr str = lift (Interactive.putStr str)
  getStr = lift Interactive.getLine
```

Now, we can define `ioMakeAndIncrement` as follows:

```
ioMakeAndIncrement : ConsoleIO io => STrans io () [] (const [])
ioMakeAndIncrement
```

```
= do putStr "Enter a number: "
     init <- getStr
     var <- new (cast init)
     putStrLn ("var = " ++ show !(read var))
     increment var
     putStrLn ("var = " ++ show !(read var))
     delete var
```

Instead of working in `IO` specifically, this works in a generic context `io`, provided that there is an implementation of `ConsoleIO` for that context. This has several advantages over the first version:

- All of the calls to `lift` are in the implementation of the interface, rather than `ioMakeAndIncrement`
- We can provide alternative implementations of `ConsoleIO`, perhaps supporting exceptions or logging in addition to basic I/O.
- As we'll see in the next section (*State Machines in Types* (page 77)), it will allow us to define safe APIs for manipulating specific resources more precisely.

Earlier, we used `runPure` to run `makeAndIncrement` in the identity context. Here, we use `run`, which allows us to execute an `STrans` program in any context (as long as it has an implementation of `Applicative`) and we can execute `ioMakeAndIncrement` at the REPL as follows:

```
*Intro> :exec run ioMakeAndIncrement
Enter a number: 93
var = 93
var = 94
```

Manipulating `State` with dependent types

In our first example of `State`, when we incremented the value its *type* remained the same. However, when we're working with *dependent* types, updating a state may also involve updating its type. For example, if we're adding an element to a vector stored in a state, its length will change:

```
addElement : (vec : Var) -> (item : a) ->
             STrans m () [vec ::: State (Vect n a)]
                  (const [vec ::: State (Vect (S n) a)])
addElement vec item = do xs <- read vec
                         write vec (item :: xs)
```

Note that you'll need to `import Data.Vect` to try this example.

Updating a state directly with `update`

Rather than using `read` and `write` separately, you can also use `update` which reads from a `State`, applies a function to it, then writes the result. Using `update` you could write `addElement` as follows:

```
addElement : (vec : Var) -> (item : a) ->
             STrans m () [vec ::: State (Vect n a)]
                  (const [vec ::: State (Vect (S n) a)])
addElement vec item = update vec (item ::)
```

We don't always know *how* exactly the type will change in the course of a sequence actions, however. For example, if we have a state containing a vector of integers, we might read an input from the console and only add it to the vector if the input is a valid integer. Somehow, we need a different type for the output state depending on whether reading the integer was successful, so neither of the following types

is quite right:

```
readAndAdd_OK : ConsoleIO io => (vec : Var) ->
                STrans m ()  -- Returns an empty tuple
                           [vec ::: State (Vect n Integer)]
                     (const [vec ::: State (Vect (S n) Integer)])
readAndAdd_Fail : ConsoleIO io => (vec : Var) ->
                  STrans m ()  -- Returns an empty tuple
                             [vec ::: State (Vect n Integer)]
                       (const [vec ::: State (Vect n Integer)])
```

Remember, though, that the *output* resource types can be *computed* from the result of a function. So far, we've used `const` to note that the output resources are always the same, but here, instead, we can use a type level function to *calculate* the output resources. We start by returning a `Bool` instead of an empty tuple, which is `True` if reading the input was successful, and leave a *hole* for the output resources:

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans m Bool [vec ::: State (Vect n Integer)]
                           ?output_res
```

If you check the type of `?output_res`, you'll see that Idris expects a function of type `Bool -> Resources`, meaning that the output resource type can be different depending on the result of `readAndAdd`:

```
  n : Nat
  m : Type -> Type
  io : Type -> Type
  constraint : ConsoleIO io
  vec : Var
--------------------------------------
output_res : Bool -> Resources
```

So, the output resource is either a `Vect n Integer` if the input is invalid (i.e. `readAndAdd` returns `False`) or a `Vect (S n) Integer` if the input is valid. We can express this in the type as follows:

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans io Bool [vec ::: State (Vect n Integer)]
                    (\res => [vec ::: State (if res then Vect (S n) Integer
                                                    else Vect n Integer)])
```

Then, when we implement `readAndAdd` we need to return the appropriate value for the output state. If we've added an item to the vector, we need to return `True`, otherwise we need to return `False`:

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans io Bool [vec ::: State (Vect n Integer)]
                    (\res => [vec ::: State (if res then Vect (S n) Integer
                                                    else Vect n Integer)])
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then do
                         update vec ((cast num) ::)
                         pure True     -- added an item, so return True
                       else pure False -- didn't add, so return False
```

There is a slight difficulty if we're developing interactively, which is that if we leave a hole, the required output state isn't easily visible until we know the value that's being returned. For example. in the following incomplete definition of `readAndAdd` we've left a hole for the successful case:

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
```

```
                    then ?whatNow
                    else pure False
```

We can look at the type of `?whatNow`, but it is unfortunately rather less than informative:

```
  vec : Var
  n : Nat
  io : Type -> Type
  constraint : ConsoleIO io
  num : String
--------------------------------------
whatNow : STrans io Bool [vec ::: State (Vect (S n) Integer)]
                 (\res =>
                    [vec :::
                     State (ifThenElse res
                                     (Delay (Vect (S n) Integer))
                                     (Delay (Vect n Integer)))])
```

The problem is that we'll only know the required output state when we know the value we're returning. To help with interactive development, `Control.ST` provides a function `returning` which allows us to specify the return value up front, and to update the state accordingly. For example, we can write an incomplete `readAndAdd` as follows:

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then returning True ?whatNow
                       else pure False
```

This states that, in the successful branch, we'll be returning `True`, and `?whatNow` should explain how to update the states appropriately so that they are correct for a return value of `True`. We can see this by checking the type of `?whatNow`, which is now a little more informative:

```
  vec : Var
  n : Nat
  io : Type -> Type
  constraint : ConsoleIO io
  num : String
--------------------------------------
whatnow : STrans io () [vec ::: State (Vect n Integer)]
                 (\value => [vec ::: State (Vect (S n) Integer)])
```

This type now shows, in the output resource list of `STrans`, that we can complete the definition by adding an item to `vec`, which we can do as follows:

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then returning True (update vec ((cast num) ::))
                       else returning False (pure ()) -- returning False, so no state update␣
↪required
```

`STrans` Primitive operations

Now that we've written a few small examples of `STrans` functions, it's a good time to look more closely at the types of the state manipulation functions we've used. First, to read and write states, we've used `read` and `write`:

```
read : (lbl : Var) -> {auto prf : InState lbl (State ty) res} ->
       STrans m ty res (const res)
write : (lbl : Var) -> {auto prf : InState lbl ty res} ->
        (val : ty') ->
        STrans m () res (const (updateRes res prf (State ty')))
```

These types may look a little daunting at first, particularly due to the implicit `prf` argument, which has the following type:

```
prf : InState lbl (State ty) res}
```

This relies on a predicate `InState`. A value of type `InState x ty res` means that the reference `x` must have type `ty` in the list of resources `res`. So, in practice, all this type means is that we can only read or write a resource if a reference to it exists in the list of resources.

Given a resource label `res`, and a proof that `res` exists in a list of resources, `updateRes` will update the type of that resource. So, the type of `write` states that the type of the resource will be updated to the type of the given value.

The type of `update` is similar to that for `read` and `write`, requiring that the resource has the input type of the given function, and updating it to have the output type of the function:

```
update : (lbl : Var) -> {auto prf : InState lbl (State ty) res} ->
         (ty -> ty') ->
         STrans m () res (const (updateRes res prf (State ty')))
```

The type of `new` states that it returns a `Var`, and given an initial value of type `state`, the output resources contains a new resource of type `State state`:

```
new : (val : state) ->
      STrans m Var res (\lbl => (lbl ::: State state) :: res)
```

It's important that the new resource has type `State state`, rather than merely `state`, because this will allow us to hide implementation details of APIs. We'll see more about what this means in the next section, *State Machines in Types* (page 77).

The type of `delete` states that the given label will be removed from the list of resources, given an implicit proof that the label exists in the input resources:

```
delete : (lbl : Var) -> {auto prf : InState lbl (State st) res} ->
         STrans m () res (const (drop res prf))
```

Here, `drop` is a type level function which updates the resource list, removing the given resource `lbl` from the list.

We've used `lift` to run functions in the underlying context. It has the following type:

```
lift : Monad m => m t -> STrans m t res (const res)
```

Given a `result` value, `pure` is an `STrans` program which produces that value, provided that the current list of resources is correct when producing that value:

```
pure : (result : ty) -> STrans m ty (out_fn result) out_fn
```

We can use `returning` to break down returning a value from an `STrans` functions into two parts: providing the value itself, and updating the resource list so that it is appropriate for returning that value:

```
returning : (result : ty) ->
            STrans m () res (const (out_fn result)) ->
            STrans m ty res out_fn
```

Finally, we've used `run` and `runPure` to execute `STrans` functions in a specific context. `run` will execute a function in any context, provided that there is an `Applicative` implementation for that context, and `runPure` will execute a function in the identity context:

```
run : Applicative m => STrans m a [] (const []) -> m a
runPure : STrans Basics.id a [] (const []) -> a
```

Note that in each case, the input and output resource list must be empty. There's no way to provide an initial resource list, or extract the final resources. This is deliberate: it ensures that *all* resource management is carried out in the controlled `STrans` environment and, as we'll see, this allows us to implement safe APIs with precise types explaining exactly how resources are tracked throughout a program.

These functions provide the core of the `ST` library; there are some others which we'll encounter later, for more advanced situations, but the functions we have seen so far already allow quite sophisticated state-aware programming and reasoning in Idris.

ST: Representing state transitions directly

We've seen a few examples of small `STrans` functions now, and their types can become quite verbose given that we need to provide explicit input and output resource lists. This is convenient for giving types for the primitive operations, but for more general use it's much more convenient to be able to express *transitions* on individual resources, rather than giving input and output resource lists in full. We can do this with `ST`:

```
ST : (m : Type -> Type) ->
     (resultType : Type) ->
     List (Action resultType) -> Type
```

`ST` is a type level function which computes an appropriate `STrans` type given a list of *actions*, which describe transitions on resources. An `Action` in a function type can take one of the following forms (plus some others which we'll see later in the tutorial):

- `lbl ::: ty` expresses that the resource `lbl` begins and ends in the state `ty`
- `lbl ::: ty_in :-> ty_out` expresses that the resource `lbl` begins in state `ty_in` and ends in state `ty_out`
- `lbl ::: ty_in :-> (\res -> ty_out)` expresses that the resource `lbl` begins in state `ty_in` and ends in a state `ty_out`, where `ty_out` is computed from the result of the function `res`.

So, we can write some of the function types we've seen so far as follows:

```
increment : (x : Var) -> ST m () [x ::: State Integer]
```

That is, `increment` begins and ends with `x` in state `State Integer`.

```
makeAndIncrement : Int -> ST m Int []
```

That is, `makeAndIncrement` begins and ends with no resources.

```
addElement : (vec : Var) -> (item : a) ->
             ST m () [vec ::: State (Vect n a) :-> State (Vect (S n) a)]
```

That is, `addElement` changes `vec` from `State (Vect n a)` to `State (Vect (S n) a)`.

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             ST io Bool
                   [vec ::: State (Vect n Integer) :->
                    \res => State (if res then Vect (S n) Integer
                                          else Vect n Integer)]
```

By writing the types in this way, we express the minimum necessary to explain how each function affects the overall resource state. If there is a resource update depending on a result, as with `readAndAdd`, then we need to describe it in full. Otherwise, as with `increment` and `makeAndIncrement`, we can write the input and output resource lists without repetition.

An `Action` can also describe *adding* and *removing* states:

- `add ty`, assuming the operation returns a `Var`, adds a new resource of type `ty`.
- `remove lbl ty` expresses that the operation removes the resource named `lbl`, beginning in state `ty` from the resource list.

So, for example, we can write:

```
newState : ST m Var [add (State Int)]
removeState : (lbl : Var) -> ST m () [remove lbl (State Int)]
```

The first of these, `newState`, returns a new resource label, and adds that resource to the list with type `State Int`. The second, `removeState`, given a label `lbl`, removes the resource from the list. These types are equivalent to the following:

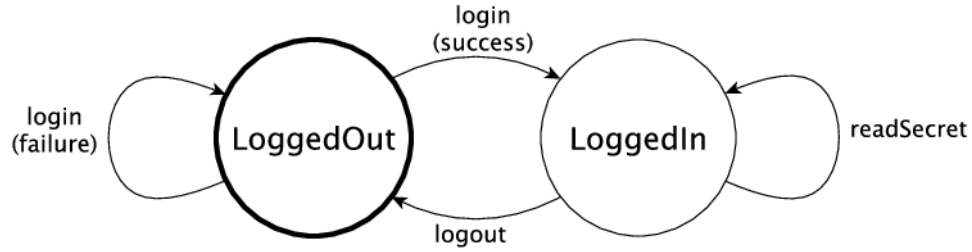
```
newState : STrans m Var [] (\lbl => [lbl ::: State Int])
removeState : (lbl : Var) -> STrans m () [lbl ::: State Int] (const [])
```

These are the primitive methods of constructing an `Action`. Later, we will encounter some other ways using type level functions to help with readability.

In the remainder of this tutorial, we will generally use `ST` except on the rare occasions we need the full precision of `STrans`. In the next section, we'll see how to use the facilities provided by `ST` to write a precise API for a system with security properties: a data store requiring a login.

State Machines in Types

In the introduction, we saw the following state transition diagram representing the (abstract) states of a data store, and the actions we can perform on the store:



We say that these are the *abstract* states of the store, because the concrete state will contain a lot more information: for example, it might contain user names, hashed passwords, the store contents, and so on. However, as far as we are concerned for the actions `login`, `logout` and `readSecret`, it's whether we are

logged in or not which affects which are valid.

We've seen how to manipulate states using `ST`, and some small examples of dependent types in states. In this section, we'll see how to use `ST` to provide a safe API for the data store. In the API, we'll encode the above diagram in the types, in such a way that we can only execute the operations `login`, `logout` and `readSecret` when the state is valid.

So far, we've used `State` and the primitive operations, `new`, `read`, `write` and `delete` to manipulate states. For the data store API, however, we'll begin by defining an *interface* (see *Interfaces* (page 22) in the Idris tutorial) which describes the operations on the store, and explains in their types exactly when each operation is valid, and how it affects the store's state. By using an interface, we can be sure that this is the *only* way to access the store.

Defining an interface for the data store

We'll begin by defining a data type, in a file `Login.idr`, which represents the two abstract states of the store, either `LoggedOut` or `LoggedIn`:

```
data Access = LoggedOut | LoggedIn
```

We can define a data type for representing the current state of a store, holding all of the necessary information (this might be user names, hashed passwords, store contents and so on) and parameterise it by the logged in status of the store:

```
Store : Access -> Type
```

Rather than defining a concrete type now, however, we'll include this in a data store *interface* and define a concrete type later:

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type
```

We can continue to populate this interface with operations on the store. Among other advantages, by separating the *interface* from its *implementation* we can provide different concrete implementations for different contexts. Furthermore, we can write programs which work with a store without needing to know any details of how the store is implemented.

We'll need to be able to `connect` to a store, and `disconnect` when we're done. Add the following methods to the `DataStore` interface:

```
connect : ST m Var [add (Store LoggedOut)]
disconnect : (store : Var) -> ST m () [remove store (Store LoggedOut)]
```

The type of `connect` says that it returns a new resource which has the initial type `Store LoggedOut`. Conversely, `disconnect`, given a resource in the state `Store LoggedOut`, removes that resource. We can see more clearly what `connect` does by trying the following (incomplete) definition:

```
doConnect : DataStore m => ST m () []
doConnect = do st <- connect
               ?whatNow
```

Note that we're working in a *generic* context `m`, constrained so that there must be an implementation of `DataStore` for `m` to be able to execute `doConnect`. If we check the type of `?whatNow`, we'll see that the remaining operations begin with a resource `st` in the state `Store LoggedOut`, and we need to finish with no resources.

```
  m : Type -> Type
  constraint : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [st ::: Store LoggedOut] (\result => [])
```

Then, we can remove the resource using `disconnect`:

```
doConnect : DataStore m => ST m () []
doConnect = do st <- connect
               disconnect st
               ?whatNow
```

Now checking the type of `?whatNow` shows that we have no resources available:

```
  m : Type -> Type
  constraint : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [] (\result => [])
```

To continue our implementation of the `DataStore` interface, next we'll add a method for reading the secret data. This requires that the `store` is in the state `Store LoggedIn`:

```
readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
```

At this point we can try writing a function which connects to a store, reads the secret, then disconnects. However, it will be unsuccessful, because `readSecret` requires us to be logged in:

```
badGet : DataStore m => ST m () []
badGet = do st <- connect
            secret <- readSecret st
            disconnect st
```

This results in the following error, because `connect` creates a new store in the `LoggedOut` state, and `readSecret` requires the store to be in the `LoggedIn` state:

```
When checking an application of function Control.ST.>>=:
    Error in state transition:
            Operation has preconditions: [st ::: Store LoggedOut]
            States here are: [st ::: Store LoggedIn]
            Operation has postconditions: \result => []
            Required result states here are: \result => []
```

The error message explains how the required input states (the preconditions) and the required output states (the postconditions) differ from the states in the operation. In order to use `readSecret`, we'll need a way to get from a `Store LoggedOut` to a `Store LoggedIn`. As a first attempt, we can try the following type for `login`:

```
login : (store : Var) -> ST m () [store ::: Store LoggedOut :-> Store LoggedIn] -- Incorrect␣
↪type!
```

Note that in the *interface* we say nothing about *how* `login` works; merely how it affects the overall state. Even so, there is a problem with the type of `login`, because it makes the assumption that it will always succeed. If it fails - for example because the implementation prompts for a password and the user enters the password incorrectly - then it must not result in a `LoggedIn` store.

Instead, therefore, `login` will return whether logging in was successful, via the following type;

```
data LoginResult = OK | BadPassword
```

Then, we can *calculate* the result state (see *Manipulating State with dependent types* (page 72)) from the result. Add the following method to the `DataStore` interface:

```
login : (store : Var) ->
        ST m LoginResult [store ::: Store LoggedOut :->
                          (\res => Store (case res of
                                               OK => LoggedIn
                                               BadPassword => LoggedOut))]
```

If `login` was successful, then the state after `login` is `Store LoggedIn`. Otherwise, the state is `Store LoggedOut`.

To complete the interface, we'll add a method for logging out of the store. We'll assume that logging out is always successful, and moves the store from the `Store LoggedIn` state to the `Store LoggedOut` state.

```
logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
```

This completes the interface, repeated in full for reference below:

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type

  connect : ST m Var [add (Store LoggedOut)]
  disconnect : (store : Var) -> ST m () [remove store (Store LoggedOut)]

  readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
  login : (store : Var) ->
          ST m LoginResult [store ::: Store LoggedOut :->
                            (\res => Store (case res of
                                                 OK => LoggedIn
                                                 BadPassword => LoggedOut))]
  logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
```

Before we try creating any implementations of this interface, let's see how we can write a function with it, to log into a data store, read the secret if login is successful, then log out again.

Writing a function with the data store

As an example of working with the `DataStore` interface, we'll write a function `getData`, which connects to a store in order to read some data from it. We'll write this function interactively, step by step, using the types of the operations to guide its development. It has the following type:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
```

This type means that there are no resources available on entry or exit. That is, the overall list of actions is `[]`, meaning that at least externally, the function has no overall effect on the resources. In other words, for every resource we create during `getData`, we'll also need to delete it before exit.

Since we want to use methods of the `DataStore` interface, we'll constraint the computation context `m` so that there must be an implementation of `DataStore`. We also have a constraint `ConsoleIO m` so that we can display any data we read from the store, or any error messages.

We start by connecting to the store, creating a new resource `st`, then trying to `login`:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             ?whatNow
```

Logging in will either succeed or fail, as reflected by the value of `ok`. If we check the type of `?whatNow`, we'll see what state the store currently has:

```
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : DataStore m
  st : Var
  ok : LoginResult
--------------------------------------
whatNow : STrans m () [st ::: Store (case ok of
                                       OK => LoggedIn
                                       BadPassword => LoggedOut)]
                      (\result => [])
```

The current state of `st` therefore depends on the value of `ok`, meaning that we can make progress by case splitting on `ok`:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => ?whatNow_1
                  BadPassword => ?whatNow_2
```

The types of the holes in each branch, `?whatNow_1` and `?whatNow_2`, show how the state changes depending on whether logging in was successful. If it succeeded, the store is `LoggedIn`:

```
--------------------------------------
whatNow_1 : STrans m () [st ::: Store LoggedIn] (\result => [])
```

On the other hand, if it failed, the store is `LoggedOut`:

```
--------------------------------------
whatNow_2 : STrans m () [st ::: Store LoggedOut] (\result => [])
```

In `?whatNow_1`, since we've successfully logged in, we can now read the secret and display it to the console:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => do secret <- readSecret st
                           putStrLn ("Secret is: " ++ show secret)
                           ?whatNow_1
                  BadPassword => ?whatNow_2
```

We need to finish the `OK` branch with no resources available. We can do this by logging out of the store then disconnecting:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => do secret <- readSecret st
```

```
                    putStrLn ("Secret is: " ++ show secret)
                    logout st
                    disconnect st
                BadPassword => ?whatNow_2
```

Note that we *must* `logout` of `st` before calling `disconnect`, because `disconnect` requires that the store is in the `LoggedOut` state.

Furthermore, we can't simply use `delete` to remove the resource, as we did with the `State` examples in the previous section, because `delete` only works when the resource has type `State ty`, for some type `ty`. If we try to use `delete` instead of `disconnect`, we'll see an error message like the following:

```
When checking argument prf to function Control.ST.delete:
        Can't find a value of type
                InState st (State st) [st ::: Store LoggedOut]
```

In other words, the type checker can't find a proof that the resource `st` has a type of the form `State st`, because its type is `Store LoggedOut`. Since `Store` is part of the `DataStore` interface, we *can't* yet know the concrete representation of the `Store`, so we need to remove the resource via the interface, with `disconnect`, rather than directly with `delete`.

We can complete `getData` as follows, using a pattern matching bind alternative (see the Idris tutorial, *Monads and do-notation* (page 25)) rather than a `case` statement to catch the possibilty of an error with `login`:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

We can't yet try this out, however, because we don't have any implementations of `getData`! If we try to execute it in an `IO` context, for example, we'll get an error saying that there's no implementation of `DataStore IO`:

```
*Login> :exec run {m = IO} getData
When checking an application of function Control.ST.run:
        Can't find implementation for DataStore IO
```

The final step in implementing a data store which correctly follows the state transition diagram, therefore, is to provide an implementation of `DataStore`.

Implementing the interface

To execute `getData` in `IO`, we'll need to provided an implementation of `DataStore` which works in the `IO` context. We can begin as follows:

```
implementation DataStore IO where
```

Then, we can ask Idris to populate the interface with skeleton definitions for the necessary methods (press `Ctrl-Alt-A` in Atom for "add definition" or the corresponding shortcut for this in the Idris mode in your favourite editor):

```
implementation DataStore IO where
  Store x = ?DataStore_rhs_1
  connect = ?DataStore_rhs_2
  disconnect store = ?DataStore_rhs_3
  readSecret store = ?DataStore_rhs_4
  login store = ?DataStore_rhs_5
  logout store = ?DataStore_rhs_6
```

The first decision we'll need to make is how to represent the data store. We'll keep this simple, and store the data as a single `String`, using a hard coded password to gain access. So, we can define `Store` as follows, using a `String` to represent the data no matter whether we are `LoggedOut` or `LoggedIn`:

```
Store x = State String
```

Now that we've given a concrete type for `Store`, we can implement operations for connecting, disconnecting, and accessing the data. And, since we used `State`, we can use `new`, `delete`, `read` and `write` to manipulate the store.

Looking at the types of the holes tells us how we need to manipulate the state. For example, the `?DataStore_rhs_2` hole tells us what we need to do to implement `connect`. We need to return a new `Var` which represents a resource of type `State String`:

```
--------------------------------------
DataStore_rhs_2 : STrans IO Var [] (\result => [result ::: State String])
```

We can implement this by creating a new variable with some data for the content of the store (we can use any `String` for this) and returning that variable:

```
connect = do store <- new "Secret Data"
             pure store
```

For `disconnect`, we only need to delete the resource:

```
disconnect store = delete store
```

For `readSecret`, we need to read the secret data and return the `String`. Since we now know the concrete representation of the data is a `State String`, we can use `read` to access the data directly:

```
readSecret store = read store
```

We'll do `logout` next and return to `login`. Checking the hole reveals the following:

```
  store : Var
--------------------------------------
DataStore_rhs_6 : STrans IO () [store ::: State String] (\result => [store ::: State String])
```

So, in this minimal implementation, we don't actually have to do anything!

```
logout store = pure ()
```

For `login`, we need to return whether logging in was successful. We'll do this by prompting for a password, and returning `OK` if it matches a hard coded password, or `BadPassword` otherwise:

```
login store = do putStr "Enter password: "
                 p <- getStr
                 if p == "Mornington Crescent"
                    then pure OK
                    else pure BadPassword
```

For reference, here is the complete implementation which allows us to execute a `DataStore` program at the REPL:

```
implementation DataStore IO where
  Store x = State String
  connect = do store <- new "Secret Data"
               pure store
  disconnect store = delete store
  readSecret store = read store
  login store = do putStr "Enter password: "
                   p <- getStr
                   if p == "Mornington Crescent"
                      then pure OK
                      else pure BadPassword
  logout store = pure ()
```

Finally, we can try this at the REPL as follows (Idris defaults to the `IO` context at the REPL if there is an implementation available, so no need to give the `m` argument explicitly here):

```
*Login> :exec run getData
Enter password: Mornington Crescent
Secret is: "Secret Data"

*Login> :exec run getData
Enter password: Dollis Hill
Failure
```

We can only use `read`, `write`, `new` and `delete` on a resource with a `State` type. So, *within* the implementation of `DataStore`, or anywhere where we know the context is `IO`, we can access the data store however we like: this is where the internal details of `DataStore` are implemented. However, if we merely have a constraint `DataStore m`, we can't know how the store is implemented, so we can only access via the API given by the `DataStore` interface.

It is therefore good practice to use a *generic* context `m` for functions like `getData`, and constrain by only the interfaces we need, rather than using a concrete context `IO`.

We've now seen how to manipulate states, and how to encapsulate state transitions for a specific system like the data store inn an interface. However, realistic systems will need to *compose* state machines. We'll either need to use more than one state machine at a time, or implement one state machine in terms of one or more others. We'll see how to achieve this in the next section.

Composing State Machines

In the previous section, we defined a `DataStore` interface and used it to implement the following small program which allows a user to log in to the store then display the store's contents;

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

This function only uses one state, the store itself. Usually, though, larger programs have lots of states, and might add, delete and update states over the course of its execution. Here, for example, a useful

extension might be to loop forever, keeping count of the number of times there was a login failure in a state.

Furthermore, we may have *hierarchies* of state machines, in that one state machine could be implemented by composing several others. For example, we can have a state machine representing the state of a graphics system, and use this to implement a *higher level* graphics API such as turtle graphics, which uses the graphics system plus some additional state for the turtle.

In this section, we'll see how to work with multiple states, and how to compose state machines to make higher level state machines. We'll begin by seeing how to add a login failure counter to `getData`.

Working with multiple resources

To see how to work with multiple resources, we'll modify `getData` so that it loops, and counts the total number of times the user fails to log in. For example, if we write a `main` program which initialises the count to zero, a session might run as follows:

```
*LoginCount> :exec main
Enter password: Mornington Crescent
Secret is: "Secret Data"
Enter password: Dollis Hill
Failure
Number of failures: 1
Enter password: Mornington Crescent
Secret is: "Secret Data"
Enter password: Codfanglers
Failure
Number of failures: 2
...
```

We'll start by adding a state resource to `getData` to keep track of the number of failures:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
```

Type checking `getData`

If you're following along in the code, you'll find that `getData` no longer compiles when you update this type. That is to be expected! For the moment, comment out the definition of `getData`. We'll come back to it shortly.

Then, we can create a `main` program which initialises the state to `0` and invokes `getData`, as follows:

```
main : IO ()
main = run (do fc <- new 0
               getData fc
               delete fc)
```

We'll start our implementation of `getData` just by adding the new argument for the failure count:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
```

```
            secret <- readSecret st
            putStrLn ("Secret is: " ++ show secret)
            logout st
            disconnect st
```

Unfortunately, this doesn't type check, because we have the wrong resources for calling `connect`. The error messages shows how the resources don't match:

```
When checking an application of function Control.ST.>>=:
    Error in state transition:
            Operation has preconditions: []
            States here are: [failcount ::: State Integer]
            Operation has postconditions: \result => [result ::: Store LoggedOut] ++ []
            Required result states here are: st2_fn
```

In other words, `connect` requires that there are *no* resources on entry, but we have *one*, the failure count! This shouldn't be a problem, though: the required resources are a *subset* of the resources we have, after all, and the additional resources (here, the failure count) are not relevant to `connect`. What we need, therefore, is a way to temporarily *hide* the additional resource.

We can achieve this with the `call` function:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
   = do st <- call connect
        ?whatNow
```

Here we've left a hole for the rest of `getData` so that you can see the effect of `call`. It has removed the unnecessary parts of the resource list for calling `connect`, then reinstated them on return. The type of `whatNow` therefore shows that we've added a new resource `st`, and still have `failcount` available:

```
  failcount : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [failcount ::: State Integer, st ::: Store LoggedOut]
                      (\result => [failcount ::: State Integer])
```

By the end of the function, `whatNow` says that we need to have finished with `st`, but still have `failcount` available. We can complete `getData` so that it works with an additional state resource by adding `call` whenever we invoke one of the operations on the data store, to reduce the list of resources:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- call connect
             OK <- call $ login st
                | BadPassword => do putStrLn "Failure"
                                    call $ disconnect st
             secret <- call $ readSecret st
             putStrLn ("Secret is: " ++ show secret)
             call $ logout st
             call $ disconnect st
```

This is a little noisy, and in fact we can remove the need for it by making `call` implicit. By default, you need to add the `call` explicitly, but if you import `Control.ST.ImplicitCall`, Idris will insert `call` where it is necessary.

```
import Control.ST.ImplicitCall
```

It's now possible to write `getData` exactly as before:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

There is a trade off here: if you import `Control.ST.ImplicitCall` then functions which use multiple resources are much easier to read, because the noise of `call` has gone. On the other hand, Idris has to work a little harder to type check your functions, and as a result it can take slightly longer, and the error messages can be less helpful.

It is instructive to see the type of `call`:

```
call : STrans m t sub new_f -> {auto res_prf : SubRes sub old} ->
       STrans m t old (\res => updateWith (new_f res) old res_prf)
```

The function being called has a list of resources `sub`, and there is an implicit proof, `SubRes sub old` that the resource list in the function being called is a subset of the overall resource list. The ordering of resources is allowed to change, although resources which appear in `old` can't appear in the `sub` list more than once (you will get a type error if you try this).

The function `updateWith` takes the *output* resources of the called function, and updates them in the current resource list. It makes an effort to preserve ordering as far as possible, although this isn't always possible if the called function does some complicated resource manipulation.

Newly created resources in called functions

If the called function creates any new resources, these will typically appear at the *end* of the resource list, due to the way `updateWith` works. You can see this in the type of `whatNow` in our incomplete definition of `getData` above.

Finally, we can update `getData` so that it loops, and keeps `failCount` updated as necessary:

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
   = do st <- call connect
        OK <- login st
           | BadPassword => do putStrLn "Failure"
                               fc <- read failcount
                               write failcount (fc + 1)
                               putStrLn ("Number of failures: " ++ show (fc + 1))
                               disconnect st
                               getData failcount
        secret <- readSecret st
        putStrLn ("Secret is: " ++ show secret)
        logout st
        disconnect st
```

```
        getData failcount
```

Note that here, we're connecting and disconnecting on every iteration. Another way to implement this would be to `connect` first, then call `getData`, and implement `getData` as follows:

```
getData : (ConsoleIO m, DataStore m) =>
          (st, failcount : Var) -> ST m () [st ::: Store {m} LoggedOut, failcount ::: State␣
↪Integer]
getData st failcount
   = do OK <- login st
          | BadPassword => do putStrLn "Failure"
                              fc <- read failcount
                              write failcount (fc + 1)
                              putStrLn ("Number of failures: " ++ show (fc + 1))
                              getData st failcount
        secret <- readSecret st
        putStrLn ("Secret is: " ++ show secret)
        logout st
        getData st failcount
```

It is important to add the explicit `{m}` in the type of `Store {m} LoggedOut` for `st`, because this gives Idris enough information to know which implementation of `DataStore` to use to find the appropriate implementation for `Store`. Otherwise, if we only write `Store LoggedOut`, there's no way to know that the `Store` is linked with the computation context `m`.

We can then `connect` and `disconnect` only once, in `main`:

```
main : IO ()
main = run (do fc <- new 0
               st <- connect
               getData st fc
               disconnect st
               delete fc)
```

By using `call`, and importing `Control.ST.ImplicitCall`, we can write programs which use multiple resources, and reduce the list of resources as necessary when calling functions which only use a subset of the overall resources.

Composite resources: Hierarchies of state machines

We've now seen how to use multiple resources in one function, which is necessary for any realistic program which manipulates state. We can think of this as "horizontal" composition: using multiple resources at once. We'll often also need "vertical" composition: implementing one resource in terms of one or more other resources.

We'll see an example of this in this section. First, we'll implement a small API for graphics, in an interface `Draw`, supporting:

- Opening a window, creating a double-buffered surface to draw on
- Drawing lines and rectangles onto a surface
- "Flipping" buffers, displaying the surface we've just drawn onto in the window
- Closing a window

Then, we'll use this API to implement a higher level API for turtle graphics, in an `interface`. This will require not only the `Draw` interface, but also a representation of the turtle state (location, direction and

pen colour).

SDL bindings

For the examples in this section, you'll need to install the (very basic!) SDL bindings for Idris, available from https://github.com/edwinb/SDL-idris. These bindings implement a small subset of the SDL API, and are for illustrative purposes only. Nevertheless, they are enough to implement small graphical programs and demonstrate the concepts of this section.

Once you've installed this package, you can start Idris with the `-p sdl` flag, for the SDL bindings, and the `-p contrib` flag, for the `Control.ST` library.

The `Draw` interface

We're going to use the Idris SDL bindings for this API, so you'll need to import `Graphics.SDL` once you've installed the bindings. We'll start by defining the `Draw` interface, which includes a data type representing a surface on which we'll draw lines and rectangles:

```
interface Draw (m : Type -> Type) where
    Surface : Type
```

We'll need to be able to create a new `Surface` by opening a window:

```
initWindow : Int -> Int -> ST m Var [add Surface]
```

However, this isn't quite right. It's possible that opening a window will fail, for example if our program is running in a termina without a windowing system available. So, somehow, `initWindow` needs to cope with the possibility of failure. We can do this by returning a `Maybe Var`, rather than a `Var`, and only adding the `Surface` on success:

```
initWindow : Int -> Int -> ST m (Maybe Var) [addIfJust Surface]
```

This uses a type level function `addIfJust`, defined in `Control.ST` which returns an `Action` that only adds a resource if the operation succeeds (that is, returns a result of the form `Just val`.

`addIfJust` and `addIfRight`

`Control.ST` defines functions for constructing new resources if an operation succeeds. As well as `addIfJust`, which adds a resource if an operation returns `Just ty`, there's also `addIfRight`:

```
addIfJust : Type -> Action (Maybe Var)
addIfRight : Type -> Action (Either a Var)
```

Each of these is implemented in terms of the following primitive action `Add`, which takes a function to construct a resource list from the result of an operation:

```
Add : (ty -> Resources) -> Action ty
```

Using this, you can create your own actions to add resources based on the result of an operation, if required. For example, `addIfJust` is implemented as follows:

```
addIfJust : Type -> Action (Maybe Var)
addIfJust ty = Add (maybe [] (\var => [var ::: ty]))
```

If we create windows, we'll also need to be able to delete them:

```
closeWindow : (win : Var) -> ST m () [remove win Surface]
```

We'll also need to respond to events such as keypresses and mouse clicks. The `Graphics.SDL` library provides an `Event` type for this, and we can `poll` for events which returns the last event which occurred, if any:

```
poll : ST m (Maybe Event) []
```

The remaining methods of `Draw` are `flip`, which flips the buffers displaying everything that we've drawn since the previous `flip`, and two methods for drawing: `filledRectangle` and `drawLine`.

```
flip : (win : Var) -> ST m () [win ::: Surface]
filledRectangle : (win : Var) -> (Int, Int) -> (Int, Int) -> Col -> ST m () [win ::: Surface]
drawLine : (win : Var) -> (Int, Int) -> (Int, Int) -> Col -> ST m () [win ::: Surface]
```

We define colours as follows, as four components (red, green, blue, alpha):

```
data Col = MkCol Int Int Int Int

black : Col
black = MkCol 0 0 0 255

red : Col
red = MkCol 255 0 0 255

green : Col
green = MkCol 0 255 0 255

-- Also blue, yellow, magenta, cyan, white, similarly...
```

If you import `Graphics.SDL`, you can implement the `Draw` interface using the SDL bindings as follows:

```
interface Draw IO where
  Surface = State SDLSurface

  initWindow x y = do Just srf <- lift (startSDL x y)
                            | pure Nothing
                      var <- new srf
                      pure (Just var)

  closeWindow win = do lift endSDL
                       delete win

  flip win = do srf <- read win
                lift (flipBuffers srf)
  poll = lift pollEvent

  filledRectangle win (x, y) (ex, ey) (MkCol r g b a)
       = do srf <- read win
            lift $ filledRect srf x y ex ey r g b a
  drawLine win (x, y) (ex, ey) (MkCol r g b a)
       = do srf <- read win
            lift $ drawLine srf x y ex ey r g b a
```

In this implementation, we've used `startSDL` to initialise a window, which, returns `Nothing` if it fails. Since the type of `initWindow` states that it adds a resource when it returns a value of the form `Just val`, we add the surface returned by `startSDL` on success, and nothing on failure. We can only successfully initialise if `startDSL` succeeds.

Now that we have an implementation of `Draw`, we can try writing some functions for drawing into a window and execute them via the SDL bindings. For example, assuming we have a surface `win` to draw onto, we can write a `render` function as follows which draws a line onto a black background:

```
render : Draw m => (win : Var) -> ST m () [win ::: Surface {m}]
render win = do filledRectangle win (0,0) (640,480) black
                drawLine win (100,100) (200,200) red
                flip win
```

The `flip win` at the end is necessary because the drawing primitives are double buffered, to prevent flicker. We draw onto one buffer, off-screen, and display the other. When we call `flip`, it displays the off-screen buffer, and creates a new off-screen buffer for drawing the next frame.

To include this in a program, we'll write a main loop which renders our image and waits for an event to indicate the user wants to close the application:

```
loop : Draw m => (win : Var) -> ST m () [win ::: Surface {m}]
loop win = do render win
              Just AppQuit <- poll
                  | _ => loop win
              pure ()
```

Finally, we can create a main program which initialises a window, if possible, then runs the main loop:

```
drawMain : (ConsoleIO m, Draw m) => ST m () []
drawMain = do Just win <- initWindow 640 480
                 | Nothing => putStrLn "Can't open window"
              loop win
              closeWindow win
```

We can try this at the REPL using `run`:

```
*Draw> :exec run drawMain
```

A higher level interface: `TurtleGraphics`

Turtle graphics involves a "turtle" moving around the screen, drawing a line as it moves with a "pen". A turtle has attributes describing its location, the direction it's facing, and the current pen colour. There are commands for moving the turtle forwards, turning through an angle, and changing the pen colour, among other things. One possible interface would be the following:

```
interface TurtleGraphics (m : Type -> Type) where
  Turtle : Type

  start : Int -> Int -> ST m (Maybe Var) [addIfJust Turtle]
  end : (t : Var) -> ST m () [Remove t Turtle]

  fd : (t : Var) -> Int -> ST m () [t ::: Turtle]
  rt : (t : Var) -> Int -> ST m () [t ::: Turtle]

  penup : (t : Var) -> ST m () [t ::: Turtle]
  pendown : (t : Var) -> ST m () [t ::: Turtle]
  col : (t : Var) -> Col -> ST m () [t ::: Turtle]

  render : (t : Var) -> ST m () [t ::: Turtle]
```

Like `Draw`, we have a command for initialising the turtle (here called `start`) which might fail if it can't create a surface for the turtle to draw on. There is also a `render` method, which is intended to render

the picture drawn so far in a window. One possible program with this interface is the following, with draws a colourful square:

```
turtle : (ConsoleIO m, TurtleGraphics m) => ST m () []
turtle = with ST do
            Just t <- start 640 480
                 | Nothing => putStr "Can't make turtle\n"
            col t yellow
            fd t 100; rt t 90
            col t green
            fd t 100; rt t 90
            col t red
            fd t 100; rt t 90
            col t blue
            fd t 100; rt t 90
            render t
            end t
```

with ST do

The purpose of `with ST do` in `turtle` is to disambiguate `(>>=)`, which could be either the version from the `Monad` interface, or the version from `ST`. Idris can work this out itself, but it takes time to try all of the possibilities, so the `with` clause can speed up type checking.

To implement the interface, we could try using `Surface` to represent the surface for the turtle to draw on:

```
implementation Draw m => TurtleGraphics m where
  Turtle = Surface {m}
```

Knowing that a `Turtle` is represented as a `Surface`, we can use the methods provided by `Draw` to implement the turtle. Unfortunately, though, this isn't quite enough. We need to store more information: in particular, the turtle has several attributes which we need to store somewhere. So, not only do we need to represent the turtle as a `Surface`, we need to store some additional state. We can achieve this using a *composite* resource.

Introducing composite resources

A *composite* resource is built up from a list of other resources, and is implemented using the following type, defined by `Control.ST`:

```
data Composite : List Type -> Type
```

If we have a composite resource, we can split it into its constituent resources, and create new variables for each of those resources, using the *split* function. For example:

```
splitComp : (comp : Var) -> ST m () [comp ::: Composite [State Int, State String]]
splitComp comp = do [int, str] <- split comp
                    ?whatNow
```

The call `split comp` extracts the `State Int` and `State String` from the composite resource `comp`, and stores them in the variables `int` and `str` respectively. If we check the type of `whatNow`, we'll see how this has affected the resource list:

```
  int : Var
  str : Var
```

```
  comp : Var
  m : Type -> Type
--------------------------------------
whatNow : STrans m () [int ::: State Int, str ::: State String, comp ::: State ()]
                      (\result => [comp ::: Composite [State Int, State String]])
```

So, we have two new resources `int` and `str`, and the type of `comp` has been updated to the unit type, so currently holds no data. This is to be expected: we've just extracted the data into individual resources after all.

Now that we've extracted the individual resources, we can manipulate them directly (say, incrementing the `Int` and adding a newline to the `String`) then rebuild the composite resource using `combine`:

```
splitComp : (comp : Var) ->
            ST m () [comp ::: Composite [State Int, State String]]
splitComp comp = do [int, str] <- split comp
                    update int (+ 1)
                    update str (++ "\n")
                    combine comp [int, str]
                    ?whatNow
```

As ever, we can check the type of `whatNow` to see the effect of `combine`:

```
  comp : Var
  int : Var
  str : Var
  m : Type -> Type
--------------------------------------
whatNow : STrans m () [comp ::: Composite [State Int, State String]]
                  (\result => [comp ::: Composite [State Int, State String]])
```

The effect of `combine`, therefore, is to take existing resources and merge them into one composite resource. Before we run `combine`, the target resource must exist (`comp` here) and must be of type `State ()`.

It is instructive to look at the types of `split` and combine to see the requirements on resource lists they work with. The type of `split` is the following:

```
split : (lbl : Var) -> {auto prf : InState lbl (Composite vars) res} ->
        STrans m (VarList vars) res (\ vs => mkRes vs ++ updateRes res prf (State ()))
```

The implicit `prf` argument says that the `lbl` being split must be a composite resource. It returns a variable list, built from the composite resource, and the `mkRes` function makes a list of resources of the appropriate types. Finally, `updateRes` updates the composite resource to have the type `State ()`.

The `combine` function does the inverse:

```
combine : (comp : Var) -> (vs : List Var) ->
          {auto prf : InState comp (State ()) res} ->
          {auto var_prf : VarsIn (comp :: vs) res} ->
          STrans m () res (const (combineVarsIn res var_prf))
```

The implicit `prf` argument here ensures that the target resource `comp` has type `State ()`. That is, we're not overwriting any other data. The implicit `var_prf` argument is similar to `SubRes` in `call`, and ensures that every variable we're using to build the composite resource really does exist in the current resource list.

We can use composite resources to implement our higher level `TurtleGraphics` API in terms of `Draw`, and any additional resources we need.

Implementing `Turtle`

Now that we've seen how to build a new resource from an existing collection, we can implement `Turtle` using a composite resource, containing the `Surface` to draw on, and individual states for the pen colour and the pen location and direction. We also have a list of lines, which describes what we'll draw onto the `Surface` when we call `render`:

```
Turtle = Composite [Surface {m}, -- surface to draw on
                    State Col,  -- pen colour
                    State (Int, Int, Int, Bool), -- pen location/direction/d
                    State (List Line)] -- lines to draw on render
```

A `Line` is defined as a start location, and end location, and a colour:

```
Line : Type
Line = ((Int, Int), (Int, Int), Col)
```

To implement `start`, which creates a new `Turtle` (or returns `Nothing`) if this is impossible, we begin by initialising the drawing surface then all of the components of the state. Finally, we combine all of these into a composite resource for the turtle:

```
start x y = do Just srf <- initWindow x y
                    | Nothing => pure Nothing
               col <- new white
               pos <- new (320, 200, 0, True)
               lines <- new []
               turtle <- new ()
               combine turtle [srf, col, pos, lines]
               pure (Just turtle)
```

To implement `end`, which needs to dispose of the turtle, we deconstruct the composite resource, close the window, then remove each individual resource. Remember that we can only `delete` a `State`, so we need to `split` the composite resource, close the drawing surface cleanly with `closeWindow`, then `delete` the states:

```
end t = do [srf, col, pos, lines] <- split t
           closeWindow srf; delete col; delete pos; delete lines; delete t
```

For the other methods, we need to `split` the resource to get each component, and `combine` into a composite resource when we're done. As an example, here's `penup`:

```
penup t = do [srf, col, pos, lines] <- split t -- Split the composite resource
             (x, y, d, _) <- read pos          -- Deconstruct the pen position
             write pos (x, y, d, False)        -- Set the pen down flag to False
             combine t [srf, col, pos, lines]  -- Recombine the components
```

The remaining operations on the turtle follow a similar pattern. See `samples/ST/Graphics/Turtle.idr` in the Idris distribution for the full details. It remains to render the image created by the turtle:

```
render t = do [srf, col, pos, lines] <- split t -- Split the composite resource
              filledRectangle srf (0, 0) (640, 480) black -- Draw a background
              drawAll srf !(read lines)          -- Render the lines drawn by the turtle
              flip srf                           -- Flip the buffers to display the image
              combine t [srf, col, pos, lines]
              Just ev <- poll
                | Nothing => render t            -- Keep going until a key is pressed
              case ev of
                   KeyUp _ => pure ()            -- Key pressed, so quit
                   _ => render t
 where drawAll : (srf : Var) -> List Line -> ST m () [srf ::: Surface {m}]
```

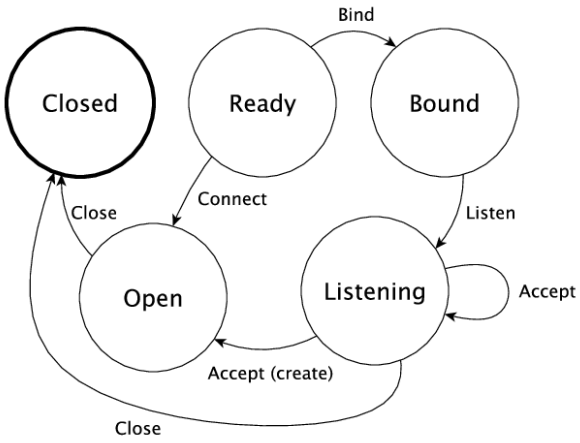
```
drawAll srf [] = pure ()
drawAll srf ((start, end, col) :: xs)
   = do drawLine srf start end col       -- Draw a line in the appropriate colour
        drawAll srf xs
```

Example: Network Socket Programming

The POSIX sockets API supports communication between processes across a network. A *socket* represents an endpoint of a network communication, and can be in one of several states:

- `Ready`, the initial state
- `Bound`, meaning that it has been bound to an address ready for incoming connections
- `Listening`, meaning that it is listening for incoming connections
- `Open`, meaning that it is ready for sending and receiving data;
- `Closed`, meaning that it is no longer active.

The following diagram shows how the operations provided by the API modify the state, where `Ready` is the initial state:



If a connection is `Open`, then we can also `send` messages to the other end of the connection, and `recv` messages from it.

The `contrib` package provides a module `Network.Socket` which provides primitives for creating sockets and sending and receiving messages. It includes the following functions:

```
bind : (sock : Socket) -> (addr : Maybe SocketAddress) -> (port : Port) -> IO Int
connect : (sock : Socket) -> (addr : SocketAddress) -> (port : Port) -> IO ResultCode
listen : (sock : Socket) -> IO Int
accept : (sock : Socket) -> IO (Either SocketError (Socket, SocketAddress))
send : (sock : Socket) -> (msg  : String) -> IO (Either SocketError ResultCode)
recv : (sock : Socket) -> (len : ByteLength) -> IO (Either SocketError (String, ResultCode))
close : Socket -> IO ()
```

These functions cover the state transitions in the diagram above, but none of them explain how the operations affect the state! It's perfectly possible, for example, to try to send a message on a socket which is not yet ready, or to try to receive a message after the socket is closed.

Using `ST`, we can provide a better API which explains exactly how each operation affects the state of a connection. In this section, we'll define a sockets API, then use it to implement an "echo" server which responds to requests from a client by echoing back a single message sent by the client.

Defining a `Sockets` interface

Rather than using `IO` for low level socket programming, we'll implement an interface using `ST` which describes precisely how each operation affects the states of sockets, and describes when sockets are created and removed. We'll begin by creating a type to describe the abstract state of a socket:

```
data SocketState = Ready | Bound | Listening | Open | Closed
```

Then, we'll begin defining an interface, starting with a `Sock` type for representing sockets, parameterised by their current state:

```
interface Sockets (m : Type -> Type) where
  Sock : SocketState -> Type
```

We create sockets using the `socket` method. The `SocketType` is defined by the sockets library, and describes whether the socket is TCP, UDP, or some other form. We'll use `Stream` for this throughout, which indicates a TCP socket.

```
socket : SocketType -> ST m (Either () Var) [addIfRight (Sock Ready)]
```

Remember that `addIfRight` adds a resource if the result of the operation is of the form `Right val`. By convention in this interface, we'll use `Either` for operations which might fail, whether or not they might carry any additional information about the error, so that we can consistently use `addIfRight` and some other type level functions.

To define a server, once we've created a socket, we need to `bind` it to a port. We can do this with the `bind` method:

```
bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
       ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Bound)]
```

Binding a socket might fail, for example if there is already a socket bound to the given port, so again it returns a value of type `Either`. The action here uses a type level function `or`, and says that:

- If `bind` fails, the socket moves to the `Sock Closed` state
- If `bind` succeeds, the socket moves to the `Sock Bound` state, as shown in the diagram above

`or` is implemented as follows:

```
or : a -> a -> Either b c -> a
or x y = either (const x) (const y)
```

So, the type of `bind` could equivalently be written as:

```
bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
       STrans m (Either () ()) [sock ::: Sock Ready]
                    (either [sock ::: Sock Closed] [sock ::: Sock Bound])
```

However, using `or` is much more concise than this, and attempts to reflect the state transition diagram as directly as possible while still capturing the possibility of failure.

Once we've bound a socket to a port, we can start listening for connections from clients:

```
listen : (sock : Var) ->
         ST m (Either () ()) [sock ::: Sock Bound :-> (Sock Closed `or` Sock Listening)]
```

A socket in the `Listening` state is ready to accept connections from individual clients:

```
accept : (sock : Var) ->
         ST m (Either () Var)
              [sock ::: Sock Listening, addIfRight (Sock Open)]
```

If there is an incoming connection from a client, `accept` adds a *new* resource to the end of the resource list (by convention, it's a good idea to add resources to the end of the list, because this works more tidily with `updateWith`, as discussed in the previous section). So, we now have *two* sockets: one continuing to listen for incoming connections, and one ready for communication with the client.

We also need methods for sending and receiving data on a socket:

```
send : (sock : Var) -> String ->
       ST m (Either () ()) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
recv : (sock : Var) ->
       ST m (Either () String) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
```

Once we've finished communicating with another machine via a socket, we'll want to `close` the connection and remove the socket:

```
close : (sock : Var) ->
        {auto prf : CloseOK st} -> ST m () [sock ::: Sock st :-> Sock Closed]
remove : (sock : Var) ->
         ST m () [Remove sock (Sock Closed)]
```

We have a predicate `CloseOK`, used by `close` in an implicit proof argument, which describes when it is okay to close a socket:

```
data CloseOK : SocketState -> Type where
     CloseOpen : CloseOK Open
     CloseListening : CloseOK Listening
```

That is, we can close a socket which is `Open`, talking to another machine, which causes the communication to terminate. We can also close a socket which is `Listening` for incoming connections, which causes the server to stop accepting requests.

In this section, we're implementing a server, but for completeness we may also want a client to connect to a server on another machine. We can do this with `connect`:

```
connect : (sock : Var) -> SocketAddress -> Port ->
          ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Open)]
```

For reference, here is the complete interface:

```
interface Sockets (m : Type -> Type) where
  Sock : SocketState -> Type
  socket : SocketType -> ST m (Either () Var) [addIfRight (Sock Ready)]
  bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
         ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Bound)]
  listen : (sock : Var) ->
           ST m (Either () ()) [sock ::: Sock Bound :-> (Sock Closed `or` Sock Listening)]
  accept : (sock : Var) ->
           ST m (Either () Var) [sock ::: Sock Listening, addIfRight (Sock Open)]
  connect : (sock : Var) -> SocketAddress -> Port ->
            ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Open)]
  close : (sock : Var) -> {auto prf : CloseOK st} ->
```

```
         ST m () [sock ::: Sock st :-> Sock Closed]
  remove : (sock : Var) -> ST m () [Remove sock (Sock Closed)]
  send : (sock : Var) -> String ->
         ST m (Either () ()) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
  recv : (sock : Var) ->
         ST m (Either () String) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
```

We'll see how to implement this shortly; mostly, the methods can be implemented in `IO` by using the raw sockets API directly. First, though, we'll see how to use the API to implement an "echo" server.

Implementing an "Echo" server with `Sockets`

At the top level, our echo server begins and ends with no resources available, and uses the `ConsoleIO` and `Sockets` interfaces:

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
```

The first thing we need to do is create a socket for binding to a port and listening for incoming connections, using `socket`. This might fail, so we'll need to deal with the case where it returns `Right sock`, where `sock` is the new socket variable, or wher it returns `Left err`:

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream
           | Left err => pure ()
     ?whatNow
```

It's a good idea to implement this kind of function interactively, step by step, using holes to see what state the overall system is in after each step. Here, we can see that after a successful call to `socket`, we have a socket available in the `Ready` state:

```
  sock : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
whatNow : STrans m () [sock ::: Sock Ready] (\result1 => [])
```

Next, we need to bind the socket to a port, and start listening for connections. Again, each of these could fail. If they do, we'll remove the socket. Failure always results in a socket in the `Closed` state, so all we can do is `remove` it:

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream         | Left err => pure ()
     Right ok <- bind sock Nothing 9442  | Left err => remove sock
     Right ok <- listen sock             | Left err => remove sock
     ?runServer
```

Finally, we have a socket which is listening for incoming connections:

```
  ok : ()
  sock : Var
  ok1 : ()
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
```

```
runServer : STrans m () [sock ::: Sock Listening]
                  (\result1 => [])
```

We'll implement this in a separate function. The type of `runServer` tells us what the type of `echoServer` must be (noting that we need to give the `m` argument to `Sock` explicitly):

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
```

We can complete the definition of `startServer` as follows:

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream        | Left err => pure ()
     Right ok <- bind sock Nothing 9442 | Left err => remove sock
     Right ok <- listen sock            | Left err => remove sock
     echoServer sock
```

In `echoServer`, we'll keep accepting requests and responding to them until something fails, at which point we'll close the sockets and return. We begin by trying to accept an incoming connection:

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
echoServer sock =
  do Right new <- accept sock | Left err => do close sock; remove sock
     ?whatNow
```

If `accept` fails, we need to close the `Listening` socket and remove it before returning, because the type of `echoServer` requires this.

As always, implementing `echoServer` incrementally means that we can check the state we're in as we develop. If `accept` succeeds, we have the existing `sock` which is still listening for connections, and a `new` socket, which is open for communication:

```
  new : Var
  sock : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
whatNow : STrans m () [sock ::: Sock Listening, new ::: Sock Open]
                      (\result1 => [])
```

To complete `echoServer`, we'll receive a message on the `new` socket, and echo it back. When we're done, we close the `new` socket, and go back to the beginning of `echoServer` to handle the next connection:

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
echoServer sock =
  do Right new <- accept sock | Left err => do close sock; remove sock
     Right msg <- recv new | Left err => do close sock; remove sock; remove ne
     Right ok <- send new ("You said " ++ msg)
           | Left err => do remove new; close sock; remove sock
     close new; remove new; echoServer sock
```

Implementing `Sockets`

To implement `Sockets` in `IO`, we'll begin by giving a concrete type for `Sock`. We can use the raw sockets API (implemented in `Network.Sockeet`) for this, and use a `Socket` stored in a `State`, no matter what abstract state the socket is in:

```
implementation Sockets IO where
  Sock _ = State Socket
```

Most of the methods can be implemented by using the raw socket API directly, returning `Left` or `Right` as appropriate. For example, we can implement `socket`, `bind` and `listen` as follows:

```
socket ty = do Right sock <- lift $ Socket.socket AF_INET ty 0
                    | Left err => pure (Left ())
               lbl <- new sock
               pure (Right lbl)
bind sock addr port = do ok <- lift $ bind !(read sock) addr port
                         if ok /= 0
                            then pure (Left ())
                            else pure (Right ())
listen sock = do ok <- lift $ listen !(read sock)
                 if ok /= 0
                    then pure (Left ())
                    else pure (Right ())
```

There is a small difficulty with `accept`, however, because when we use `new` to create a new resource for the open connection, it appears at the *start* of the resource list, not the end. We can see this by writing an incomplete definition, using `returning` to see what the resources need to be if we return `Right lbl`:

```
accept sock = do Right (conn, addr) <- lift $ accept !(read sock)
                       | Left err => pure (Left ())
                 lbl <- new conn
                 returning (Right lbl) ?fixResources
```

It's convenient for `new` to add the resource to the beginning of the list because, in general, this makes automatic proof construction with an `auto`-implicit easier for Idris. On the other hand, when we use `call` to make a smaller set of resources, `updateWith` puts newly created resources at the *end* of the list, because in general that reduces the amount of re-ordering of resources.

If we look at the type of `fixResources`, we can see what we need to do to finish `accept`:

```
  _bindApp0 : Socket
  conn : Socket
  addr : SocketAddress
  sock : Var
  lbl : Var
--------------------------------------
fixResources : STrans IO () [lbl ::: State Socket, sock ::: State Socket]
                      (\value => [sock ::: State Socket, lbl ::: State Socket])
```

The current list of resources is ordered `lbl`, `sock`, and we need them to be in the order `sock`, `lbl`. To help with this situation, `Control.ST` provides a primitive `toEnd` which moves a resource to the end of the list. We can therefore complete `accept` as follows:

```
accept sock = do Right (conn, addr) <- lift $ accept !(read sock)
                       | Left err => pure (Left ())
                 lbl <- new conn
                 returning (Right lbl) (toEnd lbl)
```

For the complete implementation of `Sockets`, take a look at `samples/ST/Net/Network.idr` in the Idris

distribution. You can also find the complete echo server there, `EchoServer.idr`. There is also a higher level network protocol, `RandServer.idr`, using a hierarchy of state machines to implement a high level network communication protocol in terms of the lower level sockets API. This also uses threading, to handle incoming requests asyncronously. You can find some more detail on threading and the random number server in the draft paper State Machines All The Way Down by Edwin Brady.

CHAPTER 4

The Effects Tutorial

A tutorial on the *Effects* package in *Idris*.

Effects and the `Control.ST` module

There is a new module in the `contrib` package, `Control.ST`, which provides the resource tracking facilities of *Effects* but with better support for creating and deleting resources, and implementing resources in terms of other resources.

Unless you have a particular reason to use *Effects* you are strongly recommended to use `Control.ST` instead. There is a tutorial available on this site for `Control.ST` with several examples (*Implementing State-aware Systems in Idris: The ST Tutorial* (page 67)).

Introduction

Pure functional languages with dependent types such as Idris support reasoning about programs directly in the type system, promising that we can *know* a program will run correctly (i.e. according to the specification in its type) simply because it compiles. Realistically, though, things are not so simple: programs have to interact with the outside world, with user input, input from a network, mutable state, and so on. In this tutorial I will introduce the library, which is included with the distribution and supports programming and reasoning with side-effecting programs, supporting mutable state, interaction with the outside world, exceptions, and verified resource management.

This tutorial assumes familiarity with pure programming in Idris, as described in Sections 1–6 of the main tutorial[1]. The examples presented are tested with Idris and can be found in the examples directory of the Idris repository.

Consider, for example, the following introductory function which illustrates the kind of properties which can be expressed in the type system:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

This function adds corresponding elements in a pair of vectors. The type guarantees that the vectors will contain only elements of type `Int`, and that the input lengths and the output length all correspond. A natural question to ask here, which is typically neglected by introductory tutorials, is "How do I turn this into a program?" That is, given some lists entered by a user, how do we get into a position to be able to apply the `vadd` function? Before doing so, we will have to:

- Read user input, either from the keyboard, a file, or some other input device.
- Check that the user inputs are valid, i.e. contain only `Int` and are the same length, and report an error if not.
- Write output

The complete program will include side-effects for I/O and error handling, before we can get to the pure core functionality. In this tutorial, we will see how Idris supports side-effects. Furthermore, we will see how we can use the dependent type system to *reason* about stateful and side-effecting programs. We will return to this specific example later.

Hello world

To give an idea of how programs with effects look, here is the ubiquitous "Hello world" program, written using the `Effects` library:

```
module Main

import Effects
import Effect.StdIO

hello : Eff () [STDIO]
hello = putStrLn "Hello world!"

main : IO ()
main = run hello
```

As usual, the entry point is `main`. All `main` has to do is invoke the `hello` function which supports the `STDIO` effect for console I/O, and returns the unit value. All programs using the `Effects` library must `import Effects`. The details of the `Eff` type will be presented in the remainder of this tutorial.

To compile and run this program, Idris needs to be told to include the `Effects` package, using the `-p effects` flag (this flag is required for all examples in this tutorial):

```
idris hello.idr -o hello -p effects
./hello Hello world!
```

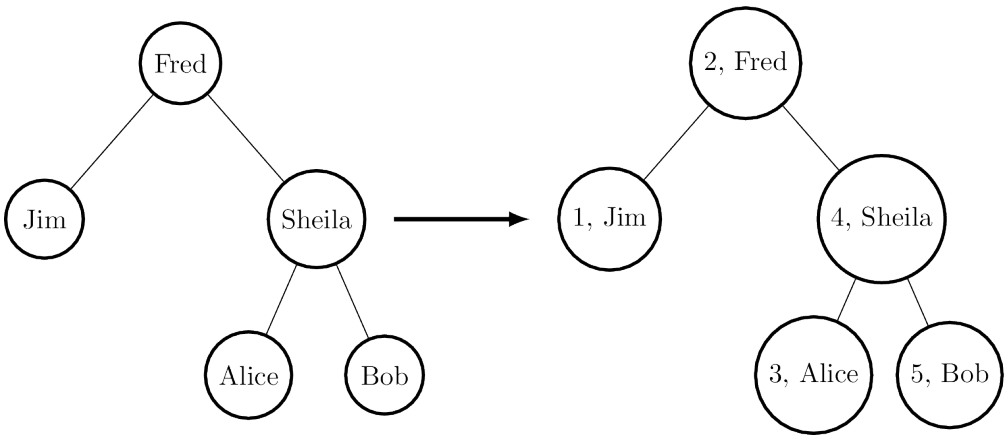
[1] You do not, however, need to know what a monad is!

Outline

The tutorial is structured as follows: first, in Section *State* (page 104), we will discuss state management, describing why it is important and introducing the `effects` library to show how it can be used to manage state. This section also gives an overview of the syntax of effectful programs. Section *Simple Effects* (page 111) then introduces a number of other effects a program may have: I/O; Exceptions; Random Numbers; and Non-determinism, giving examples for each, and an extended example combining several effects in one complete program. Section *Dependent Effects* (page 118) introduces *dependent* effects, showing how states and resources can be managed in types. Section *Creating New Effects* (page 123) shows how new effects can be implemented. Section *Example: A "Mystery Word" Guessing Game* (page 127) gives an extended example showing how to implement a "mystery word" guessing game, using effects to describe the rules of the game and ensure they are implemented accurately. References to further reading are given in Section *Further Reading* (page 132).

State

Many programs, even pure programs, can benefit from locally mutable state. For example, consider a program which tags binary tree nodes with a counter, by an inorder traversal (i.e. counting depth first, left to right). This would perform something like the following:



We can describe binary trees with the following data type `BTree` and `testTree` to represent the example input above:

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

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
                "Fred"
                (Node (Node Leaf "Alice" Leaf)
                      "Sheila"
                      (Node Leaf "Bob" Leaf))
```

Then our function to implement tagging, beginning to tag with a specific value `i`, has the following type:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
```

First attempt

Naïvely, we can implement `treeTag` by implementing a helper function which propagates a counter, returning the result of the count for each subtree:

```
treeTagAux : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTagAux i Leaf = (i, Leaf)
treeTagAux i (Node l x r)
       = let (i', l') = treeTagAux i l in
         let x' = (i', x) in
         let (i'', r') = treeTagAux (i' + 1) r in
             (i'', Node l' x' r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = snd (treeTagAux i x)
```

This gives the expected result when run at the REPL prompt:

```
*TreeTag> treeTag 1 testTree
Node (Node Leaf (1, "Jim") Leaf)
     (2, "Fred")
     (Node (Node Leaf (3, "Alice") Leaf)
           (4, "Sheila")
           (Node Leaf (5, "Bob") Leaf)) : BTree (Int, String)
```

This works as required, but there are several problems when we try to scale this to larger programs. It is error prone, because we need to ensure that state is propagated correctly to the recursive calls (i.e. passing the appropriate `i` or `i'`). It is hard to read, because the functional details are obscured by the state propagation. Perhaps most importantly, there is a common programming pattern here which should be abstracted but instead has been implemented by hand. There is local mutable state (the counter) which we have had to make explicit.

Introducing `Effects`

Idris provides a library, `Effects`[3], which captures this pattern and many others involving effectful computation[1]. An effectful program `f` has a type of the following form:

```
f : (x1 : a1) -> (x2 : a2) -> ... -> Eff t effs
```

That is, the return type gives the effects that `f` supports (`effs`, of type `List EFFECT`) and the type the computation returns `t`. So, our `treeTagAux` helper could be written with the following type:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
```

That is, `treeTagAux` has access to an integer state, because the list of available effects includes `STATE Int`. `STATE` is declared as follows in the module `Effect.State` (that is, we must `import Effect.State` to be able to use it):

```
STATE : Type -> EFFECT
```

It is an effect parameterised by a type (by convention, we write effects in all capitals). The `treeTagAux` function is an effectful program which builds a new tree tagged with `Int`s, and is implemented as follows:

[3] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 http://dl.acm.org/citation.cfm?doid=2544174.2500581

[1] The earlier paper[3] describes the essential implementation details, although the library presented there is an earlier version which is less powerful than that presented in this tutorial.

```
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r)
    = do l' <- treeTagAux l
         i <- get
         put (i + 1)
         r' <- treeTagAux r
         pure (Node l' (i, x) r')
```

There are several remarks to be made about this implementation. Essentially, it hides the state, which can be accessed using `get` and updated using `put`, but it introduces several new features. Specifically, it uses `do`-notation, binding variables with `<-`, and a `pure` function. There is much to be said about these features, but for our purposes, it suffices to know the following:

- `do` blocks allow effectful operations to be sequenced.
- **`x <- e` binds the result of an effectful operation `e` to a** variable `x`. For example, in the above code, `treeTagAux l` is an effectful operation returning `BTree (Int, a)`, so `l'` has type `BTree (Int, a)`.
- **`pure e` turns a pure value `e` into the result of an effectful** operation.

The `get` and `put` functions read and write a state `t`, assuming that the `STATE t` effect is available. They have the following types, polymorphic in the state `t` they manage:

```
get :      Eff t [STATE t]
put : t -> Eff () [STATE t]
```

A program in `Eff` can call any other function in `Eff` provided that the calling function supports at least the effects required by the called function. In this case, it is valid for `treeTagAux` to call both `get` and `put` because all three functions support the `STATE Int` effect.

Programs in `Eff` are run in some underlying *computation context*, using the `run` or `runPure` function. Using `runPure`, which runs an effectful program in the identity context, we can write the `treeTag` function as follows, using `put` to initialise the state:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i
                          treeTagAux x)
```

We could also run the program in an impure context such as `IO`, without changing the definition of `treeTagAux`, by using `run` instead of `runPure`:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
...

treeTag : (i : Int) -> BTree a -> IO (BTree (Int, a))
treeTag i x = run (do put i
                      treeTagAux x)
```

Note that the definition of `treeTagAux` is exactly as before. For reference, this complete program (including a `main` to run it) is shown in Listing [introprog].

```
module Main

import Effects
import Effect.State

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

```
Show a => Show (BTree a) where
    show Leaf = "[]"
    show (Node l x r) = "[" ++ show l ++ " "
                            ++ show x ++ " "
                            ++ show r ++ "]"

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
                "Fred"
                (Node (Node Leaf "Alice" Leaf)
                      "Sheila"
                      (Node Leaf "Bob" Leaf))

treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r) = do l' <- treeTagAux l
                             i <- get
                             put (i + 1)
                             r' <- treeTagAux r
                             pure (Node l' (i, x) r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i; treeTagAux x)

main : IO ()
main = print (treeTag 1 testTree)
```

Effects and Resources

Each effect is associated with a *resource*, which is initialised before an effectful program can be run. For example, in the case of `STATE Int` the corresponding resource is the integer state itself. The types of `runPure` and `run` show this (slightly simplified here for illustrative purposes):

```
runPure : {env : Env id xs} -> Eff a xs -> a
run : Applicative m => {env : Env m xs} -> Eff a xs -> m a
```

The `env` argument is implicit, and initialised automatically where possible using default values given by implementations of the following interface:

```
interface Default a where
    default : a
```

Implementations of `Default` are defined for all primitive types, and many library types such as `List`, `Vect`, `Maybe`, pairs, etc. However, where no default value exists for a resource type (for example, you may want a `STATE` type for which there is no `Default` implementation) the resource environment can be given explicitly using one of the following functions:

```
runPureInit : Env id xs -> Eff a xs -> a
runInit : Applicative m => Env m xs -> Eff a xs -> m a
```

To be well-typed, the environment must contain resources corresponding exactly to the effects in `xs`. For example, we could also have implemented `treeTag` by initialising the state as follows:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPureInit [i] (treeTagAux x)
```

Labelled Effects

What if we have more than one state, especially more than one state of the same type? How would `get` and `put` know which state they should be referring to? For example, how could we extend the tree tagging example such that it additionally counts the number of leaves in the tree? One possibility would be to change the state so that it captured both of these values, e.g.:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE (Int, Int)]
```

Doing this, however, ties the two states together throughout (as well as not indicating which integer is which). It would be nice to be able to call effectful programs which guaranteed only to access one of the states, for example. In a larger application, this becomes particularly important.

The library therefore allows effects in general to be *labelled* so that they can be referred to explicitly by a particular name. This allows multiple effects of the same type to be included. We can count leaves and update the tag separately, by labelling them as follows:

```
treeTagAux : BTree a ->  Eff (BTree (Int, a))
                             ['Tag ::: STATE Int,
                              'Leaves ::: STATE Int]
```

The `:::` operator allows an arbitrary label to be given to an effect. This label can be any type—it is simply used to identify an effect uniquely. Here, we have used a symbol type. In general `'name` introduces a new symbol, the only purpose of which is to disambiguate values[2].

When an effect is labelled, its operations are also labelled using the `:-` operator. In this way, we can say explicitly which state we mean when using `get` and `put`. The tree tagging program which also counts leaves can be written as follows:

```
treeTagAux Leaf = do
    'Leaves :- update (+1)
    pure Leaf
treeTagAux (Node l x r) = do
    l' <- treeTagAux l
    i <- 'Tag :- get
    'Tag :- put (i + 1)
    r' <- treeTagAux r
    pure (Node l' (i, x) r')
```

The `update` function here is a combination of `get` and `put`, applying a function to the current state.

```
update : (x -> x) -> Eff () [STATE x]
```

Finally, our top level `treeTag` function now returns a pair of the number of leaves, and the new tree. Resources for labelled effects are initialised using the `:=` operator (reminiscent of assignment in an imperative language):

```
treeTag : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTag i x = runPureInit ['Tag := i, 'Leaves := 0]
                    (do x' <- treeTagAux x
                        leaves <- 'Leaves :- get
                        pure (leaves, x'))
```

To summarise, we have:

- `:::` to convert an effect to a labelled effect.
- `:-` to convert an effectful operation to a labelled effectful operation.

[2] In practice, `'name` simply introduces a new empty type

- `:=` to initialise a resource for a labelled effect.

Or, more formally with their types (slightly simplified to account only for the situation where available effects are not updated):

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> Eff a [x] -> Eff a [l ::: x]
(:=)  : (l : lbl) -> res -> LRes l res
```

Here, `LRes` is simply the resource type associated with a labelled effect. Note that labels are polymorphic in the label type `lbl`. Hence, a label can be anything—a string, an integer, a type, etc.

!-notation

In many cases, using `do`-notation can make programs unnecessarily verbose, particularly in cases where the value bound is used once, immediately. The following program returns the length of the `String` stored in the state, for example:

```
stateLength : Eff Nat [STATE String]
stateLength = do x <- get
                 pure (length x)
```

This seems unnecessarily verbose, and it would be nice to program in a more direct style in these cases. provides `!`-notation to help with this. The above program can be written instead as:

```
stateLength : Eff Nat [STATE String]
stateLength = pure (length !get)
```

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:

```
(!) : Eff a xs -> 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 effectful.

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'
```

The Type `Eff`

Underneath, `Eff` is an overloaded function which translates to an underlying type `EffM`:

```
EffM : (m : Type -> Type) -> (t : Type)
       -> (List EFFECT)
       -> (t -> List EFFECT) -> Type
```

This is more general than the types we have been writing so far. It is parameterised over an underlying computation context `m`, a result type `t` as we have already seen, as well as a `List EFFECT` and a function type `t -> List EFFECT`.

These additional parameters are the list of *input* effects, and a list of *output* effects, computed from the result of an effectful operation. That is: running an effectful program can change the set of effects available! This is a particularly powerful idea, and we will see its consequences in more detail later. Some examples of operations which can change the set of available effects are:

- Updating a state containing a dependent type (for example adding an element to a vector).
- Opening a file for reading is an effect, but whether the file really *is* open afterwards depends on whether the file was successfully opened.
- Closing a file means that reading from the file should no longer be possible.

While powerful, this can make uses of the `EffM` type hard to read. Therefore the library provides an overloaded function `Eff` There are the following three versions:

```
SimpleEff.Eff : (t : Type) -> (input_effs : List EFFECT) -> Type
TransEff.Eff  : (t : Type) -> (input_effs : List EFFECT) ->
                              (output_effs : List EFFECT) -> Type
DepEff.Eff    : (t : Type) -> (input_effs : List EFFECT) ->
                              (output_effs_fn : x -> List EFFECT) -> Type
```

So far, we have used only the first version, `SimpleEff.Eff`, which is defined as follows:

```
Eff : (x : Type) -> (es : List EFFECT) -> Type
Eff x es = {m : Type -> Type} -> EffM m x es (\v => es)
```

i.e. the set of effects remains the same on output. This suffices for the `STATE` example we have seen so far, and for many useful side-effecting programs. We could also have written `treeTagAux` with the expanded type:

```
treeTagAux : BTree a ->
             EffM m (BTree (Int, a)) [STATE Int] (\x => [STATE Int])
```

Later, we will see programs which update effects:

```
Eff a xs xs'
```

which is expanded to

```
EffM m a xs (\_ => xs')
```

i.e. the set of effects is updated to `xs'` (think of a transition in a state machine). There is, for example, a version of `put` which updates the type of the state:

```
putM : y -> Eff () [STATE x] [STATE y]
```

Also, we have:

```
Eff t xs (\res => xs')
```

which is expanded to

```
EffM m t xs (\res => xs')
```

i.e. the set of effects is updated according to the result of the operation `res`, of type `t`.

Parameterising `EffM` over an underlying computation context allows us to write effectful programs which are specific to one context, and in some cases to write programs which *extend* the list of effects available using the `new` function, though this is beyond the scope of this tutorial.

Simple Effects

So far we have seen how to write programs with locally mutable state using the `STATE` effect. To recap, we have the definitions below in a module `Effect.State`

```
module Effect.State

STATE : Type -> EFFECT

get    :             Eff x  [STATE x]
put    : x ->        Eff () [STATE x]
putM   : y ->        Eff () [STATE x] [STATE y]
update : (x -> x) -> Eff () [STATE x]

Handler State m where { ... }
```

The last line, `Handler State m where { ... }`, means that the `STATE` effect is usable in any computation context `m`. That is, a program which uses this effect and returns something of type `a` can be evaluated to something of type `m a` using `run`, for any `m`. The lower case `State` is a data type describing the operations which make up the `STATE` effect itself—we will go into more detail about this in Section *Creating New Effects* (page 123).

In this section, we will introduce some other supported effects, allowing console I/O, exceptions, random number generation and non-deterministic programming. For each effect we introduce, we will begin with a summary of the effect, its supported operations, and the contexts in which it may be used, like that above for `STATE`, and go on to present some simple examples. At the end, we will see some examples of programs which combine multiple effects.

All of the effects in the library, including those described in this section, are summarised in Appendix *Effects Summary* (page 132).

Console I/O

Console I/O is supported with the `STDIO` effect, which allows reading and writing characters and strings to and from standard input and standard output. Notice that there is a constraint here on the computation context `m`, because it only makes sense to support console I/O operations in a context where we can perform (or at the very least simulate) console I/O:

```
module Effect.StdIO

STDIO : EFFECT

putChar  : Char ->   Eff () [STDIO]
putStr   : String -> Eff () [STDIO]
putStrLn : String -> Eff () [STDIO]

getStr   :           Eff String [STDIO]
getChar  :           Eff Char [STDIO]

Handler StdIO IO where { ... }
Handler StdIO (IOExcept a) where { ... }
```

Examples

A program which reads the user's name, then says hello, can be written as follows:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
           x <- getStr
           putStrLn ("Hello " ++ trim x ++ "!")
```

We use `trim` here to remove the trailing newline from the input. The resource associated with `STDIO` is simply the empty tuple, which has a default value `()`, so we can run this as follows:

```
main : IO ()
main = run hello
```

In `hello` we could also use `!`-notation instead of `x <- getStr`, since we only use the string that is read once:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
```

More interestingly, we can combine multiple effects in one program. For example, we can loop, counting the number of people we've said hello to:

```
hello : Eff () [STATE Int, STDIO]
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
           update (+1)
           putStrLn ("I've said hello to: " ++ show !get ++ " people")
           hello
```

The list of effects given in `hello` means that the function can call `get` and `put` on an integer state, and any functions which read and write from the console. To run this, `main` does not need to be changed.

Aside: Resource Types

To find out the resource type of an effect, if necessary (for example if we want to initialise a resource explicitly with `runInit` rather than using a default value with `run`) we can run the `resourceType` function at the REPL:

```
*ConsoleIO> resourceType STDIO
() : Type
*ConsoleIO> resourceType (STATE Int)
Int : Type
```

Exceptions

The `EXCEPTION` effect is declared in module `Effect.Exception`. This allows programs to exit immediately with an error, or errors to be handled more generally:

```
module Effect.Exception

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]
```

```
Handler (Exception a) Maybe where { ... }
Handler (Exception a) List where { ... }
Handler (Exception a) (Either a) where { ... }
Handler (Exception a) (IOExcept a) where { ... }
Show a => Handler (Exception a) IO where { ... }
```

Example

Suppose we have a `String` which is expected to represent an integer in the range `0` to `n`. We can write a function `parseNumber` which returns an `Int` if parsing the string returns a number in the appropriate range, or throws an exception otherwise. Exceptions are parameterised by an error type:

```
data Error = NotANumber | OutOfRange

parseNumber : Int -> String -> Eff Int [EXCEPTION Error]
parseNumber num str
   = if all isDigit (unpack str)
        then let x = cast str in
             if (x >=0 && x <= num)
                then pure x
                else raise OutOfRange
        else raise NotANumber
```

Programs which support the `EXCEPTION` effect can be run in any context which has some way of throwing errors, for example, we can run `parseNumber` in the `Either Error` context. It returns a value of the form `Right x` if successful:

```
*Exception> the (Either Error Int) $ run (parseNumber 42 "20")
Right 20 : Either Error Int
```

Or `Left e` on failure, carrying the appropriate exception:

```
*Exception> the (Either Error Int) $ run (parseNumber 42 "50")
Left OutOfRange : Either Error Int

*Exception> the (Either Error Int) $ run (parseNumber 42 "twenty")
Left NotANumber : Either Error Int
```

In fact, we can do a little bit better with `parseNumber`, and have it return a *proof* that the integer is in the required range along with the integer itself. One way to do this is define a type of bounded integers, `Bounded`:

```
Bounded : Int -> Type
Bounded x = (n : Int ** So (n >= 0 && n <= x))
```

Recall that `So` is parameterised by a `Bool`, and only `So True` is inhabited. We can use `choose` to construct such a value from the result of a dynamic check:

```
data So : Bool -> Type = Oh : So True

choose : (b : Bool) -> Either (So b) (So (not b))
```

We then write `parseNumber` using `choose` rather than an `if/then/else` construct, passing the proof it returns on success as the boundedness proof:

```
parseNumber : (x : Int) -> String -> Eff (Bounded x) [EXCEPTION Error]
parseNumber x str
   = if all isDigit (unpack str)
        then let num = cast str in
             case choose (num >=0 && num <= x) of
                  Left p => pure (num ** p)
                  Right p => raise OutOfRange
        else raise NotANumber
```

Random Numbers

Random number generation is also implemented by the library, in module `Effect.Random`:

```
module Effect.Random

RND : EFFECT

srand  : Integer ->            Eff () [RND]
rndInt : Integer -> Integer -> Eff Integer [RND]
rndFin : (k : Nat) ->          Eff (Fin (S k)) [RND]

Handler Random m where { ... }
```

Random number generation is considered side-effecting because its implementation generally relies on some external source of randomness. The default implementation here relies on an integer *seed*, which can be set with `srand`. A specific seed will lead to a predictable, repeatable sequence of random numbers. There are two functions which produce a random number:

- **`rndInt`, which returns a random integer between the given lower** and upper bounds.
- **`rndFin`, which returns a random element of a finite set** (essentially a number with an upper bound given in its type).

Example

We can use the `RND` effect to implement a simple guessing game. The `guess` function, given a target number, will repeatedly ask the user for a guess, and state whether the guess is too high, too low, or correct:

```
guess : Int -> Eff () [STDIO]
```

For reference, the code for `guess` is given below:

```
guess : Int -> Eff () [STDIO]
guess target
    = do putStr "Guess: "
         case run {m=Maybe} (parseNumber 100 (trim !getStr)) of
              Nothing => do putStrLn "Invalid input"
                            guess target
              Just (v ** _) =>
                        case compare v target of
                            LT => do putStrLn "Too low"
                                     guess target
                            EQ => putStrLn "You win!"
                            GT => do putStrLn "Too high"
                                     guess target
```

Note that we use `parseNumber` as defined previously to read user input, but we don't need to list the `EXCEPTION` effect because we use a nested `run` to invoke `parseNumber`, independently of the calling effectful program.

To invoke this, we pick a random number within the range 0–100, having set up the random number generator with a seed, then run `guess`:

```
game : Eff () [RND, STDIO]
game = do srand 123456789
          guess (fromInteger !(rndInt 0 100))

main : IO ()
main = run game
```

If no seed is given, it is set to the `default` value. For a less predictable game, some better source of randomness would be required, for example taking an initial seed from the system time. To see how to do this, see the `SYSTEM` effect described in *Effects Summary* (page 132).

Non-determinism

The listing below gives the definition of the non-determinism effect, which allows a program to choose a value non-deterministically from a list of possibilities in such a way that the entire computation succeeds:

```
import Effects
import Effect.Select

SELECT : EFFECT

select : List a -> Eff a [SELECT]

Handler Selection Maybe where { ... }
Handler Selection List where { ... }
```

Example

The `SELECT` effect can be used to solve constraint problems, such as finding Pythagorean triples. The idea is to use `select` to give a set of candidate values, then throw an exception for any combination of values which does not satisfy the constraint:

```
triple : Int -> Eff (Int, Int, Int) [SELECT, EXCEPTION String]
triple max = do z <- select [1..max]
                y <- select [1..z]
                x <- select [1..y]
                if (x * x + y * y == z * z)
                   then pure (x, y, z)
                   else raise "No triple"
```

This program chooses a value for `z` between `1` and `max`, then values for `y` and `x`. In operation, after a `select`, the program executes the rest of the `do`-block for every possible assignment, effectively searching depth-first. If the list is empty (or an exception is thrown) execution fails.

There are handlers defined for `Maybe` and `List` contexts, i.e. contexts which can capture failure. Depending on the context `m`, `triple` will either return the first triple it finds (if in `Maybe` context) or all triples in the range (if in `List` context). We can try this as follows:

```
main : IO ()
main = do print $ the (Maybe _) $ run (triple 100)
```

```
          print $ the (List _) $ run (triple 100)
```

vadd revisited

We now return to the vadd program from the introduction. Recall the definition:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

Using , we can set up a program so that it reads input from a user, checks that the input is valid (i.e both vectors contain integers, and are the same length) and if so, pass it on to vadd. First, we write a wrapper for vadd which checks the lengths and throw an exception if they are not equal. We can do this for input vectors of length n and m by matching on the implicit arguments n and m and using decEq to produce a proof of their equality, if they are equal:

```
vadd_check : Vect n Int -> Vect m Int ->
             Eff (Vect m Int) [EXCEPTION String]
vadd_check {n} {m} xs ys with (decEq n m)
  vadd_check {n} {m=n} xs ys | (Yes Refl) = pure (vadd xs ys)
  vadd_check {n} {m}   xs ys | (No contra) = raise "Length mismatch"
```

To read a vector from the console, we implement a function of the following type:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
```

This returns a dependent pair of a length, and a vector of that length, because we cannot know in advance how many integers the user is going to input. We can use -1 to indicate the end of input:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
read_vec = do putStr "Number (-1 when done): "
              case run (parseNumber (trim !getStr)) of
                   Nothing => do putStrLn "Input error"
                                 read_vec
                   Just v => if (v /= -1)
                                then do (_ ** xs) <- read_vec
                                        pure (_ ** v :: xs)
                                else pure (_ ** [])
  where
    parseNumber : String -> Eff Int [EXCEPTION String]
    parseNumber str
      = if all (\x => isDigit x || x == '-') (unpack str)
           then pure (cast str)
           else raise "Not a number"
```

This uses a variation on parseNumber which does not require a number to be within range.

Finally, we write a program which reads two vectors and prints the result of pairwise addition of them, throwing an exception if the inputs are of differing lengths:

```
do_vadd : Eff () [STDIO, EXCEPTION String]
do_vadd = do putStrLn "Vector 1"
             (_ ** xs) <- read_vec
             putStrLn "Vector 2"
             (_ ** ys) <- read_vec
             putStrLn (show !(vadd_check xs ys))
```

By having explicit lengths in the type, we can be sure that vadd is only being used where the lengths of

inputs are guaranteed to be equal. This does not stop us reading vectors from user input, but it does require that the lengths are checked and any discrepancy is dealt with gracefully.

Example: An Expression Calculator

To show how these effects can fit together, let us consider an evaluator for a simple expression language, with addition and integer values.

```
data Expr = Val Integer
          | Add Expr Expr
```

An evaluator for this language always returns an `Integer`, and there are no situations in which it can fail!

```
eval : Expr -> Integer
eval (Val x) = x
eval (Add l r) = eval l + eval r
```

If we add variables, however, things get more interesting. The evaluator will need to be able to access the values stored in variables, and variables may be undefined.

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
```

To start, we will change the type of `eval` so that it is effectful, and supports an exception effect for throwing errors, and a state containing a mapping from variable names (as `String`) to their values:

```
Env : Type
Env = List (String, Integer)

eval : Expr -> Eff Integer [EXCEPTION String, STATE Env]
eval (Val x)   = pure x
eval (Add l r) = pure $ !(eval l) + !(eval r)
```

Note that we are using `!`-notation to avoid having to bind subexpressions in a `do` block. Next, we add a case for evaluating `Var`:

```
eval (Var x) = case lookup x !get of
                    Nothing => raise $ "No such variable " ++ x
                    Just val => pure val
```

This retrieves the state (with `get`, supported by the `STATE Env` effect) and raises an exception if the variable is not in the environment (with `raise`, supported by the `EXCEPTION String` effect).

To run the evaluator on a particular expression in a particular environment of names and their values, we can write a function which sets the state then invokes `eval`:

```
runEval : List (String, Integer) -> Expr -> Maybe Integer
runEval args expr = run (eval' expr)
  where eval' : Expr -> Eff Integer [EXCEPTION String, STATE Env]
        eval' e = do put args
                     eval e
```

We have picked `Maybe` as a computation context here; it needs to be a context which is available for every effect supported by `eval`. In particular, because we have exceptions, it needs to be a context which supports exceptions. Alternatively, `Either String` or `IO` would be fine, for example.

What if we want to extend the evaluator further, with random number generation? To achieve this, we add a new constructor to `Expr`, which gives a random number up to a maximum value:

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
          | Random Integer
```

Then, we need to deal with the new case, making sure that we extend the list of events to include `RND`. It doesn't matter where `RND` appears in the list, as long as it is present:

```
eval : Expr -> Eff Integer [EXCEPTION String, RND, STATE Env]

eval (Random upper) = rndInt 0 upper
```

For test purposes, we might also want to print the random number which has been generated:

```
eval (Random upper) = do val <- rndInt 0 upper
                         putStrLn (show val)
                         pure val
```

If we try this without extending the effects list, we would see an error something like the following:

```
Expr.idr:28:6:When elaborating right hand side of eval:
Can't solve goal
   SubList [STDIO]
           [(EXCEPTION String), RND, (STATE (List (String, Integer)))]
```

In other words, the `STDIO` effect is not available. We can correct this simply by updating the type of `eval` to include `STDIO`.

```
eval : Expr -> Eff Integer [STDIO, EXCEPTION String, RND, STATE Env]
```

Note: Using `STDIO` will restrict the number of contexts in which `eval` can be `run` to those which support `STDIO`, such as `IO`. Once effect lists get longer, it can be a good idea instead to encapsulate sets of effects in a type synonym. This is achieved as follows, simply by defining a function which computes a type, since types are first class in Idris:

```
EvalEff : Type -> Type
EvalEff t = Eff t [STDIO, EXCEPTION String, RND, STATE Env]

eval : Expr -> EvalEff Integer
```

Dependent Effects

In the programs we have seen so far, the available effects have remained constant. Sometimes, however, an operation can *change* the available effects. The simplest example occurs when we have a state with a dependent type—adding an element to a vector also changes its type, for example, since its length is explicit in the type. In this section, we will see how the library supports this. Firstly, we will see how states with dependent types can be implemented. Secondly, we will see how the effects can depend on the *result* of an effectful operation. Finally, we will see how this can be used to implement a type-safe and resource-safe protocol for file management.

Dependent States

Suppose we have a function which reads input from the console, converts it to an integer, and adds it to a list which is stored in a `STATE`. It might look something like the following:

```
readInt : Eff () [STATE (List Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

But what if, instead of a list of integers, we would like to store a `Vect`, maintaining the length in the type?

```
readInt : Eff () [STATE (Vect n Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

This will not type check! Although the vector has length `n` on entry to `readInt`, it has length `S n` on exit. The library allows us to express this as follows:

```
readInt : Eff ()[STATE (Vect n Int), STDIO]
                [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             putM (cast x :: !get)
```

The type `Eff a xs xs'` means that the operation begins with effects `xs` available, and ends with effects `xs'` available. We have used `putM` to update the state, where the `M` suffix indicates that the *type* is being updated as well as the value. It has the following type:

```
putM : y -> Eff () [STATE x] [STATE y]
```

Result-dependent Effects

Often, whether a state is updated could depend on the success or otherwise of an operation. In our `readInt` example, we might wish to update the vector only if the input is a valid integer (i.e. all digits). As a first attempt, we could try the following, returning a `Bool` which indicates success:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
                   [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pure False
                  True => do putM (cast x :: !get)
                             pure True
```

Unfortunately, this will not type check because the vector does not get extended in both branches of the `case`!

```
MutState.idr:18:19:When elaborating right hand side of Main.case
block in readInt:
Unifying n and S n would lead to infinite value
```

Clearly, the size of the resulting vector depends on whether or not the value read from the user was valid. We can express this in the type:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
           (\ok => if ok then [STATE (Vect (S n) Int), STDIO]
                         else [STATE (Vect n Int), STDIO])
```

```
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pureM False
                  True => do putM (cast x :: !get)
                             pureM True
```

Using `pureM` rather than `pure` allows the output effects to be calculated from the value given. Its type is:

```
pureM : (val : a) -> EffM m a (xs val) xs
```

When using `readInt`, we will have to check its return value in order to know what the new set of effects is. For example, to read a set number of values into a vector, we could write the following:

```
readN : (n : Nat) ->
        Eff () [STATE (Vect m Int), STDIO]
               [STATE (Vect (n + m) Int), STDIO]
readN Z = pure ()
readN {m} (S k) = case !readInt of
                      True => rewrite plusSuccRightSucc k m in readN k
                      False => readN (S k)
```

The `case` analysis on the result of `readInt` means that we know in each branch whether reading the integer succeeded, and therefore how many values still need to be read into the vector. What this means in practice is that the type system has verified that a necessary dynamic check (i.e. whether reading a value succeeded) has indeed been done.

Note: Only `case` will work here. We cannot use `if/then/else` because the `then` and `else` branches must have the same type. The `case` construct, however, abstracts over the value being inspected in the type of each branch.

File Management

A practical use for dependent effects is in specifying resource usage protocols and verifying that they are executed correctly. For example, file management follows a resource usage protocol with the following (informally specified) requirements:

- It is necessary to open a file for reading before reading it
- Opening may fail, so the programmer should check whether opening was successful
- A file which is open for reading must not be written to, and vice versa
- When finished, an open file handle should be closed
- When a file is closed, its handle should no longer be used

These requirements can be expressed formally in , by creating a `FILE_IO` effect parameterised over a file handle state, which is either empty, open for reading, or open for writing. The `FILE_IO` effect's definition is given below. Note that this effect is mainly for illustrative purposes—typically we would also like to support random access files and better reporting of error conditions.

```
module Effect.File

import Effects
import Control.IOExcept
```

```
FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine  : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

Handler FileIO IO where { ... }
```

In particular, consider the type of `open`:

```
open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
```

This returns a `Bool` which indicates whether opening the file was successful. The resulting state depends on whether the operation was successful; if so, we have a file handle open for the stated purpose, and if not, we have no file handle. By `case` analysis on the result, we continue the protocol accordingly.

```
readFile : Eff (List String) [FILE_IO (OpenFile Read)]
readFile = readAcc [] where
    readAcc : List String -> Eff (List String) [FILE_IO (OpenFile Read)]
    readAcc acc = if (not !eof)
                     then readAcc (!readLine :: acc)
                     else pure (reverse acc)
```

Given a function `readFile`, above, which reads from an open file until reaching the end, we can write a program which opens a file, reads it, then displays the contents and closes it, as follows, correctly following the protocol:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
                    True => do putStrLn (show !readFile)
                               close
                    False => putStrLn ("Error!")
```

The type of `dumpFile`, with `FILE_IO ()` in its effect list, indicates that any use of the file resource will follow the protocol correctly (i.e. it both begins and ends with an empty resource). If we fail to follow the protocol correctly (perhaps by forgetting to close the file, failing to check that `open` succeeded, or opening the file for writing) then we will get a compile-time error. For example, changing `open name Read` to `open name Write` yields a compile-time error of the following form:

```
FileTest.idr:16:18:When elaborating right hand side of Main.case
block in testFile:
Can't solve goal
        SubList [(FILE_IO (OpenFile Read))]
                [(FILE_IO (OpenFile Write)), STDIO]
```

In other words: when reading a file, we need a file which is open for reading, but the effect list contains a `FILE_IO` effect carrying a file open for writing.

Pattern-matching bind

It might seem that having to test each potentially failing operation with a `case` clause could lead to ugly code, with lots of nested case blocks. Many languages support exceptions to improve this, but unfortunately exceptions may not allow completely clean resource management—for example, guaranteeing that any `open` which did succeed has a corresponding close.

Idris supports *pattern-matching* bindings, such as the following:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = do True <- open name Read
                   putStrLn (show !readFile)
                   close
```

This also has a problem: we are no longer dealing with the case where opening a file failed! The solution is to extend the pattern-matching binding syntax to give brief clauses for failing matches. Here, for example, we could write:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name  = do True <- open name Read | False => putStrLn "Error"
                    putStrLn (show !readFile)
                    close
```

This is exactly equivalent to the definition with the explicit `case`. In general, in a `do`-block, the syntax:

```
do pat <- val | <alternatives>
   p
```

is desugared to

```
do x <- val
   case x of
        pat => p
        <alternatives>
```

There can be several `alternatives`, separated by a vertical bar `|`. For example, there is a `SYSTEM` effect which supports reading command line arguments, among other things (see Appendix *Effects Summary* (page 132)). To read command line arguments, we can use `getArgs`:

```
getArgs : Eff (List String) [SYSTEM]
```

A main program can read command line arguments as follows, where in the list which is returned, the first element `prog` is the executable name and the second is an expected argument:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs
           putStrLn $ "Argument is " ++ arg
           {- ... rest of function ... -}
```

Unfortunately, this will not fail gracefully if no argument is given, or if too many arguments are given. We can use pattern matching bind alternatives to give a better (more informative) error:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs | [] => putStrLn "Can't happen!"
                                  | [prog] => putStrLn "No arguments!"
```

```
                              | _ => putStrLn "Too many arguments!"
          putStrLn $ "Argument is " ++ arg
          {- ... rest of function ... -}
```

If `getArgs` does not return something of the form `[prog, arg]` the alternative which does match is executed instead, and that value returned.

Creating New Effects

We have now seen several side-effecting operations provided by the `Effects` library, and examples of their use in Section *Simple Effects* (page 111). We have also seen how operations may *modify* the available effects by changing state in Section *Dependent Effects* (page 118). We have not, however, yet seen how these operations are implemented. In this section, we describe how a selection of the available effects are implemented, and show how new effectful operations may be provided.

State

Effects are described by *algebraic data types*, where the constructors describe the operations provided when the effect is available. Stateful operations are described as follows:

```
data State : Effect where
     Get :      State a  a (\x => a)
     Put : b -> State () a (\x => b)
```

`Effect` itself is a type synonym, giving the required type for an effect signature:

```
Effect : Type
Effect = (result : Type) ->
         (input_resource : Type) ->
         (output_resource : result -> Type) -> Type
```

Each effect is associated with a *resource*. The second argument to an effect signature is the resource type on *input* to an operation, and the third is a function which computes the resource type on *output*. Here, it means:

- `Get` takes no arguments. It has a resource of type `a`, which is not updated, and running the `Get` operation returns something of type `a`.
- `Put` takes a `b` as an argument. It has a resource of type `a` on input, which is updated to a resource of type `b`. Running the `Put` operation returns the element of the unit type.

The effects library provides an overloaded function `sig` which can make effect signatures more concise, particularly when the result has no effect on the resource type. For `State`, we can write:

```
data State : Effect where
     Get :      sig State a  a
     Put : b -> sig State () a b
```

There are four versions of `sig`, depending on whether we are interested in the resource type, and whether we are updating the resource. Idris will infer the appropriate version from usage.

```
NoResourceEffect.sig : Effect -> Type -> Type
NoUpdateEffect.sig   : Effect -> (ret : Type) ->
                                 (resource : Type) -> Type
UpdateEffect.sig     : Effect -> (ret : Type) ->
```

```
                                  (resource_in : Type) ->
                                  (resource_out : Type) -> Type
DepUpdateEffect.sig  : Effect -> (ret : Type) ->
                                  (resource_in : Type) ->
                                  (resource_out : ret -> Type) -> Type
```

In order to convert `State` (of type `Effect`) into something usable in an effects list, of type `EFFECT`, we write the following:

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

`MkEff` constructs an `EFFECT` by taking the resource type (here, the `t` which parameterises `STATE`) and the effect signature (here, `State`). For reference, `EFFECT` is declared as follows:

```
data EFFECT : Type where
     MkEff : Type -> Effect -> EFFECT
```

Recall that to run an effectful program in `Eff`, we use one of the `run` family of functions to run the program in a particular computation context `m`. For each effect, therefore, we must explain how it is executed in a particular computation context for `run` to work in that context. This is achieved with the following interface:

```
interface Handler (e : Effect) (m : Type -> Type) where
      handle : resource -> (eff : e t resource resource') ->
               ((x : t) -> resource' x -> m a) -> m a
```

We have already seen some implementation declarations in the effect summaries in Section *Simple Effects* (page 111). An implementation of `Handler e m` means that the effect declared with signature `e` can be run in computation context `m`. The `handle` function takes:

- The `resource` on input (so, the current value of the state for `State`)
- The effectful operation (either `Get` or `Put x` for `State`)
- A *continuation*, which we conventionally call `k`, and should be passed the result value of the operation, and an updated resource.

There are two reasons for taking a continuation here: firstly, this is neater because there are multiple return values (a new resource and the result of the operation); secondly, and more importantly, the continuation can be called zero or more times.

A `Handler` for `State` simply passes on the value of the state, in the case of `Get`, or passes on a new state, in the case of `Put`. It is defined the same way for all computation contexts:

```
Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n
```

This gives enough information for `Get` and `Put` to be used directly in `Eff` programs. It is tidy, however, to define top level functions in `Eff`, as follows:

```
get : Eff x [STATE x]
get = call Get

put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)
```

An implementation detail (aside): The `call` function converts an `Effect` to a function in `Eff`, given a proof that the effect is available. This proof can be constructed automatically by , since it is essentially an index into a statically known list of effects:

```
call : {e : Effect} ->
       (eff : e t a b) -> {auto prf : EffElem e a xs} ->
       Eff t xs (\v => updateResTy v xs prf eff)
```

This is the reason for the `Can't solve goal` error when an effect is not available: the implicit proof `prf` has not been solved automatically because the required effect is not in the list of effects `xs`.

Such details are not important for using the library, or even writing new effects, however.

Summary

The following listing summarises what is required to define the `STATE` effect:

```
data State : Effect where
     Get :      sig State a  a
     Put : b -> sig State () a b

STATE : Type -> EFFECT
STATE t = MkEff t State

Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n

get : Eff x [STATE x]
get = call Get

put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)
```

Console I/O

Then listing below gives the definition of the `STDIO` effect, including handlers for `IO` and `IOExcept`. We omit the definition of the top level `Eff` functions, as this merely invoke the effects `PutStr`, `GetStr`, `PutCh` and `GetCh` directly.

Note that in this case, the resource is the unit type in every case, since the handlers merely apply the `IO` equivalents of the effects directly.

```
data StdIO : Effect where
     PutStr : String -> sig StdIO ()
     GetStr : sig StdIO String
     PutCh : Char -> sig StdIO ()
     GetCh : sig StdIO Char

Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k x ()
    handle () (PutCh c)  k = do putChar c; k () ()
```

```
    handle () GetCh      k = do x <- getChar; k x ()

Handler StdIO (IOExcept a) where
    handle () (PutStr s) k = do ioe_lift $ putStr s; k () ()
    handle () GetStr     k = do x <- ioe_lift $ getLine; k x ()
    handle () (PutCh c)  k = do ioe_lift $ putChar c; k () ()
    handle () GetCh      k = do x <- ioe_lift $ getChar; k x ()

STDIO : EFFECT
STDIO = MkEff () StdIO
```

Exceptions

The listing below gives the definition of the `Exception` effect, including two of its handlers for `Maybe` and `List`. The only operation provided is `Raise`. The key point to note in the definitions of these handlers is that the continuation `k` is not used. Running `Raise` therefore means that computation stops with an error.

```
data Exception : Type -> Effect where
     Raise : a -> sig (Exception a) b

Handler (Exception a) Maybe where
     handle _ (Raise e) k = Nothing

Handler (Exception a) List where
     handle _ (Raise e) k = []

EXCEPTION : Type -> EFFECT
EXCEPTION t = MkEff () (Exception t)
```

Non-determinism

The following listing gives the definition of the `Select` effect for writing non-deterministic programs, including a handler for `List` context which returns all possible successful values, and a handler for `Maybe` context which returns the first successful value.

```
data Selection : Effect where
     Select : List a -> sig Selection a

Handler Selection Maybe where
     handle _ (Select xs) k = tryAll xs where
         tryAll [] = Nothing
         tryAll (x :: xs) = case k x () of
                                 Nothing => tryAll xs
                                 Just v => Just v

Handler Selection List where
     handle r (Select xs) k = concatMap (\x => k x r) xs

SELECT : EFFECT
SELECT = MkEff () Selection
```

Here, the continuation is called multiple times in each handler, for each value in the list of possible values. In the `List` handler, we accumulate all successful results, and in the `Maybe` handler we try the first value in the list, and try later values only if that fails.

File Management

Result-dependent effects are no different from non-dependent effects in the way they are implemented. The listing below illustrates this for the `FILE_IO` effect. The syntax for state transitions `{ x ==> {res} x' }`, where the result state `x'` is computed from the result of the operation `res`, follows that for the equivalent `Eff` programs.

```
data FileIO : Effect where
     Open : (fname: String)
            -> (m : Mode)
            -> sig FileIO Bool () (\res => case res of
                                               True => OpenFile m
                                               False => ())
     Close : sig FileIO () (OpenFile m)

     ReadLine  :           sig FileIO String (OpenFile Read)
     WriteLine : String -> sig FileIO ()     (OpenFile Write)
     EOF       :           sig FileIO Bool   (OpenFile Read)

Handler FileIO IO where
    handle () (Open fname m) k = do h <- openFile fname m
                                    if !(validFile h)
                                            then k True (FH h)
                                            else k False ()
    handle (FH h) Close      k = do closeFile h
                                    k () ()

    handle (FH h) ReadLine        k = do str <- fread h
                                         k str (FH h)
    handle (FH h) (WriteLine str) k = do fwrite h str
                                         k () (FH h)
    handle (FH h) EOF             k = do e <- feof h
                                         k e (FH h)

FILE_IO : Type -> EFFECT
FILE_IO t = MkEff t FileIO
```

Note that in the handler for `Open`, the types passed to the continuation `k` are different depending on whether the result is `True` (opening succeeded) or `False` (opening failed). This uses `validFile`, defined in the `Prelude`, to test whether a file handler refers to an open file or not.

Example: A "Mystery Word" Guessing Game

In this section, we will use the techniques and specific effects discussed in the tutorial so far to implement a larger example, a simple text-based word-guessing game. In the game, the computer chooses a word, which the player must guess letter by letter. After a limited number of wrong guesses, the player loses[1].

We will implement the game by following these steps:

1. Define the game state, in enough detail to express the rules
2. Define the rules of the game (i.e. what actions the player may take, and how these actions affect the game state)
3. Implement the rules of the game (i.e. implement state updates for each action)
4. Implement a user interface which allows a player to direct actions

[1] Readers may recognise this game by the name "Hangman".

Step 2 may be achieved by defining an effect which depends on the state defined in step 1. Then step 3 involves implementing a `Handler` for this effect. Finally, step 4 involves implementing a program in `Eff` using the newly defined effect (and any others required to implement the interface).

Step 1: Game State

First, we categorise the game states as running games (where there are a number of guesses available, and a number of letters still to guess), or non-running games (i.e. games which have not been started, or games which have been won or lost).

```
data GState = Running Nat Nat | NotRunning
```

Notice that at this stage, we say nothing about what it means to make a guess, what the word to be guessed is, how to guess letters, or any other implementation detail. We are only interested in what is necessary to describe the game rules.

We will, however, parameterise a concrete game state `Mystery` over this data:

```
data Mystery : GState -> Type
```

Step 2: Game Rules

We describe the game rules as a dependent effect, where each action has a *precondition* (i.e. what the game state must be before carrying out the action) and a *postcondition* (i.e. how the action affects the game state). Informally, these actions with the pre- and postconditions are:

Guess Guess a letter in the word.

- Precondition: The game must be running, and there must be both guesses still available, and letters still to be guessed.
- Postcondition: If the guessed letter is in the word and not yet guessed, reduce the number of letters, otherwise reduce the number of guesses.

Won Declare victory

- Precondition: The game must be running, and there must be no letters still to be guessed.
- Postcondition: The game is no longer running.

Lost Accept defeat

- Precondition: The game must be running, and there must be no guesses left.
- Postcondition: The game is no longer running.

NewWord Set a new word to be guessed

- Precondition: The game must not be running.
- Postcondition: The game is running, with 6 guesses available (the choice of 6 is somewhat arbitrary here) and the number of unique letters in the word still to be guessed.

Get Get a string representation of the game state. This is for display purposes; there are no pre- or postconditions.

We can make these rules precise by declaring them more formally in an effect signature:

```
data MysteryRules : Effect where
     Guess : (x : Char) ->
             sig MysteryRules Bool
                 (Mystery (Running (S g) (S w)))
                 (\inword => if inword
                             then Mystery (Running (S g) w)
                             else Mystery (Running g (S w)))
     Won  : sig MysteryRules () (Mystery (Running g 0))
                                (Mystery NotRunning)
     Lost : sig MysteryRules () (Mystery (Running 0 g))
                                (Mystery NotRunning)
     NewWord : (w : String) ->
               sig MysteryRules () (Mystery NotRunning) (Mystery (Running 6 (length (letters␣
↪w))))
     Get : sig MysteryRules String (Mystery h)
```

This description says nothing about how the rules are implemented. In particular, it does not specify *how* to tell whether a guessed letter was in a word, just that the result of `Guess` depends on it.

Nevertheless, we can still create an `EFFECT` from this, and use it in an `Eff` program. Implementing a `Handler` for `MysteryRules` will then allow us to play the game.

```
MYSTERY : GState -> EFFECT
MYSTERY h = MkEff (Mystery h) MysteryRules
```

Step 3: Implement Rules

To *implement* the rules, we begin by giving a concrete definition of game state:

```
data Mystery : GState -> Type where
     Init     : Mystery NotRunning
     GameWon  : (word : String) -> Mystery NotRunning
     GameLost : (word : String) -> Mystery NotRunning
     MkG      : (word : String) ->
                (guesses : Nat) ->
                (got : List Char) ->
                (missing : Vect m Char) ->
                Mystery (Running guesses m)
```

If a game is `NotRunning`, that is either because it has not yet started (`Init`) or because it is won or lost (`GameWon` and `GameLost`, each of which carry the word so that showing the game state will reveal the word to the player). Finally, `MkG` captures a running game's state, including the target word, the letters successfully guessed, and the missing letters. Using a `Vect` for the missing letters is convenient since its length is used in the type.

To initialise the state, we implement the following functions: `letters`, which returns a list of unique letters in a `String` (ignoring spaces) and `initState` which sets up an initial state considered valid as a postcondition for `NewWord`.

```
letters : String -> List Char
initState : (x : String) -> Mystery (Running 6 (length (letters x)))
```

When checking if a guess is in the vector of missing letters, it is convenient to return a *proof* that the guess is in the vector, using `isElem` below, rather than merely a `Bool`:

```
data IsElem : a -> Vect n a -> Type where
     First : IsElem x (x :: xs)
     Later : IsElem x xs -> IsElem x (y :: xs)
```

```
isElem : DecEq a => (x : a) -> (xs : Vect n a) -> Maybe (IsElem x xs)
```

The reason for returning a proof is that we can use it to remove an element from the correct position in a vector:

```
shrink : (xs : Vect (S n) a) -> IsElem x xs -> Vect n a
```

We leave the definitions of `letters`, `init`, `isElem` and `shrink` as exercises. Having implemented these, the `Handler` implementation for `MysteryRules` is surprisingly straightforward:

```
Handler MysteryRules m where
    handle (MkG w g got []) Won k = k () (GameWon w)
    handle (MkG w Z got m) Lost k = k () (GameLost w)

    handle st Get k = k (show st) st
    handle st (NewWord w) k = k () (initState w)

    handle (MkG w (S g) got m) (Guess x) k =
        case isElem x m of
             Nothing => k False (MkG w _ got m)
             (Just p) => k True (MkG w _ (x :: got) (shrink m p))
```

Each case simply involves directly updating the game state in a way which is consistent with the declared rules. In particular, in `Guess`, if the handler claims that the guessed letter is in the word (by passing `True` to `k`), there is no way to update the state in such a way that the number of missing letters or number of guesses does not follow the rules.

Step 4: Implement Interface

Having described the rules, and implemented state transitions which follow those rules as an effect handler, we can now write an interface for the game which uses the `MYSTERY` effect:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
              [MYSTERY NotRunning, STDIO]
```

The type indicates that the game must start in a running state, with some guesses available, and eventually reach a not-running state (i.e. won or lost). The only way to achieve this is by correctly following the stated rules.

Note that the type of `game` makes no assumption that there are letters to be guessed in the given word (i.e. it is `w` rather than `S w`). This is because we will be choosing a word at random from a vector of `String`, and at no point have we made it explicit that those `String` are non-empty.

Finally, we need to initialise the game by picking a word at random from a list of candidates, setting it as the target using `NewWord`, then running `game`:

```
runGame : Eff () [MYSTERY NotRunning, RND, SYSTEM, STDIO]
runGame = do srand !time
             let w = index !(rndFin _) words
             call $ NewWord w
             game
             putStrLn !(call Get)
```

We use the system time (`time` from the `SYSTEM` effect; see Appendix *Effects Summary* (page 132)) to initialise the random number generator, then pick a random `Fin` to index into a list of `words`. For example, we could initialise a word list as follows:

```
words : ?wtype
words = with Vect ["idris","agda","haskell","miranda",
         "java","javascript","fortran","basic",
         "coffeescript","rust"]

wtype = proof search
```

Note: Rather than have to explicitly declare a type with the vector's length, it is convenient to give a hole `?wtype` and let Idris's proof search mechanism find the type. This is a limited form of type inference, but very useful in practice.

A possible complete implementation of `game` is presented below:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
              [MYSTERY NotRunning, STDIO]
game {w=Z} = Won
game {w=S _}
     = do putStrLn !Get
          putStr "Enter guess: "
          let guess = trim !getStr
          case choose (not (guess == "")) of
               (Left p) => processGuess (strHead' guess p)
               (Right p) => do putStrLn "Invalid input!"
                               game
  where
    processGuess : Char -> Eff () [MYSTERY (Running (S g) (S w)), STDIO]
                                  [MYSTERY NotRunning, STDIO]
    processGuess {g} {w} c
      = case !(Main.Guess c) of
             True => do putStrLn "Good guess!"
                        case w of
                             Z => Won
                             (S k) => game
             False => do putStrLn "No, sorry"
                         case g of
                              Z => Lost
                              (S k) => game
```

Discussion

Writing the rules separately as an effect, then an implementation which uses that effect, ensures that the implementation must follow the rules. This has practical applications in more serious contexts; `MysteryRules` for example can be though of as describing a *protocol* that a game player most follow, or alternative a *precisely-typed API*.

In practice, we wouldn't really expect to write rules first then implement the game once the rules were complete. Indeed, I didn't do so when constructing this example! Rather, I wrote down a set of likely rules making any assumptions *explicit* in the state transitions for `MysteryRules`. Then, when implementing `game` at first, any incorrect assumption was caught as a type error. The following errors were caught during development:

- Not realising that allowing `NewWord` to be an arbitrary string would mean that `game` would have to deal with a zero-length word as a starting state.
- Forgetting to check whether a game was won before recursively calling `processGuess`, thus accidentally continuing a finished game.

- Accidentally checking the number of missing letters, rather than the number of remaining guesses, when checking if a game was lost.

These are, of course, simple errors, but were caught by the type checker before any testing of the game.

Further Reading

This tutorial has given an introduction to writing and reasoning about side-effecting programs in Idris, using the `Effects` library. More details about the *implementation* of the library, such as how `run` works, how handlers are invoked, etc, are given in a separate paper[1].

Some libraries and programs which use `Effects` can be found in the following places:

- https://github.com/edwinb/SDL-idris — some bindings for the SDL media library, supporting graphics in particular.
- https://github.com/edwinb/idris-demos — various demonstration programs, including several examples from this tutorial, and a "Space Invaders" game.
- https://github.com/SimonJF/IdrisNet2 — networking and socket libraries.
- https://github.com/edwinb/Protocols — a high level communication protocol description language.

The inspiration for the `Effects` library was Bauer and Pretnar's Eff language[2], which describes a language based on algebraic effects and handlers. Other recent languages and libraries have also been built on this ideas, for example[3] and[4]. The theoretical foundations are also well-studied see[5],[6],[7],[8].

Effects Summary

This appendix gives interfaces for the core effects provided by the library.

EXCEPTION

```
module Effect.Exception

import Effects
import System
```

[1] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 https://dl.acm.org/citation.cfm?doid=2544174.2500581

[2] Andrej Bauer, Matija Pretnar, Programming with algebraic effects and handlers, Journal of Logical and Algebraic Methods in Programming, Volume 84, Issue 1, January 2015, Pages 108-123, ISSN 2352-2208, http://math.andrej.com/wp-content/uploads/2012/03/eff.pdf

[3] Ben Lippmeier. 2009. Witnessing Purity, Constancy and Mutability. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09), Zhenjiang Hu (Ed.). Springer-Verlag, Berlin, Heidelberg, 95-110. DOI=10.1007/978-3-642-10672-9_9 http://link.springer.com/chapter/10.1007%2F978-3-642-10672-9_9

[4] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. SIGPLAN Not. 48, 9 (September 2013), 145-158. DOI=10.1145/2544174.2500590 https://dl.acm.org/citation.cfm?doid=2544174.2500590

[5] Martin Hyland, Gordon Plotkin, John Power, Combining effects: Sum and tensor, Theoretical Computer Science, Volume 357, Issues 1–3, 25 July 2006, Pages 70-99, ISSN 0304-3975, (https://www.sciencedirect.com/science/article/pii/S0304397506002659)

[6] Paul Blain Levy. 2004. Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2). Kluwer Academic Publishers, Norwell, MA, USA.

[7] Plotkin, Gordon, and Matija Pretnar. "Handlers of algebraic effects." Programming Languages and Systems. Springer Berlin Heidelberg, 2009. 80-94.

[8] Pretnar, Matija. "Logic and handling of algebraic effects." (2010).

```
import Control.IOExcept

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]

Handler (Exception a) Maybe where { ... }
Handler (Exception a) List where { ... }
Handler (Exception a) (Either a) where { ... }
Handler (Exception a) (IOExcept a) where { ... }
Show a => Handler (Exception a) IO where { ... }
```

FILE_IO

```
module Effect.File

import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine  : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

Handler FileIO IO where { ... }
```

RND

```
module Effect.Random

import Effects
import Data.Vect
import Data.Fin

RND : EFFECT

srand  : Integer ->            Eff m () [RND]
rndInt : Integer -> Integer -> Eff m Integer [RND]
rndFin : (k : Nat) ->          Eff m (Fin (S k)) [RND]

Handler Random m where { ... }
```

SELECT

```
module Effect.Select

import Effects

SELECT : EFFECT

select : List a -> Eff m a [SELECT]

Handler Selection Maybe where { ... }
Handler Selection List where { ... }
```

STATE

```
module Effect.State

import Effects

STATE : Type -> EFFECT

get    :             Eff m x [STATE x]
put    : x ->        Eff m () [STATE x]
putM   : y ->        Eff m () [STATE x] [STATE y]
update : (x -> x) -> Eff m () [STATE x]

Handler State m where { ... }
```

STDIO

```
module Effect.StdIO

import Effects
import Control.IOExcept

STDIO : EFFECT

putChar  : Handler StdIO m => Char ->   Eff m () [STDIO]
putStr   : Handler StdIO m => String -> Eff m () [STDIO]
putStrLn : Handler StdIO m => String -> Eff m () [STDIO]

getStr   : Handler StdIO m =>           Eff m String [STDIO]
getChar  : Handler StdIO m =>           Eff m Char [STDIO]

Handler StdIO IO where { ... }
Handler StdIO (IOExcept a) where { ... }
```

SYSTEM

```
module Effect.System

import Effects
import System
import Control.IOExcept
```

```
SYSTEM : EFFECT

getArgs : Handler System e =>           Eff e (List String) [SYSTEM]
time    : Handler System e =>           Eff e Int [SYSTEM]
getEnv  : Handler System e => String -> Eff e (Maybe String) [SYSTEM]

Handler System IO where { ... }
Handler System (IOExcept a) where { ... }
```

CHAPTER 5

Theorem Proving

A tutorial on theorem proving in Idris.

Running example: Addition of Natural Numbers

Throughout this tutorial, we will be working with the following function, defined in the Idris prelude, which defines addition on natural numbers:

```
plus : Nat -> Nat -> Nat
plus Z     m = m
plus (S k) m = S (plus k m)
```

It is defined by the above equations, meaning that we have for free the properties that adding `m` to zero always results in `m`, and that adding `m` to any non-zero number `S k` always results in `S (plus k m)`. We can see this by evaluation at the Idris REPL (i.e. the prompt, the read-eval-print loop):

```
Idris> \m => plus Z m
\m => m : Nat -> Nat

Idris> \k,m => plus (S k) m
\k => \m => S (plus k m) : Nat -> Nat -> Nat
```

Note that unlike many other language REPLs, the Idris REPL performs evaluation on *open* terms, meaning that it can reduce terms which appear inside lambda bindings, like those above. Therefore, we can introduce unknowns `k` and `m` as lambda bindings and see how `plus` reduces.

The `plus` function has a number of other useful properties, for example:

- It is *commutative*, that is for all `Nat` inputs `n` and `m`, we know that `plus n m = plus m n`.
- It is *associative*, that is for all `Nat` inputs `n`, `m` and `p`, we know that `plus n (plus m p) = plus (plus m n) p`.

We can use these properties in an Idris program, but in order to do so we must *prove* them.

Equality Proofs

Idris has a built-in propositional equality type, conceptually defined as follows:

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

Note that this must be built-in, rather than defined in the library, because `=` is a reserved operator — you cannot define this directly in your own code.

It is *propositional* equality, where the type states that any two values in different types `a` and `b` may be proposed to be equal. There is only one way to *prove* equality, however, which is by reflexivity (`Refl`).

We have a *type* for propositional equality here, and correspondingly a *program* inhabiting an instance of this type can be seen as a proof of the corresponding proposition[1]. So, trivially, we can prove that `4` equals `4`:

```
four_eq : 4 = 4
four_eq = Refl
```

However, trying to prove that `4 = 5` results in failure:

```
four_eq_five : 4 = 5
four_eq_five = Refl
```

The type `4 = 5` is a perfectly valid type, but is uninhabited, so when trying to type check this definition, Idris gives the following error:

```
When elaborating right hand side of four_eq_five:
Type mismatch between
        x = x (Type of Refl)
and
        4 = 5 (Expected type)
```

Type checking equality proofs

An important step in type checking Idris programs is *unification*, which attempts to resolve implicit arguments such as the implicit argument `x` in `Refl`. As far as our understanding of type checking proofs is concerned, it suffices to know that unifying two terms involves reducing both to normal form then trying to find an assignment to implicit arguments which will make those normal forms equal.

When type checking `Refl`, Idris requires that the type is of the form `x = x`, as we see from the type of `Refl`. In the case of `four_eq_five`, Idris will try to unify the expected type `4 = 5` with the type of `Refl`, `x = x`, notice that a solution requires that `x` be both `4` and `5`, and therefore fail.

Since type checking involves reduction to normal form, we can write the following equalities directly:

[1] This is known as the Curry-Howard correspondence.

```
twoplustwo_eq_four : 2 + 2 = 4
twoplustwo_eq_four = Refl

plus_reduces_Z : (m : Nat) -> plus Z m = m
plus_reduces_Z m = Refl

plus_reduces_Sk : (k, m : Nat) -> plus (S k) m = S (plus k m)
plus_reduces_Sk k m = Refl
```

Heterogeneous Equality

Equality in Idris is *heterogeneous*, meaning that we can even propose equalities between values in different types:

```
idris_not_php : 2 = "2"
```

Obviously, in Idris the type `2 = "2"` is uninhabited, and one might wonder why it is useful to be able to propose equalities between values in different types. However, with dependent types, such equalities can arise naturally. For example, if two vectors are equal, their lengths must be equal:

```
vect_eq_length : (xs : Vect n a) -> (ys : Vect m a) ->
                 (xs = ys) -> n = m
```

In the above declaration, `xs` and `ys` have different types because their lengths are different, but we would still like to draw a conclusion about the lengths if they happen to be equal. We can define `vect_eq_length` as follows:

```
vect_eq_length xs xs Refl = Refl
```

By matching on `Refl` for the third argument, we know that the only valid value for `ys` is `xs`, because they must be equal, and therefore their types must be equal, so the lengths must be equal.

Alternatively, we can put an underscore for the second `xs`, since there is only one value which will type check:

```
vect_eq_length xs _ Refl = Refl
```

Properties of `plus`

Using the `(=)` type, we can now state the properties of `plus` given above as Idris type declarations:

```
plus_commutes : (n, m : Nat) -> plus n m = plus m n
plus_assoc : (n, m, p : Nat) -> plus n (plus m p) = plus (plus n m) p
```

Both of these properties (and many others) are proved for natural number addition in the Idris standard library, using `(+)` from the `Num` interface rather than using `plus` directly. They have the names `plusCommutative` and `plusAssociative` respectively.

In the remainder of this tutorial, we will explore several different ways of proving `plus_commutes` (or, to put it another way, writing the function.) We will also discuss how to use such equality proofs, and see where the need for them arises in practice.

Inductive Proofs

Before embarking on proving `plus_commutes` in Idris itself, let us consider the overall structure of a proof of some property of natural numbers. Recall that they are defined recursively, as follows:

```
data Nat : Type where
     Z : Nat
     S : Nat -> Nat
```

A *total* function over natural numbers must both terminate, and cover all possible inputs. Idris checks functions for totality by checking that all inputs are covered, and that all recursive calls are on *structurally smaller* values (so recursion will always reach a base case). Recalling `plus`:

```
plus : Nat -> Nat -> Nat
plus Z     m = m
plus (S k) m = S (plus k m)
```

This is total because it covers all possible inputs (the first argument can only be `Z` or `S k` for some `k`, and the second argument `m` covers all possible `Nat`) and in the recursive call, `k` is structurally smaller than `S k` so the first argument will always reach the base case `Z` in any sequence of recursive calls.

In some sense, this resembles a mathematical proof by induction (and this is no coincidence!). For some property `P` of a natural number `x`, we can show that `P` holds for all `x` if:

- `P` holds for zero (the base case).
- Assuming that `P` holds for `k`, we can show `P` also holds for `S k` (the inductive step).

In `plus`, the property we are trying to show is somewhat trivial (for all natural numbers `x`, there is a `Nat` which need not have any relation to `x`). However, it still takes the form of a base case and an inductive step. In the base case, we show that there is a `Nat` arising from `plus n m` when `n = Z`, and in the inductive step we show that there is a `Nat` arising when `n = S k` and we know we can get a `Nat` inductively from `plus k m`. We could even write a function capturing all such inductive definitions:

```
nat_induction : (P : Nat -> Type) ->              -- Property to show
                (P Z) ->                          -- Base case
                ((k : Nat) -> P k -> P (S k)) -> -- Inductive step
                (x : Nat) ->                      -- Show for all x
                P x
nat_induction P p_Z p_S Z = p_Z
nat_induction P p_Z p_S (S k) = p_S k (nat_induction P p_Z p_S k)
```

Using `nat_induction`, we can implement an equivalent inductive version of `plus`:

```
plus_ind : Nat -> Nat -> Nat
plus_ind n m
   = nat_induction (\x => Nat)
                   m                        -- Base case, plus_ind Z m
                   (\k, k_rec => S k_rec)   -- Inductive step plus_ind (S k) m
                                            -- where k_rec = plus_ind k m
                   n
```

To prove that `plus n m = plus m n` for all natural numbers `n` and `m`, we can also use induction. Either we can fix `m` and perform induction on `n`, or vice versa. We can sketch an outline of a proof; performing induction on `n`, we have:

- Property `P` is `\x => plus x m = plus m x`.
- Show that `P` holds in the base case and inductive step:

– Base case: `P Z`, i.e.
`plus Z m = plus m Z`, which reduces to
`m = plus m Z` due to the definition of `plus`.

– Inductive step: Inductively, we know that `P k` holds for a specific, fixed `k`, i.e.
`plus k m = plus m k` (the induction hypothesis). Given this, show `P (S k)`, i.e.
`plus (S k) m = plus m (S k)`, which reduces to
`S (plus k m) = plus m (S k)`. From the induction hypothesis, we can rewrite this to
`S (plus m k) = plus m (S k)`.

To complete the proof we therefore need to show that `m = plus m Z` for all natural numbers `m`, and that `S (plus m k) = plus m (S k)` for all natural numbers `m` and `k`. Each of these can also be proved by induction, this time on `m`.

We are now ready to embark on a proof of commutativity of `plus` formally in Idris.

Pattern Matching Proofs

In this section, we will provide a proof of `plus_commutes` directly, by writing a pattern matching definition. We will use interactive editing features extensively, since it is significantly easier to produce a proof when the machine can give the types of intermediate values and construct components of the proof itself. The commands we will use are summarised below. Where we refer to commands directly, we will use the Vim version, but these commands have a direct mapping to Emacs commands.

Command	Vim binding	Emacs binding	Explanation
Check type	`\t`	`C-c C-t`	Show type of identifier or hole under the cursor.
Proof search	`\o`	`C-c C-a`	Attempt to solve hole under the cursor by applying simple proof search.
Make new definition	`\d`	`C-c C-s`	Add a template definition for the type defined under the cursor.
Make lemma	`\l`	`C-c C-e`	Add a top level function with a type which solves the hole under the cursor.
Split cases	`\c`	`C-c C-c`	Create new constructor patterns for each possible case of the variable under the cursor.

Creating a Definition

To begin, create a file `pluscomm.idr` containing the following type declaration:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
```

To create a template definition for the proof, press `\d` (or the equivalent in your editor of choice) on the line with the type declaration. You should see:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes n m = ?plus_commutes_rhs
```

To prove this by induction on `n`, as we sketched in Section sect-inductive, we begin with a case split on `n` (press `\c` with the cursor over the `n` in the definition.) You should see:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = ?plus_commutes_rhs_1
plus_commutes (S k) m = ?plus_commutes_rhs_2
```

If we inspect the types of the newly created holes, `plus_commutes_rhs_1` and `plus_commutes_rhs_2`, we see that the type of each reflects that `n` has been refined to `Z` and `S k` in each respective case. Pressing `\t` over `plus_commutes_rhs_1` shows:

```
  m : Nat
--------------------------------------
plus_commutes_rhs_1 : m = plus m 0
```

Note that `Z` renders as `0` because the pretty printer renders natural numbers as integer literals for readability. Similarly, for `plus_commutes_rhs_2`:

```
  k : Nat
  m : Nat
--------------------------------------
plus_commutes_rhs_2 : S (plus k m) = plus m (S k)
```

It is a good idea to give these slightly more meaningful names:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = ?plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

Base Case

We can create a separate lemma for the base case interactively, by pressing `\l` with the cursor over `plus_commutes_Z`. This yields:

```
plus_commutes_Z : m = plus m 0

plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

That is, the hole has been filled with a call to a top level function `plus_commutes_Z`. The argument `m` has been made implicit because it can be inferred from context when it is applied.

Unfortunately, we cannot prove this lemma directly, since `plus` is defined by matching on its *first* argument, and here `plus m 0` has a specific value for its *second argument* (in fact, the left hand side of the equality has been reduced from `plus 0 m`.) Again, we can prove this by induction, this time on `m`.

First, create a template definition with `\d`:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z = ?plus_commutes_Z_rhs
```

Since we are going to write this by induction on `m`, which is implicit, we will need to bring `m` into scope manually:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m} = ?plus_commutes_Z_rhs
```

Now, case split on `m` with `\c`:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = ?plus_commutes_Z_rhs_1
plus_commutes_Z {m = (S k)} = ?plus_commutes_Z_rhs_2
```

Checking the type of `plus_commutes_Z_rhs_1` shows the following, which is easily proved by reflection:

```
--------------------------------------
plus_commutes_Z_rhs_1 : 0 = 0
```

For such trivial proofs, we can let write the proof automatically by pressing `\o` with the cursor over `plus_commutes_Z_rhs_1`. This yields:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = ?plus_commutes_Z_rhs_2
```

For `plus_commutes_Z_rhs_2`, we are not so lucky:

```
  k : Nat
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S (plus k 0)
```

Inductively, we should know that `k = plus k 0`, and we can get access to this inductive hypothesis by making a recursive call on k, as follows:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  ?plus_commutes_Z_rhs_2
```

For `plus_commutes_Z_rhs_2`, we now see:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S (plus k 0)
```

Again, the `fromInteger 0` is merely due to `Nat` having an implementation of the `Num` interface. So we know that `k = plus k 0`, but how do we use this to update the goal to `S k = S k`?

To achieve this, Idris provides a `replace` function as part of the prelude:

```
*pluscomm> :t replace
replace : (x = y) -> P x -> P y
```

Given a proof that `x = y`, and a property `P` which holds for `x`, we can get a proof of the same property for `y`, because we know `x` and `y` must be the same. In practice, this function can be a little tricky to use because in general the implicit argument `P` can be hard to infer by unification, so Idris provides a high level syntax which calculates the property and applies `replace`:

```
rewrite prf in expr
```

If we have `prf : x = y`, and the required type for `expr` is some property of `x`, the `rewrite ... in` syntax will search for `x` in the required type of `expr` and replace it with `y`. Concretely, in our example, we can say:

```
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite rec in ?plus_commutes_Z_rhs_2
```

Checking the type of `plus_commutes_Z_rhs_2` now gives:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
  _rewrite_rule : plus k 0 = k
```

```
--------------------------------------
plus_commutes_Z_rhs_2 : S (plus k 0) = S (plus k 0)
```

Using the rewrite rule `rec` (which we can see in the context here as `_rewrite_rule`[1], the goal type has been updated with `k` replaced by `plus k 0`.

Alternatively, we could have applied the rewrite in the other direction using the `sym` function:

```
*pluscomm> :t sym
sym : (l = r) -> r = l


plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite sym rec in ?plus_commutes_Z_rhs_2
```

In this case, inspecting the type of the hole gives:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
  _rewrite_rule : k = plus k 0
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S k
```

Either way, we can use proof search (`\o`) to complete the proof, giving:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite rec in Refl
```

The base case is now complete.

Inductive Step

Our main theorem, `plus_commutes` should currently be in the following state:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

Looking again at the type of `plus_commutes_S`, we have:

```
  k : Nat
  m : Nat
--------------------------------------
plus_commutes_S : S (plus k m) = plus m (S k)
```

Conveniently, by induction we can immediately tell that `plus k m = plus m k`, so let us rewrite directly by making a recursive call to `plus_commutes`. We add this directly, by hand, as follows:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = rewrite plus_commutes k m in ?plus_commutes_S
```

Checking the type of `plus_commutes_S` now gives:

[1] Note that the left and right hand sides of the equality have been swapped, because `replace` takes a proof of `x=y` and the property for `x`, not `y`.

```
  k : Nat
  m : Nat
  _rewrite_rule : plus m k = plus k m
--------------------------------------
plus_commutes_S : S (plus m k) = plus m (S k)
```

The good news is that `m` and `k` now appear in the correct order. However, we still have to show that the successor symbol `S` can be moved to the front in the right hand side of this equality. This remaining lemma takes a similar form to the `plus_commutes_Z`; we begin by making a new top level lemma with `\l`. This gives:

```
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
```

Unlike the previous case, `k` and `m` are not made implicit because we cannot in general infer arguments to a function from its result. Again, we make a template definition with `\d`:

```
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
plus_commutes_S k m = ?plus_commutes_S_rhs
```

Again, this is defined by induction over `m`, since `plus` is defined by matching on its first argument. The complete definition is:

```
total
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
plus_commutes_S k Z = Refl
plus_commutes_S k (S j) = rewrite plus_commutes_S k j in Refl
```

All holes have now been solved.

The `total` annotation means that we require the final function to pass the totality checker; i.e. it will terminate on all possible well-typed inputs. This is important for proofs, since it provides a guarantee that the proof is valid in *all* cases, not just those for which it happens to be well-defined.

Now that `plus_commutes` has a `total` annotation, we have completed the proof of commutativity of addition on natural numbers.

DEPRECATED: Interactive Theorem Proving

> **Warning:** The interactive theorem-proving interface documented here has been deprecated in favor of *Elaborator Reflection* (page 197).

Idris also supports interactive theorem proving via tactics. This is generally not recommended to be used directly, but rather used as a mechanism for building proof automation which is beyond the scope of this tutorial. In this section, we briefly discus tactics.

One way to write proofs interactively is to write the general *structure* of the proof, and use the interactive mode to complete the details. Consider the following definition, proved in *Theorem Proving* (page 42):

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

We'll be constructing the proof by *induction*, so we write the cases for `Z` and `S`, with a recursive call in the `S` case giving the inductive hypothesis, and insert *holes* for the rest of the definition:

```
plusReducesZ' : (n:Nat) -> n = plus n Z
plusReducesZ' Z     = ?plusredZ_Z
plusReducesZ' (S k) = let ih = plusReducesZ' k in
                      ?plusredZ_S
```

On running , two global names are created, `plusredZ_Z` and `plusredZ_S`, with no definition. We can use the `:m` command at the prompt to find out which holes are still to be solved (or, more precisely, which functions exist but have no definitions), then the `:t` command to see their types:

```
*theorems> :m
Global holes:
        [plusredZ_S,plusredZ_Z]


*theorems> :t plusredZ_Z
plusredZ_Z : Z = plus Z Z

*theorems> :t plusredZ_S
plusredZ_S : (k : Nat) -> (k = plus k Z) -> S k = plus (S k) Z
```

The `:p` command enters interactive proof mode, which can be used to complete the missing definitions.

```
*theorems> :p plusredZ_Z

--------------------------------- (plusredZ_Z) --------
{hole0} : Z = plus Z Z
```

This gives us a list of premises (above the line; there are none here) and the current goal (below the line; named `{hole0}` here). At the prompt we can enter tactics to direct the construction of the proof. In this case, we can normalise the goal with the `compute` tactic:

```
-plusredZ_Z> compute

--------------------------------- (plusredZ_Z) --------
{hole0} : Z = Z
```

Now we have to prove that `Z` equals `Z`, which is easy to prove by `Refl`. To apply a function, such as `Refl`, we use `refine` which introduces subgoals for each of the function's explicit arguments (`Refl` has none):

```
-plusredZ_Z> refine Refl
plusredZ_Z: no more goals
```

Here, we could also have used the `trivial` tactic, which tries to refine by `Refl`, and if that fails, tries to refine by each name in the local context. When a proof is complete, we use the `qed` tactic to add the proof to the global context, and remove the hole from the unsolved holes list. This also outputs a trace of the proof:

```
-plusredZ_Z> qed
plusredZ_Z = proof
    compute
    refine Refl


*theorems> :m
Global holes:
        [plusredZ_S]
```

The `:addproof` command, at the interactive prompt, will add the proof to the source file (effectively in an appendix). Let us now prove the other required lemma, `plusredZ_S`:

```
*theorems> :p plusredZ_S

---------------------------------- (plusredZ_S) --------
{hole0} : (k : Nat) -> (k = plus k Z) -> S k = plus (S k) Z
```

In this case, the goal is a function type, using `k` (the argument accessible by pattern matching) and `ih` — the local variable containing the result of the recursive call. We can introduce these as premises using the `intro` tactic twice (or `intros`, which introduces all arguments as premises). This gives:

```
  k : Nat
  ih : k = plus k Z
---------------------------------- (plusredZ_S) --------
{hole2} : S k = plus (S k) Z
```

Since plus is defined by recursion on its first argument, the term `plus (S k) Z` in the goal can be simplified, so we use `compute`.

```
  k : Nat
  ih : k = plus k Z
---------------------------------- (plusredZ_S) --------
{hole2} : S k = S (plus k Z)
```

We know, from the type of `ih`, that `k = plus k Z`, so we would like to use this knowledge to replace `plus k Z` in the goal with `k`. We can achieve this with the `rewrite` tactic:

```
-plusredZ_S> rewrite ih

  k : Nat
  ih : k = plus k Z
---------------------------------- (plusredZ_S) --------
{hole3} : S k = S k

-plusredZ_S>
```

The `rewrite` tactic takes an equality proof as an argument, and tries to rewrite the goal using that proof. Here, it results in an equality which is trivially provable:

```
-plusredZ_S> trivial
plusredZ_S: no more goals
-plusredZ_S> qed
plusredZ_S = proof {
    intros;
    rewrite ih;
    trivial;
}
```

Again, we can add this proof to the end of our source file using the `:addproof` command at the interactive prompt.

CHAPTER 6

Language Reference

This is the reference guide for the Idris Language. It documents the language specification and internals. This will tell you how Idris works, for using it you should read the Idris Tutorial.

Code Generation Targets

`Idris` has been designed such that the compiler can generate code for different backends upon request. By default `Idris` generates a `C` backend when generating an executable. Included within the standard Idris installation are backends for Javascript and Node.js.

However, there are third-party code generators out there. Below we describe some of these backends and how you can use them when compiling your `Idris` code. If you want to write your own codegen for your language there is a stub project on GitHub that can help point you in the right direction.

Official Backends

C Language

Javascript

To generate code that is tailored for running in the browser issue the following command:

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

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
```

Idris can produce very big chunks of JavaScript code (hello world weighs in at 1500 lines). However, the generated code can be minified using the closure-compiler from Google.

```
java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --js hello.js
```

Node.js

Third Party

Note: These are third-party code generations and may have bit-rotted or do not work with current versions of Idris. Please speak to the project's maintainors if there are any problems.

CIL (.NET, Mono, Unity)

```
idris --codegen cil Main.idr -o HelloWorld.exe \
  && mono HelloWorld.exe
```

The resulting assemblies can also be used with .NET or Unity.

Requires idris-cil.

Erlang

Available online

Java

Available online

```
idris hello.idr --codegen java -o hello.jar
```

Note: The resulting .jar is automatically prefixed by a header including an .sh script to allow executing it directly.

JVM

Available online

LLVM

Available online

Malfunction

Available online

Ocaml

Available online

PHP

Available online

Python

Available online

Ruby

Available online

WS

Available online

Documenting Idris Code

Idris documentation comes in two major forms: comments, which exist for a reader's edification and are ignored by the compiler, and inline API documentation, which the compiler parses and stores for future reference. To consult the documentation for a declaration `f`, write `:doc f` at the REPL or use the appropriate command in your editor (`C-c C-d` in Emacs, `<LocalLeader>h` in Vim).

Comments

Use comments to explain why code is written the way that it is. Idris's comment syntax is the same as that of Haskell: lines beginning with `--` are comments, and regions bracketed by `{-` and `-}` are comments even if they extend across multiple lines. These can be used to comment out lines of code or provide simple documentation for the readers of Idris code.

Inline Documentation

Idris also supports a comprehensive and rich inline syntax for Idris code to be generated. This syntax also allows for named parameters and variables within type signatures to be individually annotated using a syntax similar to Javadoc parameter annotations.

Documentation always comes before the declaration being documented. Inline documentation applies to either top-level declarations or to constructors. Documentation for specific arguments to constructors, type constructors, or functions can be associated with these arguments using their names.

The inline documentation for a declaration is an unbroken string of lines, each of which begins with `|||` (three pipe symbols). The first paragraph of the documentation is taken to be an overview, and in some contexts, only this overview will be shown. After the documentation for the declaration as a whole, it is possible to associate documentation with specific named parameters, which can either be explicitly name or the results of converting free variables to implicit parameters. Annotations are the same as with Javadoc annotations, that is for the named parameter `(n : T)`, the corresponding annotation is `||| @ n Some description` that is placed before the declaration.

Documentation is written in Markdown, though not all contexts will display all possible formatting (for example, images are not displayed when viewing documentation in the REPL, and only some terminals render italics correctly). A comprehensive set of examples is given below.

```
||| Modules can also be documented.
module Docs

||| Add some numbers.
|||
||| Addition is really great. This paragraph is not part of the overview.
||| Still the same paragraph.
|||
||| You can even provide examples which are inlined in the documentation:
||| ```idris example
||| add 4 5
||| ```
|||
||| Lists are also nifty:
||| * Really nifty!
||| * Yep!
||| * The name `add` is a **bold** choice
||| @ n is the recursive param
||| @ m is not
add : (n, m : Nat) -> Nat
add Z     m = m
add (S n) m = S (add n m)


||| Append some vectors
||| @ a the contents of the vectors
||| @ xs the first vector (recursive param)
||| @ ys the second vector (not analysed)
appendV : (xs : Vect n a) -> (ys : Vect m a) -> Vect (add n m) a
appendV []      ys = ys
appendV (x::xs) ys = x :: appendV xs ys

||| Here's a simple datatype
data Ty =
  ||| Unit
  UNIT |
  ||| Functions
  ARR Ty Ty

||| Points to a place in a typing context
```

```
data Elem : Vect n Ty -> Ty -> Type where
  Here : {ts : Vect n Ty} -> Elem (t::ts) t
  There : {ts : Vect n Ty} -> Elem ts t -> Elem (t'::ts) t

||| A more interesting datatype
||| @ n the number of free variables
||| @ ctxt a typing context for the free variables
||| @ ty the type of the term
data Term : (ctxt : Vect n Ty) -> (ty : Ty) -> Type where

  ||| The constructor of the unit type
  ||| More comment
  ||| @ ctxt the typing context
  UnitCon : {ctxt : Vect n Ty} -> Term ctxt UNIT

  ||| Function application
  ||| @ f the function to apply
  ||| @ x the argument
  App : {ctxt : Vect n Ty} -> (f : Term ctxt (ARR t1 t2)) -> (x : Term ctxt t1) -> Term ctxt t2

  ||| Lambda
  ||| @ body the function body
  Lam : {ctxt : Vect n Ty} -> (body : Term (t1::ctxt) t2) -> Term ctxt (ARR t1 t2)

  ||| Variables
  ||| @ i de Bruijn index
  Var : {ctxt : Vect n Ty} -> (i : Elem ctxt t) -> Term ctxt t

||| A computation that may someday finish
codata Partial : Type -> Type where

  ||| A finished computation
  ||| @ value the result
  Now : (value : a) -> Partial a

  ||| A not-yet-finished computation
  ||| @ rest the remaining work
  Later : (rest : Partial a) -> Partial a

||| We can document records, including their fields and constructors
record Yummy where
  ||| Make a yummy
  constructor MkYummy
  ||| What to eat
  food : String
```

Packages

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

Package Descriptions

A package description includes the following:

- A header, consisting of the keyword package followed by the 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, a library test which has two modules `foo.idr` and `bar.idr` as source files would be written as follows:

```
package foo

modules = foo, bar
```

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

Metadata

From Idris *v0.12* the *iPKG* format supports additional metadata associated with the package. The added fields are:

- `brief = "<text>"`, a string literal containing a brief description of the package.
- `version = <text>`, a version string to associate with the package.
- `readme = <file>`, location of the README file.
- `license = <text>`, a string description of the licensing information.
- `author = <text>`, the author information.
- `maintainer = <text>`, Maintainer information.
- `homepage = <url>`, the website associated with the package.
- `sourceloc = <url>`, the location of the DVCS where the source can be found.
- `bugtracker = <url>`, the location of the project's bug tracker.

Common Fields

Other common fields which may be present in an `ipkg` file are:

- `sourcedir = <dir>`, which takes the directory (relative to the current directory) which contains the source. Default is the current directory.
- `executable = <output>`, which takes the name of the executable file to generate. Executable names can be any valid Idris identifier. the iPKG format also takes a quoted version that accepts any valid filename.
- `main = <module>`, which takes the name of the main module, and must be present if the executable field is present.
- `opts = "<idris options>"`, which allows options to be passed to Idris.
- `pkgs = <pkg name> (',' <pkg name>)+`, a comma separated list of package names that the Idris package requires.

Binding to C

In more advanced cases, particularly to support creating bindings to external `C` libraries, the following options are available:

- `makefile = <file>`, which specifies a `Makefile`, to be built before the Idris modules, for example to support linking with a `C` library.
- `libs = <libs>`, which takes a comma separated list of libraries which must be present for the package to be usable.
- `objs = <objs>`, which takes a comma separated list of additional files to be installed (object files, headers), perhaps generated by the `Makefile`.

Testing

For testing Idris packages there is a rudimentary testing harness, run in the `IO` context. The `iPKG` file is used to specify the functions used for testing. The following option is available:

- `tests = <test functions>`, which takes the qualified names of all test functions to be run.

Important: The modules containing the test functions must also be added to the list of modules.

Comments

Package files support comments using the standard Idris singleline `--` and multiline `{- -}` format.

Using Package files

Given an Idris package file `text.ipkg` it can be used with the Idris compiler as follows:

- `idris --build test.ipkg` will build all modules in the package
- `idris --install test.ipkg` will install the package, making it accessible by other Idris libraries and programs.
- `idris --clean test.ipkg` will delete all intermediate code and executable files generated when building.
- `idris --mkdoc test.ipkg` will build HTML documentation for your package in the folder `test_doc` in your project's root directory.
- `idris --installdoc test.ipkg` will install the packages documentation into Idris' central documentation folder located at `idris --docdir`.
- `idris --checkpkg test.ipkg` will type check all modules in the package only. This differs from build that type checks **and** generates code.
- `idris --testpkg test.ipkg` will compile and run any embedded tests you have specified in the `tests` parameter.

When building or install packages the commandline flag `--warnipkg` will audit the project and warn of any potentiable problems.

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

```
idris -p test Main.idr
```

Uniqueness Types

Uniqueness Types are an experimental feature available from Idris 0.9.15. A value with a unique type is guaranteed to have *at most one* reference to it at run-time, which means that it can safely be updated in-place, reducing the need for memory allocation and garbage collection. The motivation is that we would like to be able to write reactive systems, programs which run in limited memory environments, device drivers, and any other system with hard real-time requirements, ideally while giving up as little high level conveniences as possible.

They are inspired by linear types, Uniqueness Types in the Clean programming language, and ownership types and borrowed pointers in the Rust programming language.

Some things we hope to be able to do eventually with uniqueness types include:

- Safe, pure, in-place update of arrays, lists, etc
- Provide guarantees of correct resource usage, state transitions, etc
- Provide guarantees that critical program fragments will *never* allocate

Using Uniqueness

If `x : T` and `T : UniqueType`, then there is at most one reference to `x` at any time during run-time execution. For example, we can declare the type of unique lists as follows:

```
data UList : Type -> UniqueType where
     Nil  : UList a
     (::) : a -> UList a -> UList a
```

If we have a value `xs : UList a`, then there is at most one reference to `xs` at run-time. The type checker preserves this guarantee by ensuring that there is at most one reference to any value of a unique type in a pattern clause. For example, the following function definition would be valid:

```
umap : (a -> b) -> UList a -> UList b
umap f [] = []
umap f (x :: xs) = f x :: umap f xs
```

In the second clause, `xs` is a value of a unique type, and only appears once on the right hand side, so this clause is valid. Not only that, since we know there can be no other reference to the `UList a` argument, we can reuse its space for building the result! The compiler is aware of this, and compiles this definition to an in-place update of the list.

The following function definition would not be valid (even assuming an implementation of `++`), however, since `xs` appears twice:

```
dupList : UList a -> UList a
dupList xs = xs ++ xs
```

This would result in a shared pointer to `xs`, so the typechecker reports:

```
unique.idr:12:5:Unique name xs is used more than once
```

If we explicitly copy, however, the typechecker is happy:

```
dup : UList a -> UList a
dup [] = []
dup (x :: xs) = x :: x :: dup xs
```

Note that it's fine to use `x` twice, because `a` is a `Type`, rather than a `UniqueType`.

There are some other restrictions on where a `UniqueType` can appear, so that the uniqueness property is preserved. In particular, the type of the function type, `(x : a) -> b` depends on the type of `a` or `b` - if either is a `UniqueType`, then the function type is also a `UniqueType`. Then, in a data declaration, if the type constructor builds a `Type`, then no constructor can have a `UniqueType`. For example, the following definition is invalid, since it would embed a unique value in a possible non-unique value:

```
data BadList : UniqueType -> Type where
     Nil  : {a : UniqueType} -> BadList a
     (::) : {a : UniqueType} -> a -> BadList a -> BadList a
```

Finally, types may be polymorphic in their uniqueness, to a limited extent. Since `Type` and `UniqueType` are different types, we are limited in how much we can use polymorphic functions on unique types. For example, if we have function composition defined as follows:

```
(.) : {a, b, c : Type} -> (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

And we have some functions over unique types:

```
foo : UList a -> UList b
bar : UList b -> UList c
```

Then we cannot compose `foo` and `bar` as `bar . foo`, because `UList` does not compute a `Type`! Instead, we can define composition as follows:

```
(.) : {a, b, c : Type*} -> (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

The `Type*` type stands for either unique or non-unique types. Since such a function may be passed a `UniqueType`, any value of type `Type*` must also satisfy the requirement that it appears at most once on the right hand side.

Borrowed Types

It quickly becomes obvious when working with uniqueness types that having only one reference at a time can be painful. For example, what if we want to display a list before updating it?

```
showU : Show a => UList a -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : UList a -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (x :: xs) = show x ++ ", " ++ showU' xs
```

This is a valid definition of `showU`, but unfortunately it consumes the list! So the following function would be invalid:

```
printAndUpdate : UList Int -> IO ()
printAndUpdate xs = do putStrLn (showU xs)
                       let xs' = umap (*2) xs -- xs no longer available!
                       putStrLn (showU xs')
```

Still, one would hope to be able to display a unique list without problem, since it merely *inspects* the list; there are no updates. We can achieve this, using the notion of *borrowing*. A Borrowed type is a Unique type which can be inspected at the top level (by pattern matching, or by *lending* to another function) but no further. This ensures that the internals (i.e. the arguments to top level patterns) will not be passed to any function which will update them.

`Borrowed` converts a `UniqueType` to a `BorrowedType`. It is defined as follows (along with some additional rules in the typechecker):

```
data Borrowed : UniqueType -> BorrowedType where
     Read : {a : UniqueType} -> a -> Borrowed a

implicit
lend : {a : UniqueType} -> a -> Borrowed a
lend x = Read x
```

A value can be "lent" to another function using `lend`. Arguments to `lend` are not counted by the type checker as a reference to a unique value, therefore a value can be lent as many times as desired. Using this, we can write `showU` as follows:

```
showU : Show a => Borrowed (UList a) -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : Borrowed (UList a) -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (Read (x :: xs)) = show x ++ ", " ++ showU' (lend xs)
```

Unlike a unique value, a borrowed value may be referred to as many times as desired. However, there is a restriction on how a borrowed value can be used. After all, much like a library book or your neighbour's lawnmower, if a function borrows a value it is expected to return it in exactly the condition in which it was received!

The restriction is that when a `Borrowed` type is matched, any pattern variables under the `Read` which have a unique type may not be referred to at all on the right hand side (unless they are themselves `lent` to another function).

Uniqueness information is stored in the type, and in particular in function types. Once we're in a unique context, any new function which is constructed will be required to have unique type, which prevents the following sort of bad program being implemented:

```
foo : UList Int -> IO ()
foo xs = do let f = \x : Int => showU xs
            putStrLn $ free xs
            putStrLn $ f 42
            pure ()
```

Since `lend` is implicit, in practice for functions to lend and borrow values merely requires the argument to be marked as `Borrowed`. We can therefore write `showU` as follows:

```
showU : Show a => Borrowed (UList a) -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : Borrowed (UList a) -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (x :: xs) = show x ++ ", " ++ showU' xs
```

Problems/Disadvantages/Still to do...

This is a work in progress, there is lots to do. The most obvious problem is the loss of abstraction. On the one hand, we have more precise control over memory usage with `UniqueType` and `BorrowedType`, but they are not in general compatible with functions polymorphic over `Type`. In the short term, we can start to write reactive and low memory systems with this, but longer term it would be nice to support more abstraction.

We also haven't checked any of the metatheory, so this could all be fatally flawed! The implementation is based to a large extent on Uniqueness Typing Simplified, by de Vries et al, so there is reason to believe things should be fine, but we still have to do the work.

Much as there are with linear types, there are some annoyances when trying to prove properties of functions with unique types (for example, what counts as a use of a value). Since we require *at most* one use of a value, rather than *exactly* one, this seems to be less of an issue in practice, but still needs thought.

New Foreign Function Interface

Ever since Idris has had multiple backends compiling to different target languages on potentially different platforms, we have had the problem that the foreign function interface (FFI) was written under the assumption of compiling to C. As a result, it has been hard to write generic code for multiple targets, or even to be sure that if code compiles that it will run on the expected target.

As of 0.9.17, Idris will have a new foreign function interface (FFI) which is aware of multiple targets. Users who are working with the default code generator can happily continue writing programs as before with no changes, but if you are writing bindings for an external library, writing a back end, or working with a non-C back end, there are some things you will need to be aware of, which this page describes.

The `IO'` monad, and `main`

The `IO` monad exists as before, but is now specific to the C backend (or, more precisely, any backend whose foreign function calls are compatible with C.) Additionally, there is now an `IO'` monad, which is parameterised over a FFI descriptor:

```
data IO' : (lang : FFI) -> Type -> Type
```

The Prelude defines two FFI descriptors which are imported automatically, for C and JavaScript/Node, and defines `IO` to use the C FFI and `JS_IO` to use the JavaScript FFI:

```
FFI_C  : FFI
FFI_JS : FFI

IO : Type -> Type
IO a = IO' FFI_C a

JS_IO : Type -> Type
JS_IO a = IO' FFI_JS a
```

As before, the entry point to an Idris program is `main`, but the type of `main` can now be any implementation of `IO'`, e.g. the following are both valid:

```
main : IO ()
main : JS_IO ()
```

The FFI descriptor includes details about which types can be marshalled between the foreign language and Idris, and the "target" of a foreign function call (typically just a String representation of the function's name, but potentially something more complicated such as an external library file or even a URL).

FFI descriptors

An FFI descriptor is a record containing a predicate which holds when a type can be marshalled, and the type of the target of a foreign call:

```
record FFI where
     constructor MkFFI
     ffi_types : Type -> Type
     ffi_fn : Type
```

For C, this is:

```
||| Supported C integer types
public export
data C_IntTypes : Type -> Type where
    C_IntChar   : C_IntTypes Char
    C_IntNative : C_IntTypes Int
    C_IntBits8  : C_IntTypes Bits8
    C_IntBits16 : C_IntTypes Bits16
    C_IntBits32 : C_IntTypes Bits32
    C_IntBits64 : C_IntTypes Bits64

||| Supported C function types
public export
data C_FnTypes : Type -> Type where
    C_Fn : C_Types s -> C_FnTypes t -> C_FnTypes (s -> t)
    C_FnIO : C_Types t -> C_FnTypes (IO' FFI_C t)
    C_FnBase : C_Types t -> C_FnTypes t

||| Supported C foreign types
public export
data C_Types : Type -> Type where
    C_Str   : C_Types String
    C_Float : C_Types Double
    C_Ptr   : C_Types Ptr
    C_MPtr  : C_Types ManagedPtr
    C_Unit  : C_Types ()
    C_Any   : C_Types (Raw a)
    C_FnT   : C_FnTypes t -> C_Types (CFnPtr t)
    C_IntT  : C_IntTypes i -> C_Types i

||| A descriptor for the C FFI. See the constructors of `C_Types`
||| and `C_IntTypes` for the concrete types that are available.
%error_reverse
public export
FFI_C : FFI
    FFI_C = MkFFI C_Types String String
```

Foreign calls

To call a foreign function, the `foreign` function is used. For example:

```
do_fopen : String -> String -> IO Ptr
do_fopen f m
   = foreign FFI_C "fileOpen" (String -> String -> IO Ptr) f m
```

The `foreign` function takes an FFI description, a function name (the type is given by the `ffi_fn` field of `FFI_C` here), and a function type, which gives the expected types of the remaining arguments. Here, we're calling an external function `fileOpen` which takes, in the C, a `char*` file name, a `char*` mode, and returns a file pointer. It is the job of the C back end to convert Idris `String` to C `char*` and vice versa.

The argument types and return type given here must be present in the `fn_types` predicate of the `FFI_C` description for the foreign call to be valid.

Note The arguments to `foreign` *must* be known at compile time, because the foreign calls are generated statically. The `%inline` directive on a function can be used to give hints to help this, for example a shorthand for calling external JavaScript functions:

```
%inline
jscall : (fname : String) -> (ty : Type) ->
         {auto fty : FTy FFI_JS [] ty} -> ty
jscall fname ty = foreign FFI_JS fname ty
```

C callbacks

It is possible to pass an Idris function to a C function taking a function pointer by using `CFnPtr` in the function type. The Idris function is passed to `MkCFnPtr` in the arguments. The example below shows declaring the C standard library function `qsort` which takes a pointer to a comparison function.

```
myComparer : Ptr -> Ptr -> Int
myComparer = ...

qsort : Ptr -> Int -> Int -> IO ()
qsort data elems elsize = foreign FFI_C "qsort"
                (Ptr -> Int -> Int -> CFnPtr (Ptr -> Ptr -> Int) -> IO ())
                data elems elsize (MkCFnPtr myComparer)
```

There are a few limitations to callbacks in the C FFI. The foreign function can't take the function to make a callback of as an argument. This will give a compilation error:

```
-- This does not work
example : (Int -> ()) -> IO ()
example f = foreign FFI_C "callbacker" (CFnPtr (Int -> ()) -> IO ()) f
```

The other big limitation is that it doesn't support IO functions. Use `unsafePerformIO` to wrap them (i.e. to make an IO function usable as a callback, change the return type from IOr to r, and change the = do to = unsafePerformIO $ do).

There are two special function names: `%wrapper` returns the function pointer that wraps an Idris function. This is useful if the function pointer isn't taken by a C function directly but should be inserted into a data structure. A foreign declaration using `%wrapper` must return `IO Ptr`.

```
-- this returns the C function pointer to a qsort comparer
example_wrapper : IO Ptr
example_wrapper = foreign FFI_C "%wrapper" (CFnPtr (Ptr -> Ptr -> Int) -> IO Ptr)
                          (MkCFnPtr myComparer)
```

`%dynamic` calls a C function pointer with some arguments. This is useful if a C function returns or data structure contains a C function pointer, for example structs of function pointers are common in object-oriented C such as in COM or the Linux kernel. The function type contains an extra `Ptr` at the start for the function pointer. `%dynamic` can be seen as a pseudo-function that calls the function in the first argument, passing the remaining arguments to it.

```
-- we have a pointer to a function with the signature int f(int), call it
example_dynamic : Ptr -> Int -> IO Int
example_dynamic fn x = foreign FFI_C "%dynamic" (Ptr -> Int -> IO Int) fn x
```

If the foreign name is prefixed by a `&`, it is treated as a pointer to the global variable with the following name. The type must be just `IO Ptr`.

```
-- access the global variable errno
errno : IO Ptr
errno = foreign FFI_C "&errno" (IO Ptr)
```

For more complicated interactions with C (such as reading and setting fields of a C struct), there is a module CFFI available in the contrib package.

FFI implementation

In order to write bindings to external libraries, the details of how `foreign` works are unnecessary — you simply need to know that `foreign` takes an FFI descriptor, the function name, and its type. It is instructive to look a little deeper, however:

The type of `foreign` is as follows:

```
foreign : (ffi : FFI)
       -> (fname : ffi_fn f)
       -> (ty : Type)
       -> {auto fty : FTy ffi [] ty}
       -> ty
```

The important argument here is the implicit `fty`, which contains a proof (`FTy`) that the given type is valid according to the FFI description `ffi`:

```
data FTy : FFI -> List Type -> Type -> Type where
     FRet : ffi_types f t -> FTy f xs (IO' f t)
     FFun : ffi_types f s -> FTy f (s :: xs) t -> FTy f xs (s -> t)
```

Notice that this uses the `ffi_types` field of the FFI descriptor — these arguments to `FRet` and `FFun` give explicit proofs that the type is valid in this FFI. For example, the above `do_fopen` builds the following implicit proof as the `fty` argument to `foreign`:

```
FFun C_Str (FFun C_Str (FRet C_Ptr))
```

Compiling foreign calls

(This section assumes some knowledge of the Idris internals.)

When writing a back end, we now need to know how to compile `foreign`. We'll skip the details here of how a `foreign` call reaches the intermediate representation (the IR), though you can look in `IO.idr` in the `prelude` package to see a bit more detail — a `foreign` call is implemented by the primitive function `mkForeignPrim`. The important part of the IR as defined in `Lang.hs` is the following constructor:

```
data LExp = ...
          | LForeign FDesc -- Function descriptor
                     FDesc -- Return type descriptor
                     [(FDesc, LExp)]
```

So, a `foreign` call appears in the IR as the `LForeign` constructor, which takes a function descriptor (of a type given by the `ffi_fn` field in the FFI descriptor), a return type descriptor (given by an application of `FTy`), and a list of arguments with type descriptors (also given by an application of `FTy`).

An `FDesc` describes an application of a name to some arguments, and is really just a simplified subset of an `LExp`:

```
data FDesc = FCon Name
           | FStr String
           | FUnknown
           | FApp Name [FDesc]
```

There are corresponding structures in the lower level IRs, such as the defunctionalised, simplified and bytecode forms.

Our `do_fopen` example above arrives in the `LExp` form as:

```
LForeign (FStr "fileOpen") (FCon (sUN "C_Ptr"))
         [(FCon (sUN "C_Str"), f), (FCon (sUN "C_Str"), m)]
```

(Assuming that `f` and `m` stand for the `LExp` representations of the arguments.) This information should be enough for any back end to marshal the arguments and return value appropriately.

Note: When processing `FDesc`, be aware that there may be implicit arguments, which have not been erased. For example, `C_IntT` has an implicit argument `i`, so will appear in an `FDesc` as something of the form `FApp (sUN "C_IntT") [i, t]` where `i` is the implicit argument (which can be ignored) and `t` is the descriptor of the integer type. See `CodegenC.hs`, specifically the function `toFType`, to see how this works in practice.

JavaScript FFI descriptor

The JavaScript FFI descriptor is a little more complex, because the JavaScript FFI supports marshalling functions. It is defined as follows:

```
mutual
  data JsFn t = MkJsFn t

  data JS_IntTypes  : Type -> Type where
       JS_IntChar   : JS_IntTypes Char
       JS_IntNative : JS_IntTypes Int

  data JS_FnTypes : Type -> Type where
       JS_Fn     : JS_Types s -> JS_FnTypes t -> JS_FnTypes (s -> t)
       JS_FnIO   : JS_Types t -> JS_FnTypes (IO' l t)
       JS_FnBase : JS_Types t -> JS_FnTypes t

  data JS_Types : Type -> Type where
       JS_Str   : JS_Types String
       JS_Float : JS_Types Double
       JS_Ptr   : JS_Types Ptr
       JS_Unit  : JS_Types ()
```

```
    JS_FnT   : JS_FnTypes a -> JS_Types (JsFn a)
    JS_IntT  : JS_IntTypes i -> JS_Types i
```

The reason for wrapping function types in a `JsFn` is to help the proof search when building `FTy`. We hope to improve proof search eventually, but for the moment it works much more reliably if the indices are disjoint! An example of using this appears in IdrisScript when setting timeouts:

```
setTimeout : (() -> JS_IO ()) -> (millis : Int) -> JS_IO Timeout
setTimeout f millis = do
  timeout <- jscall "setTimeout(%0, %1)"
                    (JsFn (() -> JS_IO ()) -> Int -> JS_IO Ptr)
                    (MkJsFn f) millis
  pure $ MkTimeout timeout
```

Syntax Guide

Examples are mostly adapted from the Idris tutorial.

Source File Structure

Source files consist of:

1. An optional *Module Header* (page 162).
2. Zero or more *Imports* (page 163).
3. Zero or more declarations, e.g. *Variables* (page 163), *Data types* (page 164), etc.

For example:

```
module MyModule   -- module header

import Data.Vect  -- an import

%default total    -- a directive

foo : Nat         -- a declaration
foo = 5
```

Module Header

A file can start with a module header, introduced by the `module` keyword:

```
module Semantics
```

Module names can be hierarchical, with parts separated by `.`:

```
module Semantics.Transform
```

Each file can define only a single module, which includes everything defined in that file.

Like with declarations, a *docstring* (page 168) can be used to provide documentation for a module:

```
||| Implementation of predicate transformer semantics.
module Semantics.Transform
```

Imports

An `import` makes the names in another module available for use by the current module:

```
import Data.Vect
```

All the declarations in an imported module are available for use in the file. In a case where a name is ambiguous — e.g. because it is imported from multiple modules, or appears in multiple visible namespaces — the ambiguity can be resolved using *Qualified Names* (page 167). (Often, the compiler can resolve the ambiguity for you, using the types involved.)

Imported modules can be given aliases to make qualified names more compact:

```
import Data.Vect as V
```

Note that names made visible by import are not, by default, re-exported to users of the module being written. This can be done using `import public`:

```
import public Data.Vect
```

Variables

A variable is always defined by defining its type on one line, and its value on the next line, using the syntax

```
<id> : <type>
<id> = <value>
```

Examples

```
x : Int
x = 100
hello : String
hello = "hello"
```

Types

In Idris, types are first class values. So a type declaration is the same as just declaration of a variable whose type is `Type`. In Idris, variables that denote a type need not be capitalised. Example:

```
MyIntType : Type
MyIntType = Int
```

a more interesting example:

```
MyListType : Type
MyListType = List Int
```

While capitalising types is not required, the rules for generating implicit arguments mean it is often a good idea.

Data types

Idris provides two kinds of syntax for defining data types. The first, Haskell style syntax, defines a regular algebraic data type. For example

```
data Either a b = Left a | Right b
```

or

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

The second, more general kind of data type, is defined using Agda or GADT style syntax. This syntax defines a data type that is parameterised by some values (in the `Vect` example, a value of type `Nat` and a value of type `Type`).

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
```

The signature of type constructors may use dependent types

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

Records

There is a special syntax for data types with one constructors and multiple fields.

```
record A a where
  constructor MkA
  foo, bar : a
  baz : Nat
```

This defines a constructor as well as getter and setter function for each field.

```
MkA : a -> a -> Nat -> A a
foo : A a -> a
set_foo : a -> A a -> A a
```

The types of record fields may depend on the value of other fields

```
record Collection a where
  constructor MkCollection
  size : Nat
  items : Vect size a
```

Setter functions are only provided for fields that do not use dependant types. In the example above neither `set_size` nor `set_items` are defined.

Co-data

Inifinite data structures can be introduced with the `codata` keyword.

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

This is syntactic sugar for the following, which is usually preferred:

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

Every occurence of the the defined type in a constructor argument will be wrapped in the `Inf` type constructor. This has the effect of delaying the evaluation of the second argument when the data constructor is applied. An `Inf` argument is constructed using `Delay` (which Idris will insert implicitly) adn evaluated using `Force` (again inserted implicitly).

Furthermore, recursive calls under a `Delay` must be guarded by a constructor to pass the totality checker.

Operators

Arithmetic

```
x + y
x - y
x * y
x / y
(x * y) + (a / b)
```

Equality and Relational

```
x == y
x /= y
x >= y
x > y
x <= y
x < y
```

Conditional

```
x && y
x || y
not x
```

Conditionals

If Then Else

```
if <test> then <true> else <false>
```

Case Expressions

```
case <test> of
    <case 1>  => <expr>
    <case 2>  => <expr>
    ...
    otherwise => <expr>
```

Functions

Named

Named functions are defined in the same way as variables, with the type followed by the definition.

```
<id> : <argument type> -> <return type>
<id> arg = <expr>
```

Example

```
plusOne : Int -> Int
plusOne x = x + 1
```

Functions can also have multiple inputs, for example

```
makeHello : String -> String -> String
makeHello first last = "hello, my name is " ++ first ++ " " ++ last
```

Functions can also have named arguments. This is required if you want to annotate parameters in a docstring. The following shows the same `makeHello` function as above, but with named parameters which are also annotated in the docstring

```
||| Makes a string introducing a person
||| @first The person's first name
||| @last The person's last name
makeHello : (first : String) -> (last : String) -> String
makeHello first last = "hello, my name is " ++ first ++ " " ++ last
```

Like Haskell, Idris functions can be defined by pattern matching. For example

```
sum : List Int -> Int
sum []        = 0
sum (x :: xs) = x + (sum xs)
```

Similarly case analysis looks like

```
answerString : Bool -> String
answerString False = "Wrong answer"
answerString True = "Correct answer"
```

Dependent Functions

Dependent functions are functions where the type of the return value depends on the input value. In order to define a dependent function, named parameters must be used, since the parameter will appear in the return type. For example, consider

```
zeros : (n : Nat) -> Vect n Int
zeros Z     = []
zeros (S k) = 0 :: (zeros k)
```

In this example, the return type is `Vect n Int` which is an expression which depends on the input parameter `n`.

Anonymous

Arguments in anonymous functions are separated by comma.

```
(\x => <expr>)
(\x, y => <expr>)
```

Modifiers

Visibility

```
public export
export
private
```

Totality

```
total
implicit
partial
covering
```

Options

```
%export
%hint
%no_implicit
%error_handler
%error_reverse
%assert_total
%reflection
%specialise [<name list>]
```

Misc

Qualified Names

If multiple declarations with the same name are visible, using the name can result in an ambiguous situation. The compiler will attempt to resolve the ambiguity using the types involved. If it's unable — for example, because the declarations with the same name also have the same type signatures — the situation can be cleared up using a *qualified name*.

A qualified name has the symbol's namespace prefixed, separated by a `.`:

```
Data.Vect.length
```

This would specifically reference a `length` declaration from `Data.Vect`.

Qualified names can be written using two different shorthands:

1. Names in modules that are *imported* (page 163) using an alias can be qualified by the alias.

2. The name can be qualified by the *shortest unique suffix* of the namespace in question. For example, the `length` case above can likely be shortened to `Vect.length`.

Comments

```
-- Single Line
{- Multiline -}
||| Docstring (goes before definition)
```

Multi line String literals

```
foo = """
this is a
string literal"""
```

Directives

```
%lib <path>
%link <path>
%flag <path>
%include <path>
%hide <function>
%freeze <name>
%access <accessibility>
%default <totality>
%logging <level 0--11>
%dynamic <list of libs>
%name <list of names>
%error_handlers <list of names>
%language <extension>
```

Erasure By Usage Analysis

This work stems from this feature proposal (obsoleted by this page). Beware that the information in the proposal is out of date — and sometimes even in direct contradiction with the eventual implementation.

Motivation

Traditional dependently typed languages (Agda, Coq) are good at erasing *proofs* (either via irrelevance or an extra universe).

```
half : (n : Nat) -> Even n -> Nat
half Z EZ = Z
half (S (S n)) (ES pf) = S (half n pf)
```

For example, in the above snippet, the second argument is a proof, which is used only to convince the compiler that the function is total. This proof is never inspected at runtime and thus can be erased. In this case, the mere existence of the proof is sufficient and we can use irrelevance-related methods to achieve erasure.

However, sometimes we want to erase *indices* and this is where the traditional approaches stop being useful, mainly for reasons described in the original proposal.

```
uninterleave : {n : Nat} -> Vect (n * 2) a -> (Vect n a, Vect n a)
uninterleave [] = ([] , [])
uninterleave (x :: y :: rest) with (unzipPairs rest)
  | (xs, ys) = (x :: xs, y :: ys)
```

Notice that in this case, the second argument is the important one and we would like to get rid of the `n` instead, although the shape of the program is generally the same as in the previous case.

There are methods described by Brady, McBride and McKinna in *[BMM04]* (page 222) to remove the indices from data structures, exploiting the fact that functions operating on them either already have a copy of the appropriate index or the index can be quickly reconstructed if needed. However, we often want to erase the indices altogether, from the whole program, even in those cases where reconstruction is not possible.

The following two sections describe two cases where doing so improves the runtime performance asymptotically.

Binary numbers

- O(n) instead of O(log n)

Consider the following `Nat`-indexed type family representing binary numbers:

```
data Bin : Nat -> Type where
  N : Bin 0
  O : {n : Nat} -> Bin n -> Bin (0 + 2*n)
  I : {n : Nat} -> Bin n -> Bin (1 + 2*n)
```

These are supposed to be (at least asymptotically) fast and memory-efficient because their size is logarithmic compared to the numbers they represent.

Unfortunately this is not the case. The problem is that these binary numbers still carry the *unary* indices with them, performing arithmetic on the indices whenever arithmetic is done on the binary numbers themselves. Hence the real representation of the number 15 looks like this:

```
I -> I -> I -> I -> N
S    S    S    Z
S    S    Z
S    S
S    Z
S
S
S
Z
```

The used memory is actually *linear*, not logarithmic and therefore we cannot get below O(n) with time complexities.

One could argue that Idris in fact compiles `Nat` via GMP but that's a moot point for two reasons:

- First, whenever we try to index our data structures with anything else than `Nat`, the compiler is not going to come to the rescue.
- Second, even with `Nat`, the GMP integers are *still* there and they slow the runtime down.

This ought not to be the case since the `Nat` are never used at runtime and they are only there for

typechecking purposes. Hence we should get rid of them and get runtime code similar to what a idris programmer would write.

U-views of lists

- O(n^2) instead of O(n)

Consider the type of U-views of lists:

```
data U : List a -> Type where
  nil : U []
  one : (z : a) -> U [z]
  two : {xs : List a} -> (x : a) -> (u : U xs) -> (y : a) -> U (x :: xs ++ [y])
```

For better intuition, the shape of the U-view of `[x0,x1,x2,z,y2,y1,y0]` looks like this:

```
x0   y0    (two)
x1   y1    (two)
x2   y1    (two)
   z       (one)
```

When recursing over this structure, the values of `xs` range over `[x0,x1,x2,z,y2,y1,y0]`, `[x1,x2,z,y2,y1]`, `[x2,z,y2]`, `[z]`. No matter whether these lists are stored or built on demand, they take up a quadratic amount of memory (because they cannot share nodes), and hence it takes a quadratic amount of time just to build values of this index alone.

But the reasonable expectation is that operations with U-views take linear time — so we need to erase the index `xs` if we want to achieve this goal.

Changes to Idris

Usage analysis is run at every compilation and its outputs are used for various purposes. This is actually invisible to the user but it's a relatively big and important change, which enables the new features.

Everything that is found to be unused is erased. No annotations are needed, just don't use the thing and it will vanish from the generated code. However, if you wish, you can use the dot annotations to get a warning if the thing is accidentally used.

"Being used" in this context means that the value of the "thing" may influence run-time behaviour of the program. (More precisely, it is not found to be irrelevant to the run-time behaviour by the usage analysis algorithm.)

"Things" considered for removal by erasure include:

- function arguments
- data constructor fields (including record fields and dictionary fields of interface implementations)

For example, `Either` often compiles to the same runtime representation as `Bool`. Constructor field removal sometimes combines with the newtype optimisation to have quite a strong effect.

There is a new compiler option `--warnreach`, which will enable warnings coming from erasure. Since we have full usage analysis, we can compile even those programs that violate erasure annotations – it's just that the binaries may run slower than expected. The warnings will be enabled by default in future versions of Idris (and possibly turned to errors). However, in this transitional period, we chose to keep them on-demand to avoid confusion until better documentation is written.

Case-tree elaboration tries to avoid using dotted "things" whenever possible. (NB. This is not yet perfect and it's being worked on: https://gist.github.com/ziman/10458331)

Postulates are no longer required to be collapsible. They are now required to be *unused* instead.

Changes to the language

You can use dots to mark fields that are not intended to be used at runtime.

```
data Bin : Nat -> Type where
  N : Bin 0
  O : .{n : Nat} -> Bin n -> Bin (0 + 2*n)
  I : .{n : Nat} -> Bin n -> Bin (1 + 2*n)
```

If these fields are found to be used at runtime, the dots will trigger a warning (with `--warnreach`).

Note that free (unbound) implicits are dotted by default so, for example, the constructor `O` can be defined as:

```
O : Bin n -> Bin (0 + 2*n)
```

and this is actually the preferred form.

If you have a free implicit which is meant to be used at runtime, you have to change it into an (undotted) `{bound : implicit}`.

You can also put dots in types of functions to get more guarantees.

```
half : (n : Nat) -> .(pf : Even n) -> Nat
```

and free implicits are automatically dotted here, too.

What it means

Dot annotations serve two purposes:

- influence case-tree elaboration to avoid dotted variables
- trigger warnings when a dotted variable is used

However, there's no direct connection between being dotted and being erased. The compiler erases everything it can, dotted or not. The dots are there mainly to help the programmer (and the compiler) refrain from using the values they want to erase.

How to use it

Ideally, few or no extra annotations are needed – in practice, it turns out that having free implicits automatically dotted is enough to get good erasure.

Therefore, just compile with `--warnreach` to see warnings if erasure cannot remove parts of the program.

However, those programs that have been written without runtime behaviour in mind, will need some help to get in the form that compiles to a reasonable binary. Generally, it's sufficient to follow erasure warnings (which may be sometimes unhelpful at the moment).

Benchmarks

- source: https://github.com/ziman/idris-benchmarks
- results: http://ziman.functor.sk/erasure-bm/

It can be clearly seen that asymptotics are improved by erasure.

Shortcomings

You can't get warnings in libraries because usage analysis starts from `Main.main`. This will be solved by the planned `%default_usage` pragma.

Usage warnings are quite bad and unhelpful at the moment. We should include more information and at least translate argument numbers to their names.

There is no decent documentation yet. This wiki page is the first one.

There is no generally accepted terminology. We switch between "dotted", "unused", "erased", "irrelevant", "inaccessible", while each has a slightly different meaning. We need more consistent and understandable naming.

If the same type is used in both erased and non-erased context, it will retain its fields to accommodate the least common denominator – the non-erased context. This is particularly troublesome in the case of the type of (dependent) pairs, where it actually means that no erasure would be performed. We should probably locate disjoint uses of data types and split them into "sub-types". There are three different flavours of dependent types now: `Sigma` (nothing erased), `Exists` (first component erased), `Subset` (second component erased).

Case-tree building does not avoid dotted values coming from pattern-matched constructors (https://gist.github.com/ziman/10458331). This is to be fixed soon. (Fixed.)

Higher-order function arguments and opaque functional variables are considered to be using all their arguments. To work around this, you can force erasure via the type system, using the `Erased` wrapper: https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Erased.idr

Interface methods are considered to be using the union of all their implementations. In other words, an argument of a method is unused only if it is unused in every implementation of the method that occurs in the program.

Planned features

- Fixes to the above shortcomings in general.
- **Improvements to the case-tree elaborator so that it properly avoids** dotted fields of data constructors. Done.
- **Compiler pragma `%default_usage` `used/unused` and per-function** overrides `used` and `unused`, which allow the programmer to mark the return value of a function as used, even if the function is not used in `main` (which is the case when writing library code). These annotations will help library writers discover usage violations in their code before it is actually published and used in compiled programs.

Troubleshooting

My program is slower

The patch introducing erasure by usage analysis also disabled some optimisations that were in place before; these are subsumed by the new erasure. However, in some erasure-unaware programs, where erasure by usage analysis does not exercise its full potential (but the old optimisations would have worked), certain slowdown may be observed (up to ~10% according to preliminary benchmarking), due to retention and computation of information that should not be necessary at runtime.

A simple check whether this is the case is to compile with `--warnreach`. If you see warnings, there is some unnecessary code getting compiled into the binary.

The solution is to change the code so that there are no warnings.

Usage warnings are unhelpful

This is a known issue and we are working on it. For now, see the section *How to read and resolve erasure warnings* (page 173).

There should be no warnings in this function

A possible cause is non-totality of the function (more precisely, non-coverage). If a function is non-covering, the program needs to inspect all arguments in order to detect coverage failures at runtime. Since the function inspects all its arguments, nothing can be erased and this may transitively cause usage violations. The solution is to make the function total or accept the fact that it will use its arguments and remove some dots from the appropriate constructor fields and function arguments. (Please note that this is not a shortcoming of erasure and there is nothing we can do about it.)

Another possible cause is the currently imperfect case-tree elaboration, which does not avoid dotted constructor fields (see https://gist.github.com/ziman/10458331). You can either rephrase the function or wait until this is fixed, hopefully soon. Fixed.

The compiler refuses to recognise this thing as erased

You can force anything to be erased by wrapping it in the `Erased` monad. While this program triggers usage warnings,

```
f : (g : Nat -> Nat) -> .(x : Nat) -> Nat
f g x = g x  -- WARNING: g uses x
```

the following program does not:

```
f : (g : Erased Nat -> Nat) -> .(x : Nat) -> Nat
f g x = g (Erase x)  -- OK
```

How to read and resolve erasure warnings

Example 1

Consider the following program:

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

```
sumLengths : List (Vect n a) -> Nat
sumLengths       []  = 0
sumLengths (v :: vs) = vlen v + sumLengths vs

main : IO ()
main = print . sumLengths $ [[0,1],[2,3]]
```

When you compile it using `--warnreach`, there is one warning:

```
Main.sumLengths: inaccessible arguments reachable:
  n (no more information available)
```

The warning does not contain much detail at this point so we can try compiling with `--dumpcases cases.txt` and look up the compiled definition in `cases.txt`:

```
Main.sumLengths {e0} {e1} {e2} =
  case {e2} of
  | Prelude.List.::({e6}) => LPlus (ATInt ITBig)({e0}, Main.sumLengths({e0}, ____, {e6}))
  | Prelude.List.Nil() => 0
```

The reason for the warning is that `sumLengths` calls `vlen`, which gets inlined. The second clause of `sumLengths` then accesses the variable `n`, compiled as `{e0}`. Since `n` is a free implicit, it is automatically considered dotted and this triggers the warning.

A solution would be either making the argument `n` a bound implicit parameter to indicate that we wish to keep it at runtime,

```
sumLengths : {n : Nat} -> List (Vect n a) -> Nat
```

or fixing `vlen` to not use the index:

```
vlen : Vect n a -> Nat
vlen [] = Z
vlen (x :: xs) = S (vlen xs)
```

Which solution is appropriate depends on the usecase.

Example 2

Consider the following program manipulating value-indexed binary numbers.

```
data Bin : Nat -> Type where
    N : Bin Z
    O : Bin n -> Bin (0 + n + n)
    I : Bin n -> Bin (1 + n + n)

toN : (b : Bin n) -> Nat
toN  N = Z
toN (O {n} bs) = 0 + n + n
toN (I {n} bs) = 1 + n + n

main : IO ()
main = print . toN $ I (I (O (O (I N))))
```

In the function `toN`, we attempted to "cheat" and instead of traversing the whole structure, we just projected the value index `n` out of constructors `I` and `O`. However, this index is a free implicit, therefore it is considered dotted.

Inspecting it then produces the following warnings when compiling with `--warnreach`:

```
Main.I: inaccessible arguments reachable:
  n from Main.toN arg# 1
Main.O: inaccessible arguments reachable:
  n from Main.toN arg# 1
```

We can see that the argument `n` of both `I` and `O` is used in the function `toN`, argument 1.

At this stage of development, warnings only contain argument numbers, not names; this will hopefully be fixed. When numbering arguments, we go from 0, taking free implicits first, left-to-right; then the bound arguments. The function `toN` has therefore in fact two arguments: `n` (argument 0) and `b` (argument 1). And indeed, as the warning says, we project the dotted field from `b`.

Again, one solution is to fix the function `toN` to calculate its result honestly; the other one is to accept that we carry a `Nat` with every constructor of `Bin` and make it a bound implicit:

```
O : {n : Nat} -> Bin n -> Bin (0 + n + n)
I : {n : Nat} -> bin n -> Bin (1 + n + n)
```

References

The IDE Protocol

The Idris REPL has two modes of interaction: a human-readable syntax designed for direct use in a terminal, and a machine-readable syntax designed for using Idris as a backend for external tools.

Protocol Overview

The communication protocol is of asynchronous request-reply style: a single request from the client is handled by Idris at a time. Idris waits for a request on its standard input stream, and outputs the answer or answers to standard output. The result of a request can be either success, failure, or intermediate output; and furthermore, before the result is delivered, there might be additional meta-messages.

A reply can consist of multiple messages: any number of messages to inform the user about the progress of the request or other informational output, and finally a result, either `ok` or `error`.

The wire format is the length of the message in characters, encoded in 6 characters hexadecimal, followed by the message encoded as S-expression (sexp). Additionally, each request includes a unique integer (counting upwards), which is repeated in all messages corresponding to that request.

An example interaction from loading the file `/home/hannes/empty.idr` looks as follows on the wire::

```
00002a((:load-file "/home/hannes/empty.idr") 1)
000039(:write-string "Type checking /home/hannes/empty.idr" 1)
000025(:set-prompt "/home/hannes/empty" 1)
000032(:return (:ok "Loaded /home/hannes/empty.idr") 1)
```

The first message is the request from idris-mode to load the specific file, which length is hex 2a, decimal 42 (including the newline at the end). The request identifier is set to 1. The first message from Idris is to write the string `Type checking /home/hannes/empty.idr`, another is to set the prompt to `*/home/hannes/empty`. The answer, starting with `:return` is `ok`, and additional information is that the file was loaded.

There are three atoms in the wire language: numbers, strings, and symbols. The only compound object

is a list, which is surrounded by parenthesis. The syntax is:

```
A ::= NUM | '"' STR '"' | ':' ALPHA+
S ::= A | '(' S* ')' | nil
```

where `NUM` is either 0 or a positive integer, `ALPHA` is an alphabetical character, and `STR` is the contents of a string, with `"` escaped by a backslash. The atom `nil` is accepted instead of `()` for compatibility with some regexp pretty-printing routines.

The state of the Idris process is mainly the active file, which needs to be kept synchronised between the editor and Idris. This is achieved by the already seen `:load-file` command.

The available commands include:

`(:load-file FILENAME)` Load the named file.

`(:interpret STRING)` Interpret `STRING` at the Idris REPL, returning a highlighted result.

`(:repl-completions STRING)` Return the result of tab-completing `STRING` as a REPL command.

`(:type-of STRING)` Return the type of the name, written with Idris syntax in the `STRING`. The reply may contain highlighting information.

`(:case-split LINE NAME)` Generate a case-split for the pattern variable `NAME` on program line `LINE`. The pattern-match cases to be substituted are returned as a string with no highlighting.

`(:add-clause LINE NAME)` Generate an initial pattern-match clause for the function declared as `NAME` on program line `LINE`. The initial clause is returned as a string with no highlighting.

`(:add-proof-clause LINE NAME)` Add a clause driven by the `<==` syntax.

`(:add-missing LINE NAME)` Add the missing cases discovered by totality checking the function declared as `NAME` on program line `LINE`. The missing clauses are returned as a string with no highlighting.

`(:make-with LINE NAME)` Create a with-rule pattern match template for the clause of function `NAME` on line `LINE`. The new code is returned with no highlighting.

`(:make-case LINE NAME)` Create a case pattern match template for the clause of function `NAME` on line `LINE`. The new code is returned with no highlighting.

`(:make-lemma LINE NAME)` Create a top level function with a type which solves the hole named `NAME` on line `LINE`.

`(:proof-search LINE NAME HINTS)` Attempt to fill out the holes on `LINE` named `NAME` by proof search. `HINTS` is a possibly-empty list of additional things to try while searching.

`(:docs-for NAME)` Look up the documentation for `NAME`, and return it as a highlighted string.

`(:metavariables WIDTH)` List the currently-active holes, with their types pretty-printed with `WIDTH` columns.

`(:who-calls NAME)` Get a list of callers of `NAME`.

`(:calls-who NAME)` Get a list of callees of `NAME`.

`(:browse-namespace NAMESPACE)` Return the contents of `NAMESPACE`, like `:browse` at the

command-line REPL.

(:normalise-term TM) Return a highlighted string consisting of the results of normalising the serialised term `TM` (which would previously have been sent as the `tt-term` property of a string).

(:show-term-implicits TM) Return a highlighted string consisting of the results of making all arguments in serialised term `TM` (which would previously have been sent as the `tt-term` property of a string) explicit.

(:hide-term-implicits TM) Return a highlighted string consisting of the results of making all arguments in serialised term `TM` (which would previously have been sent as the `tt-term` property of a string) follow their usual implicitness setting.

(:elaborate-term TM) Return a highlighted string consisting of the the core language term corresponding to serialised term `TM` (which would previously have been sent as the `tt-term` property of a string).

(:print-definition NAME) Return the definition of `NAME` as a highlighted string.

(:repl-completions NAME) Search names, types and documentations which contain `NAME`.

(:version UID) Return the version information of the Idris compiler.

Possible replies include a normal final reply::

```
(:return (:ok SEXP [HIGHLIGHTING]))
(:return (:error String [HIGHLIGHTING]))
```

A normal intermediate reply::

```
(:output (:ok SEXP [HIGHLIGHTING]))
(:output (:error String [HIGHLIGHTING]))
```

Informational and/or abnormal replies::

```
(:write-string String)
(:set-prompt String)
(:warning (FilePath (LINE COL) (LINE COL) String [HIGHLIGHTING]))
```

Proof mode replies::

```
(:start-proof-mode)
(:write-proof-state [String] [HIGHLIGHTING])
(:end-proof-mode)
(:write-goal String)
```

Output Highlighting

Idris mode supports highlighting the output from Idris. In reality, this highlighting is controlled by the Idris compiler. Some of the return forms from Idris support an optional extra parameter: a list mapping spans of text to metadata about that text. Clients can then use this list both to highlight the displayed output and to enable richer interaction by having more metadata present. For example, the Emacs mode allows right-clicking identifiers to get a menu with access to documentation and type signatures.

A particular semantic span is a three element list. The first element of the list is the index at which the span begins, the second element is the number of characters included in the span, and the third is the semantic data itself. The semantic data is a list of lists. The head of each list is a key that denotes what kind of metadata is in the list, and the tail is the metadata itself.

The following keys are available:

`name` gives a reference to the fully-qualified Idris name

`implicit` provides a Boolean value that is True if the region is the name of an implicit argument

`decor` describes the category of a token, which can be `type`, `function`, `data`, `keyword`, or `bound`.

`source-loc` states that the region refers to a source code location. Its body is a collection of key-value pairs, with the following possibilities:

`filename` provides the filename

`start` provides the line and column that the source location starts at as a two-element tail

`end` provides the line and column that the source location ends at as a two-element tail

`text-formatting` provides an attribute of formatted text. This is for use with natural-language text, not code, and is presently emitted only from inline documentation. The potential values are `bold`, `italic`, and `underline`.

`link-href` provides a URL that the corresponding text is a link to.

`quasiquotation` states that the region is quasiquoted.

`antiquotation` states that the region is antiquoted.

`tt-term` A serialised representation of the Idris core term corresponding to the region of text.

Source Code Highlighting

Idris supports instructing editors how to colour their code. When elaborating source code or REPL input, Idris will locate regions of the source code corresponding to names, and emit information about these names using the same metadata as output highlighting.

These messages will arrive as replies to the command that caused elaboration to occur, such as `:load-file` or `:interpret`. They have the format::

```
(:output (:ok (:highlight-source POSNS)))
```

where `POSNS` is a list of positions to highlight. Each of these is a two-element list whose first element is a position (encoded as for the `source-loc` property above) and whose second element is highlighting metadata in the same format used for output.

Semantic Highlighting & Pretty Printing

Since `v0.9.18` Idris comes with support for semantic highlighting. When using the `REPL` or IDE support, Idris will highlight your code accordingly to its meaning within the Idris structure. A precursor to semantic highlighting support is the pretty printing of definitions to console, LaTeX, or HTML.

The default styling scheme used was inspired by Conor McBride's own set of stylings, informally known as *Conor Colours*.

Legend

The concepts and their default stylings are as follows:

Idris Term	HTML	LaTeX	IDE/REPL
Bound Variable	Purple	Magenta	
Keyword	Bold	Underlined	
Function	Green	Green	
Type	Blue	Blue	
Data	Red	Red	
Implicit	Italic Purple	Italic Magenta	

Pretty Printing

Idris also supports the pretty printing of code to HTML and LaTeX using the commands:

- `:pp <latex|html> <width> <function name>`
- `:pprint <latex|html> <width> <function name>`

Customisation

If you are not happy with the colours used, the VIM and Emacs editor support allows for customisation of the colours. When pretty printing Idris code as LaTeX and HTML, commands and a CSS style are provided. The colours used by the REPL can be customised through the initialisation script.

Further Information

Please also see the Idris Extras project for links to editor support, and pre-made style files for LaTeX and HTML.

DEPRECATED: Tactics and Theorem Proving

> **Warning:** The interactive theorem-proving interface documented here has been deprecated in favor of *Elaborator Reflection* (page 197).

Idris supports interactive theorem proving, and the analyse of context through holes. To list all unproven holes, use the command `:m`. This will display their qualified names and the expected types. To interactively prove a holes, use the command `:p name` where `name` is the hole. Once the proof is complete, the command `:a` will append it to the current module.

Once in the interactive prover, the following commands are available:

Basic commands

- `:q` - Quits the prover (gives up on proving current lemma).
- `:abandon` - Same as :q

- `:state` - Displays the current state of the proof.
- `:term` - Displays the current proof term complete with its yet-to-be-filled holes (is only really useful for debugging).
- `:undo` - Undoes the last tactic.
- `:qed` - Once the interactive theorem prover tells you "No more goals," you get to type this in celebration! (Completes the proof and exits the prover)

Commonly Used Tactics

Compute

- `compute` - Normalises all terms in the goal (note: does not normalise assumptions)

```
----------                  Goal:                  ----------
 (Vect (S (S Z + (S Z) + (S n))) Nat) -> Vect (S (S (S (S n)))) Nat
-lemma> compute
----------                  Goal:                  ----------
 (Vect (S (S (S (S n)))) Nat) -> Vect (S (S (S (S n)))) Nat
-lemma>
```

Exact

- `exact` - Provide a term of the goal type directly.

```
----------                  Goal:                  ----------
 Nat
-lemma> exact Z
lemma: No more goals.
-lemma>
```

Refine

- `refine` - Use a name to refine the goal. If the name needs arguments, introduce them as new goals.

Trivial

- `trivial` - Satisfies the goal using an assumption that matches its type.

```
----------              Assumptions:              ----------
 value : Nat
----------                  Goal:                  ----------
 Nat
-lemma> trivial
lemma: No more goals.
-lemma>
```

Intro

- `intro` - If your goal is an arrow, turns the left term into an assumption.

```
----------                  Goal:                  ----------
 Nat -> Nat -> Nat
-lemma> intro
----------              Assumptions:               ----------
 n : Nat
----------                  Goal:                  ----------
 Nat -> Nat
-lemma>
```

You can also supply your own name for the assumption:

```
----------                  Goal:                  ----------
Nat -> Nat -> Nat
-lemma> intro number
----------              Assumptions:               ----------
 number : Nat
----------                  Goal:                  ----------
Nat -> Nat
```

Intros

- `intros` - Exactly like intro, but it operates on all left terms at once.

```
----------                  Goal:                  ----------
 Nat -> Nat -> Nat
-lemma> intros
----------              Assumptions:               ----------
 n : Nat
 m : Nat
----------                  Goal:                  ----------
 Nat
-lemma>
```

let

- `let` - Introduces a new assumption; you may use current assumptions to define the new one.

```
----------              Assumptions:               ----------
 n : Nat
----------                  Goal:                  ----------
 BigInt
-lemma> let x = toIntegerNat n
----------              Assumptions:               ----------
 n : Nat
  x = toIntegerNat n: BigInt
----------                  Goal:                  ----------
 BigInt
-lemma>
```

rewrite

- `rewrite` - Takes an expression with an equality type (x = y), and replaces all instances of x in the goal with y. Is often useful in combination with 'sym'.

```
----------              Assumptions:              ----------
 n : Nat
 a : Type
 value : Vect Z a
----------                 Goal:                  ----------
 Vect (mult n Z) a
-lemma> rewrite sym (multZeroRightZero n)
----------              Assumptions:              ----------
 n : Nat
 a : Type
 value : Vect Z a
----------                 Goal:                  ----------
 Vect Z a
-lemma>
```

induction

- `induction` - (`Note that this is still experimental` and you may get strange results and error messages. We are aware of these and will finish the implementation eventually!) Prove the goal by induction. Each constructor of the datatype becomes a goal. Constructors with recursive arguments become induction steps, while simple constructors become base cases. Note that this only works for datatypes that have eliminators: a datatype definition must have the `%elim` modifier.

sourceLocation

- `sourceLocation` - Solve the current goal with information about the location in the source code where the tactic was invoked. This is mostly for embedded DSLs and programmer tools like assertions that need to know where they are called. See `Language.Reflection.SourceLocation` for more information.

Less commonly-used tactics

- `applyTactic` - Apply a user-defined tactic. This should be a function of type `List (TTName, Binder TT) -> TT -> Tactic`, where the first argument represents the proof context and the second represents the goal. If your tactic will produce a proof term directly, use the `Exact` constructor from `Tactic`.
- `attack` - ?
- `equiv` - Replaces the goal with a new one that is convertible with the old one
- `fill` - ?
- `focus` - ?
- `mrefine` - Refining by matching against a type
- `reflect` - ?
- `solve` - Takes a guess with the correct type and fills a hole with it, closing a proof obligation. This happens automatically in the interactive prover, so `solve` is really only relevant in tactic scripts used for helping implicit argument resolution.
- `try` - ?

The Idris REPL

Idris comes with a `REPL`.

Evaluation

Being a fully dependently typed language, Idris has two phases where it evaluates things, compile-time and run-time. At compile-time it will only evaluate things which it knows to be total (i.e. terminating and covering all possible inputs) in order to keep type checking decidable. The compile-time evaluator is part of the Idris kernel, and is implemented in Haskell using a HOAS (higher order abstract syntax) style representation of values. Since everything is known to have a normal form here, the evaluation strategy doesn't actually matter because either way it will get the same answer, and in practice it will do whatever the Haskell run-time system chooses to do.

The REPL, for convenience, uses the compile-time notion of evaluation. As well as being easier to implement (because we have the evaluator available) this can be very useful to show how terms evaluate in the type checker. So you can see the difference between:

```
Idris> \n, m => (S n) + m
\n => \m => S (plus n m) : Nat -> Nat -> Nat

Idris> \n, m => n + (S m)
\n => \m => plus n (S m) : Nat -> Nat -> Nat
```

Customisation

Idris supports initialisation scripts.

Initialisation scripts

When the Idris REPL starts up, it will attempt to open the file repl/init in Idris's application data directory. The application data directory is the result of the Haskell function call `getAppUserDataDirectory "idris"`, which on most Unix-like systems will return $HOME/.idris and on various versions of Windows will return paths such as `C:/Documents And Settings/user/Application Data/appName`.

The file repl/init is a newline-separate list of REPL commands. Not all commands are supported in initialisation scripts — only the subset that will not interfere with the normal operation of the REPL. In particular, setting colours, display options such as showing implicits, and log levels are supported.

Example initialisation script

```
:colour prompt white italic bold
:colour implicit magenta italic
```

The `REPL` Commands

The current set of supported commands are:

Command	Arguments	Purpose
<expr>		Evaluate an expression
:t :type	<expr>	Check the type of an expression
:core	<expr>	View the core language representation of a term
:miss :missing	<name>	Show missing clauses
:doc	<name>	Show internal documentation
:mkdoc	<namespace>	Generate IdrisDoc for namespace(s) and dependencies
:apropos	[<package list>] <name>	Search names, types, and documentation
:s :search	[<package list>] <expr>	Search for values by type
:wc :whocalls	<name>	List the callers of some name
:cw :callswho	<name>	List the callees of some name
:browse	<namespace>	List the contents of some namespace
:total	<name>	Check the totality of a name
:r :reload		Reload current file
:l :load	<filename>	Load a new file
:cd	<filename>	Change working directory
:module	<module>	Import an extra module
:e :edit		Edit current file using $EDITOR or $VISUAL
:m :metavars		Show remaining proof obligations (holes)
:p :prove	<hole>	Prove a hole
:a :addproof	<name>	Add proof to source file
:rmproof	<name>	Remove proof from proof stack
:showproof	<name>	Show proof
:proofs		Show available proofs
:x	<expr>	Execute IO actions resulting from an expression using the interpre
:c :compile	<filename>	Compile to an executable [codegen] <filename>
:exec :execute	[<expr>]	Compile to an executable and run
:dynamic	<filename>	Dynamically load a C library (similar to %dynamic)
:dynamic		List dynamically loaded C libraries
:? :h :help		Display this help text
:set	<option>	Set an option (errorcontext, showimplicits, originalerrors, autosolv
:unset	<option>	Unset an option
:color :colour	<option>	Turn REPL colours on or off; set a specific colour
:consolewidth	auto|infinite|<number>	Set the width of the console
:printerdepth	<number-or-blank>	Set the maximum pretty-printing depth, or infinite if nothing spec
:q :quit		Exit the Idris system
:w :warranty		Displays warranty information
:let	(<top-level-declaration>)...	Evaluate a declaration, such as a function definition, instance imp
:unlet :undefine	(<name>)...	Remove the listed repl definitions, or all repl definitions if no nam
:printdef	<name>	Show the definition of a function
:pp :pprint	<option> <number> <name>	Pretty prints an Idris function in either LaTeX or HTML and for

Using the REPL

Getting help

The command `:help` (or `:h` or `:?`) prints a short summary of the available commands.

Quitting Idris

If you would like to leave Idris, simply use `:q` or `:quit`.

Evaluating expressions

To evaluate an expression, simply type it. If Idris is unable to infer the type, it can be helpful to use the operator `the` to manually provide one, as Idris's syntax does not allow for direct type annotations. Examples of `the` include:

```
Idris> the Nat 4
4 : Nat
Idris> the Int 4
4 : Int
Idris> the (List Nat) [1,2]
[1,2] : List Nat
Idris> the (Vect _ Nat) [1,2]
[1,2] : Vect 2 Nat
```

This may not work in cases where the expression still involves ambiguous names. The name can be disambiguated by using the `with` keyword:

```
Idris> sum [1,2,3]
When elaborating an application of function Prelude.Foldable.sum:
        Can't disambiguate name: Prelude.List.::,
                                 Prelude.Stream.::,
                                 Prelude.Vect.::
Idris> with List sum [1,2,3]
6 : Integer
```

Adding let bindings

To add a let binding to the REPL, use `:let`. It's likely you'll also need to provide a type annotation. `:let` also works for other declarations as well, such as `data`.

```
Idris> :let x : String; x = "hello"
Idris> x
"hello" : String
Idris> :let y = 10
Idris> y
10 : Integer
Idris> :let data Foo : Type where Bar : Foo
Idris> Bar
Bar : Foo
```

Getting type information

To ask Idris for the type of some expression, use the `:t` command. Additionally, if used with an overloaded name, Idris will provide all overloadings and their types. To ask for the type of an infix operator, surround it in parentheses.

```
Idris> :t "foo"
"foo" : String
Idris> :t plus
Prelude.Nat.plus : Nat -> Nat -> Nat
Idris> :t (++)
Builtins.++ : String -> String -> String
Prelude.List.++ : (List a) -> (List a) -> List a
Prelude.Vect.++ : (Vect m a) -> (Vect n a) -> Vect (m + n) a
Idris> :t plus 4
plus (Builtins.fromInteger 4) : Nat -> Nat
```

You can also ask for basic information about interfaces with `:doc`:

```
Idris> :doc Monad
Interface Monad

Parameters:
    m

Methods:
    (>>=) : Monad m => m a -> (a -> m b) -> m b

        infixl 5

Instances:
    Monad id
    Monad PrimIO
    Monad IO
    Monad Maybe

...
```

Other documentation is also available from `:doc`:

```
Idris> :doc (+)
Prelude.Interfaces.(+) : Num ty => ty -> ty -> ty

infixl 8

The function is Total


Idris> :doc Vect
Data type Prelude.Vect.Vect : Nat -> Type -> Type

Arguments:
        Nat
        Type

Constructors:

Prelude.Vect.Nil : (a : Type) -> Vect 0 a


Prelude.Vect.:: : (a : Type) -> (n : Nat) -> a -> (Vect n a) -> Vect (S n) a

infixr 7

Arguments:
        a
        Vect n a


Idris> :doc Monad
Interface Monad

Parameters:
    m

Methods:
    (>>=) : Monad m => m a -> (a -> m b) -> m b
        Also called bind.
        infixl 5

        The function is Total
```

```
    join : Monad m => m (m a) -> m a
        Also called flatten or mu

        The function is Total
Implementations:
    Monad (IO' ffi)
    Monad Stream
    Monad Provider
    Monad Elab
    Monad PrimIO
    Monad Maybe
    Monad (Either e)
    Monad List
```

Finding things

The command `:apropos` searches names, types, and documentation for some string, and prints the results. For example:

```
Idris> :apropos eq
eqPtr : Ptr -> Ptr -> IO Bool


eqSucc : (left : Nat) -> (right : Nat) -> (left = right) -> S left = S right
S preserves equality

lemma_both_neq : ((x = x') -> _|_) -> ((y = y') -> _|_) -> ((x, y) = (x', y')) -> _|_


lemma_fst_neq_snd_eq : ((x = x') -> _|_) -> (y = y') -> ((x, y) = (x', y)) -> _|_


lemma_snd_neq : (x = x) -> ((y = y') -> _|_) -> ((x, y) = (x, y')) -> _|_


lemma_x_eq_xs_neq : (x = y) -> ((xs = ys) -> _|_) -> (x :: xs = y :: ys) -> _|_


lemma_x_neq_xs_eq : ((x = y) -> _|_) -> (xs = ys) -> (x :: xs = y :: ys) -> _|_


lemma_x_neq_xs_neq : ((x = y) -> _|_) -> ((xs = ys) -> _|_) -> (x :: xs = y :: ys) -> _|_


prim__eqB16 : Bits16 -> Bits16 -> Int

prim__eqB16x8 : Bits16x8 -> Bits16x8 -> Bits16x8

prim__eqB32 : Bits32 -> Bits32 -> Int

prim__eqB32x4 : Bits32x4 -> Bits32x4 -> Bits32x4

prim__eqB64 : Bits64 -> Bits64 -> Int

prim__eqB64x2 : Bits64x2 -> Bits64x2 -> Bits64x2

prim__eqB8 : Bits8 -> Bits8 -> Int

prim__eqB8x16 : Bits8x16 -> Bits8x16 -> Bits8x16
```

```
prim__eqBigInt : Integer -> Integer -> Int

prim__eqChar : Char -> Char -> Int

prim__eqFloat : Double -> Double -> Int

prim__eqInt : Int -> Int -> Int

prim__eqString : String -> String -> Int

prim__syntactic_eq : (a : Type) -> (b : Type) -> (x : a) -> (y : b) -> Maybe (x = y)

sequence : Traversable t => Applicative f => (t (f a)) -> f (t a)


sequence_ : Foldable t => Applicative f => (t (f a)) -> f ()


Eq : Type -> Type
The Eq interface defines inequality and equality.

GTE : Nat -> Nat -> Type
Greater than or equal to

LTE : Nat -> Nat -> Type
Proofs that n is less than or equal to m

gte : Nat -> Nat -> Bool
Boolean test than one Nat is greater than or equal to another

lte : Nat -> Nat -> Bool
Boolean test than one Nat is less than or equal to another

ord : Char -> Int
Convert the number to its ASCII equivalent.

replace : (x = y) -> (P x) -> P y
Perform substitution in a term according to some equality.

sym : (l = r) -> r = l
Symmetry of propositional equality

trans : (a = b) -> (b = c) -> a = c
Transitivity of propositional equality
```

`:search` does a type-based search, in the spirit of Hoogle. See Type-directed search (:search) for more details. Here is an example:

```
Idris> :search a -> b -> a
= Prelude.Basics.const : a -> b -> a
Constant function. Ignores its second argument.

= assert_smaller : a -> b -> b
Assert to the totality checker than y is always structurally
smaller than x (which is typically a pattern argument)

> malloc : Int -> a -> a


> Prelude.pow : Num a => a -> Nat -> a
```

```
> Prelude.Interfaces.(*) : Num a => a -> a -> a


> Prelude.Interfaces.(+) : Num a => a -> a -> a
... (More results)
```

`:search` can also look for dependent types:

```
Idris> :search plus (S n) n = plus n (S n)
< Prelude.Nat.plusSuccRightSucc : (left : Nat) ->
                                  (right : Nat) ->
                                  S (left + right) = left + S right
```

Loading and reloading Idris code

The command `:l File.idr` will load File.idr into the currently-running REPL, and `:r` will reload the last file that was loaded.

Totality

All Idris definitions are checked for totality. The command `:total <NAME>` will display the result of that check. If a definition is not total, this may be due to an incomplete pattern match. If that is the case, `:missing` or `:miss` will display the missing cases.

Editing files

The command `:e` launches your default editor on the current module. After control returns to Idris, the file is reloaded.

Invoking the compiler

The current module can be compiled to an executable using the command `:c <FILENAME>` or `:compile <FILENAME>`. This command allows to specify codegen, so for example JavaScript can be generated using `:c javascript <FILENAME>`. The `:exec` command will compile the program to a temporary file and run the resulting executable.

IO actions

Unlike GHCI, the Idris REPL is not inside of an implicit IO monad. This means that a special command must be used to execute IO actions. `:x tm` will execute the IO action `tm` in an Idris interpreter.

Dynamically loading C libraries

Sometimes, an Idris program will depend on external libraries written in C. In order to use these libraries from the Idris interpreter, they must first be dynamically loaded. This is achieved through the `%dynamic <LIB>` directive in Idris source files or through the `:dynamic <LIB>` command at the REPL. The current set of dynamically loaded libraries can be viewed by executing `:dynamic` with no arguments. These libraries are available through the Idris FFI in *type providers* (page **??**) and `:exec`.

Colours

Idris terms are available in amazing colour! By default, the Idris REPL uses colour to distinguish between data constructors, types or type constructors, operators, bound variables, and implicit arguments. This feature is available on all POSIX-like systems, and there are plans to allow it to work on Windows as well.

If you do not like the default colours, they can be turned off using the command

```
:colour off
```

and, when boredom strikes, they can be re-enabled using the command

```
:colour on
```

To modify a colour, use the command

```
:colour <CATEGORY> <OPTIONS>
```

where `<CATEGORY` is one of `keyword`, `boundvar`, `implicit`, `function`, `type`, `data`, or `prompt`, and is a space-separated list drawn from the colours and the font options. The available colours are `default`, `black`, `yellow`, `cyan`, `red`, `blue`, `white`, `green`, and `magenta`. If more than one colour is specified, the last one takes precedence. The available options are `dull` and `vivid`, `bold` and `nobold`, `italic` and `noitalic`, `underline` and `nounderline`, forming pairs of opposites. The colour `default` refers to your terminal's default colour.

The colours used at startup can be changed using REPL initialisation scripts.

Colour can be disabled at startup by the `--nocolour` command-line option.

Compilation, Logging, and Reporting

This section provides information about the Idris compilation process, and provides details over how you can follow the process through logging.

Compilation Process

Idris follows the following compilation process:

1. Parsing
2. Type Checking
   (a) Elaboration
   (b) Coverage
   (c) Unification
   (d) Totality Checking
   (e) Erasure
3. Code Generation
   (a) Defunctionalisation

(b) Inlining

(c) Resolving variables

(d) Code Generation

Type Checking Only

With Idris you can ask it to terminate the compilation process after type checking has completed. This is achieved through use of either:

- The command line options
  - `--check` for files
  - `--checkpkg` for packages
- The REPL command: `:check`

Use of this option will still result in the generation of the Idris binary `.ibc` files, and is suitable if you do not wish to generate code from one of the supported backends.

Reporting Compilation Process

During compilation the reporting of Idris' progress can be controlled by setting a verbosity level.

- `-V`, or alternatively `--verbose` and `--V0`, will report which file Idris is currently type checking.
- `--V1` will additionally report: Parsing, IBC Generation, and Code Generation.
- `--V2` will additionally report: Totality Checking, Universe Checking, and the individual steps prior to code generation.

By default Idris' progress reporting is set to quiet-`-q`, or `--quiet`.

Logging Internal Operation

For those that develop on the Idris compiler, the internal operation of Idris is captured using a category based logger. Currently, the logging infrastructure has support for the following categories:

- Parser
- Elaborator
- Code generation
- Erasure
- Coverage Checking
- IBC generation

These categories are specified using the command-line option: `--logging-categories CATS`, where `CATS` is a quoted colon seperated string of the categories you want to see. By default if this option is not specified all categories are allowed. Sub-categories have yet to be defined but will be in the future, especially for the elaborator.

Further, the verbosity of logging can be controlled by specifying a logging level between: 1 to 10 using the command-line option: `--log <level>`.

- Level 0: Show no logging output. Default level
- Level 1: High level details of the compilation process.
- Level 2: Provides details of the coverage checking, and further details the elaboration process specifically: Interface, Clauses, Data, Term, and Types,
- Level 3: Provides details of compilation of the IRTS, erasure, parsing, case splitting, and further details elaboration of: Implementations, Providers, and Values.
- Level 4: Provides further details on: Erasure, Coverage Checking, Case splitting, and elaboration of clauses.
- Level 5: Provides details on the prover, and further details elaboration (adding declarations) and compilation of the IRTS.
- Level 6: Further details elaboration and coverage checking.
- Level 7:
- Level 8:
- Level 9:
- Level 10: Further details elaboration.

Environment Variables

Several paths set by default within the Idris compiler can be overridden through environment variables. The provided variables are:

- *IDRIS_CC* Change the *C* compiler used by the *C* backend.
- *IDRIS_CFLAGS* Change the *C* flags passed to the *C* compiler.
- *TARGET* Change the target directory i.e. *data dir* where Idris installs files when installing using Cabal/Stack.
- *IDRIS_LIBRARY_PATH* Change the location of where installed packages are found/installed.
- *IDRIS_DOC_PATH* Change the location of where generated idrisdoc for packages are installed.

Note: In versions of Idris prior to 0.12.3 the environment variables *IDRIS_LIBRARY_PATH* and *TARGET* were both used to affect the installation of single packages and direct where Idris installed its data. The meaning of these variables has changed, and command line options are preferred when changing where individual packages are installed.

The CLI option *–ibcsubdir* can be used to direct where generated IBC files are placed. However, this means Idris will install files in a non-standard location separate from the rest of the installed packages. The CLI option *–idrispath <dir>* allows you to add a directory to the library search path; this option can be used multiple times and can be shortened to *-i <dir>*. Similary, the *–sourcepath <dir>* option can be used to add directories to the source search path. There is no shortened version for this option as *-s* is a reserved flag.

Further, Idris also supports options to augment the paths used, and pass options to the code generator

backend. The option *–cg-opt <ARG>* can be used to pass options to the code generator. The format of *<ARG>* is dependent on the selected backend.

Idris' Internals

Note: this is still a fairly raw set of notes taken by David Christiansen at Edwin's presentation at the 2013 Idris Developers Meeting. They're in the process of turning into a useful guide - feel free to contribute.

This document assumes that you are already familiar with Idris. It is intended for those who want to work on the internals.

People looking to develop new back ends may want to look at [[Idris back end IRs|Idris-back-end-IRs]]

Core/TT.hs

Idris is compiled to a simple, explicit core language. This core language is called TT because it looks a bit like a Π. It's a minimal language, with a locally nameless representation. That is, local variables are represented with de Bruijn indices and globally-defined constants are represented with names.

The `TT` datatype uses a trick that is common in the Idris code: it is polymorphic over the type of names stored in it, and it derives `Functor`. This allows `fmap` to be used as a general-purpose traversal.

There is a general construction for binders, used for λ, Π, and let-bindings. These are distinguished using a `BinderType`.

During compilation, some terms (especially types) will be erased. This is represented using the `Erased` constructor of `TT`. A handy trick when generating TT terms is to insert `Erased` where a term is uniquely determined, as the typechecker will fill it out.

The constructor `Proj` is a result of the optimizer. It is used to extract a specific constructor argument, in a more economical way than defining a new pattern-matching operation.

The datatype `Raw` represents terms that have not yet been typechecked. The typechecker converts a `Raw` to a `TT` if it can.

Core/CaseTree.hs

Case trees are used to represent top-level pattern-matching definitions in the TT language.

Just as with the `TT` datatype, the `deriving Functor` trick is used with `SC` and `CaseAlt` to get GHC to generate a function for mapping over contained terms.

Constructor cases (`ConCase` in `CaseAlt`) refer to numbered constructors. Every constructor is numbered 0,1,2,.... At this stage in the compiler, the tags are datatype-local. After defunctionalization, however, they are made globally unique.

The `n+1` patterns (`SucCase`) and hacky-seeming things are to make code fast – please ask before "cleaning up" the representation.

Core/Evaluate.hs

This module contains the main evaluator for Idris. The evaluator is used both at the REPL and during type checking, where normalised terms need to be compared for equality.

A key datatype in the evaluator is a *context*. Contexts are mappings from global names to their values, but they are organized to make type-directed disambiguation quick. In particular, the main part of a name that a user might type is used as the key, and its values are maps from namespaces to actual values.

The datatype `Def` represents a definition in the global context. All global names map to this structure.

`Type` and `Term` are both synonyms for `TT`.

Datatypes are represented by a `TyDecl` with the appropriate `NameType`. A `Function` is a global constant term with an annotated type, `Operator` represents primitives implemented in Haskell, and `CaseOp` represents ordinary pattern-matching definitions. `CaseOp` has four versions for different purposes, and all are saved because that's easiest.

`CaseInfo`: the `tc_dictionary` is because it's a type class dictionary which makes totality checking easier.

The `normalise*` functions give different behaviors - but `normalise` is the most common.

`normaliseC` - "resolved" means with names converted to de Bruijn indices as appropriate.

`normaliseAll` - reduce everything, even if it's non-total

`normaliseTrace` - special-purpose for debugging

`simplify` - reduce the things that are small - the list argument is the things to not reduce.

Core/Typecheck.hs

Standard stuff. Hopefully no changes are necessary.

Core/Elaborate.hs

Idris definitions are elaborated one by one and turned into the corresponding TT. This is done with a tactic language as an EDSL in the Elab monad (or Elab' when there's a custom state).

Lots of plumbing for errors.

All elaboration is relative to a global context.

The string in the pair returned by elaborate is log information.

See JFP paper, but the names don't necessarily map to each other. The paper is the "idealized version" without logging, additional state, etc.

All the tactics take Raws, typechecking happens there.

claim (x : t) assumes a new x : t.

PLEASE TIDY THINGS UP!

proofSearch flag to try' is whether the failure came from a human (so fail) or from a machine (so continue)

Idris-level syntax for providing alternatives explicitly: (| x, y, z |) try x, y, z in order, and take the first that succeeds.

Core/ProofState.hs

Core/Unify.hs

Deals with unification. Unification can reply with: - this works - this can never work - this will work if these other unification problems work out (eg unifying f x with 1)

match_unify: same thing as unification except it's just matching name against name, term against term. x + y matches to 0 + y with x = 0. Used for <== syntax as well as type class resolution.

Idris/AbsSyntaxTree.hs

PTerm is the datatype of Idris syntax. P is for Program. Each PTerm turns into a TT term by applying a series of tactics.

IState is the major interpreter state. The global context is the tt_ctxt field.

Ctxt maps possibly ambiguous names to their referents.

Idris/ElabDecls.hs

This is where the actual elaboration from PTerm to TT happens.

Idris/ElabTerm.hs

build is the function that creates a Raw. All the "junk" is to deal with things like metavars and so forth. It has to remember what names are still to be defined, and it doesn't yet know the type (filled in by unificaiton later). Also case expressions have to turn into top-level functions.

resolveTC is type class resolution.

Core Language Features

- Full-spectrum dependent types
- Strict evaluation (plus `Lazy : Type -> Type` type constructor for explicit laziness)
- Lambda, Pi (forall), Let bindings
- Pattern matching definitions
- Export modifiers `public`, `abstract`, `private`
- Function options `partial`, `total`
- `where` clauses
- "magic with"
- Implicit arguments (in top level types)
- "Bound" implicit arguments `{n : Nat} -> {a : Type} -> Vect n a`

- "Unbound" implicit arguments — `Vect n a` is equivalent to the above in a type, `n` and `a` are implicitly bound. This applies to names beginning with a lower case letter in an argument position.
- 'Tactic' implicit arguments, which are solved by running a tactic script or giving a default argument, rather than by unification.
- Unit type `()`, empty type `Void`
- Tuples (desugaring to nested pairs)
- Dependent pair syntax `(x : T ** P x)` (there exists an `x` of type `T` such that `P x`)
- Inline `case` expressions
- Heterogeneous equality
- `do` notation
- Idiom brackets
- Interfaces (like type classes), supporting default methods and dependencies between methods
- `rewrite` prf `in` expr
- Metavariables
- Inline proof/tactic scripts
- Implicit coercion
- `codata`
- Also `Inf : Type -> Type` type constructor for mixed data/codata. In fact `codata` is implemented by putting recursive arguments under `Inf`.
- `syntax` rules for defining pattern and term syntactic sugar
- these are used in the standard library to define `if ... then ... else` expressions and an Agda-style preorder reasoning syntax.
- Uniqueness typing using the `UniqueType` universe.
- Partial evaluation by `%static` argument annotations.
- Error message reflection
- Eliminators
- Label types `'name`
- `%logging n`
- `%unifyLog`

Language Extensions

Type Providers

Idris type providers are a way to get the type system to reflect observations about the world outside of Idris. Similarly to F# type providers, they cause effectful computations to run during type checking, returning information that the type checker can use when checking the rest of the program. While F# type providers are based on code generation, Idris type providers use only the ordinary execution semantics of Idris to generate the information.

A type provider is simply a term of type `IO (Provider t)`, where `Provider` is a data type with constructors for a successful result and an error. The type `t` can be either `Type` (the type of types) or a concrete type. Then, a type provider `p` is invoked using the syntax `%provide (x : t) with p`. When the type checker encounters this line, the IO action `p` is executed. Then, the resulting term is extracted from the IO monad. If it is `Provide y` for some `y : t`, then `x` is bound to `y` for the remainder of typechecking and in the compiled code. If execution fails, a generic error is reported and type checking terminates. If the resulting term is `Error e` for some string `e`, then type checking fails and the error `e` is reported to the user.

Example Idris type providers can be seen at this repository. More detailed descriptions are available in David Christiansen's WGP '13 paper and M.Sc. thesis.

Elaborator Reflection

The Idris elaborator is responsible for converting high-level Idris code into the core language. It is implemented as a kind of embedded tactic language in Haskell, where tactic scripts are written in an *elaboration monad* that provides error handling and a proof state. For details, see Edwin Brady's 2013 paper in the Journal of Functional Programming.

Elaborator reflection makes the elaboration type as well as a selection of its tactics available to Idris code. This means that metaprograms written in Idris can have complete control over the elaboration process, generating arbitrary code, and they have access to all of the facilities available in the elaborator, such as higher-order unification, type checking, and emitting auxiliary definitions.

The Elaborator State

The elaborator state contains information about the ongoing elaboration process. In particular, it contains a *goal type*, which is to be filled by an under-construction *proof term*. The proof term can contain *holes*, each of which has a scope in which it is valid and a type. Some holes may additionally contain *guesses*, which can be substituted in the scope of the hole. The holes are tracked in a *hole queue*, and one of them is *focused*. In addition to the goal type, proof term, and holes, the elaborator state contains a collection of unsolved unification problems that can affect elaboration.

The elaborator state is not directly available to Idris programs. Instead, it is modified through the use of *tactics*, which are operations that affect the elaborator state. A tactic that returns a value of type `a`, potentially modifying the elaborator state, has type `Elab a`. The default tactics are all in the namespace `Language.Reflection.Elab.Tactics`.

Running Elaborator Scripts

On their own, tactics have no effect. The meta-operation `%runElab script` runs `script` in the current elaboration context. For example, the following script constructs the identity function at type `Nat`:

```
idNat : Nat -> Nat
idNat = %runElab (do intro `{{x}}
```

```
                fill (Var `{{x}})
                solve)
```

On the right-hand side, the Idris elaborator has the goal `Nat -> Nat`. When it encounters the `%runElab` directive, it fulfills this goal by running the provided script. The first tactic, `intro`, constructs a lambda that binds the name `x`. The name argument is optional because a default name can be taken from the function type. Now, the proof term is of the form `\x : Nat => {hole}`. The second tactic, `fill`, fills this hole with a guess, giving the term `\x : Nat => {hole≈x}`. Finally, the `solve` tactic instantiates the guess, giving the result `\x : Nat => x`.

Because elaborator scripts are ordinary Idris expressions, it is also possible to use them in multiple contexts. Note that there is nothing `Nat`-specific about the above script. We can generate identity functions at any concrete type using the same script:

```
mkId : Elab ()
mkId = do intro `{{x}}
          fill (Var `{{x}})
          solve

idNat : Nat -> Nat
idNat = %runElab mkId

idUnit : () -> ()
idUnit = %runElab mkId

idString : String -> String
idString = %runElab mkId
```

Interactively Building Elab Scripts

You can build an `Elab` script interactively at the REPL. Use the command `:metavars`, or `:m` for short, to list the available holes. Then, issue the `:elab <hole>` command at the REPL to enter the elaboration shell.

At the shell, you can enter proof tactics to alter the proof state. You can view the system-provided tactics prior to entering the shell by issuing the REPL command `:browse Language.Reflection.Elab.Tactics`. When you have discharged all goals, you can complete the proof using the `:qed` command and receive in return an elaboration script that fills the hole.

The interactive elaboration shell accepts a limited number of commands, including a subset of the commands understood by the normal Idris REPL as well as some elaboration-specific commands.

General-purpose commands:

- `:eval <EXPR>`, or `:e <EXPR>` for short, evaluates the provided expression and prints the result.
- `:type <EXPR>`, or `:t <EXPR>` for short, prints the provided expression together with its type.
- `:search <TYPE>` searches for definitions having the provided type.
- `:doc <NAME>` searches for definitions with the provided name and prints their documentation.

Commands for viewing the proof state:

- `:state` displays the current state of the term being constructed. It lists both other goals and the current goal.
- `:term` displays the current proof term as well as its yet-to-be-filled holes.

Commands for manipulating the proof state:

- `:undo` undoes the effects of the last tactic.
- `:abandon` gives up on proving the current lemma and quits the elaboration shell.
- `:qed` finishes the script and exits the elaboration shell. The shell will only accept this command once it reports, "No more goals." On exit, it will print out the finished elaboration script for you to copy into your program.

Failure

Some tactics may *fail*. For example, `intro` will fail if the focused hole does not have a function type, `solve` will fail if the current hole does not contain a guess, and `fill` will fail if the term to be filled in has the wrong type. Scripts can also fail explicitly using the `fail` tactic.

To account for failure, there is an `Alternative` implementation for `Elab`. The `<|>` operator first tries the script to its left. If that script fails, any changes that it made to the state are undone and the right argument is executed. If the first argument succeeds, then the second argument is not executed.

Querying the Elaboration State

`Elab` includes operations to query the elaboration state, allowing scripts to use information about their environment to steer the elaboration process. The ordinary Idris bind syntax can be used to propagate this information. For example, a tactic that solves the current goal when it is the unit type might look like this:

```
triv : Elab ()
triv = do compute
          g <- getGoal
          case (snd g) of
            `(() : Type) => do fill `(() : ())
                               solve
            otherGoal => fail [ TermPart otherGoal
                              , TextPart "is not trivial"
                              ]
```

The tactic `compute` normalises the type of its goal with respect to the current context. While not strictly necessary, this allows `triv` to be used in contexts where the triviality of the goal is not immediately apparent. Then, `getGoal` is used, and its result is bound to `g`. Because it returns a pair consisting of the current goal's name and type, we case-split on its second projection. If the goal type turns out to have been the unit type, we fill using the unit constructor and solve the goal. Otherwise, we fail with an error message informing the user that the current goal is not trivial.

Additionally, the elaboration state can be dumped into an error message with the `debug` tactic. A variant, `debugMessage`, allows arbitrary messages to be included with the state, allowing for a kind of "`printf` debugging" of elaboration scripts. The message format used by `debugMessage` is the same for errors produced by the error reflection mechanism, allowing the re-use of the Idris pretty-printer when rendering messages.

Changing the Global Context

`Elab` scripts can modify the global context during execution. Just as the Idris elaborator produces auxiliary definitions to implement features such as `where`-blocks and `case` expressions, user elaboration scripts may need to define functions. Furthermore, this allows `Elab` reflection to be used to implement features

such as interface deriving. The operations `declareType`, `defineFunction`, and `addImplementation` allow `Elab` scripts to modify the global context.

Using Idris's Features

The Idris compiler has a number of ways to automate the construction of terms. On its own, the `Elab` state and its interactions with the unifier allow implicits to be solved using unification. Additional operations use further features of Idris. In particular, `resolveTC` solves the current goal using interface resolution, `search` invokes the proof search mechanism, and `sourceLocation` finds the context in the original file at which the elaboration script is invoked.

Recursive Elaboration

The elaboration mechanism can be invoked recursively using the `runElab` tactic. This tactic takes a goal type and an elaboration script as arguments and runs the script in a fresh lexical environment to create an inhabitant of the provided goal type. This is primarily useful for code generation, particularly for generating pattern-matching clauses, where variable scope needs to be one that isn't the present local context.

Learn More

While this documentation is still incomplete, elaboration reflection works in Idris today. As you wait for the completion of the documentation, the list of built-in tactics can be obtained using the `:browse` command in an Idris REPL or the corresponding feature in one of the graphical IDE clients to explore the `Language.Reflection.Elab.Tactics` namespace. All of the built-in tactics contain documentation strings.

Type Directed Search `:search`

Idris' `:search` command searches for terms according to their approximate type signature (much like Hoogle for Haskell). For example:

```
Idris> :search e -> List e -> List e
= Prelude.List.(::) : a -> List a -> List a
Cons cell

= Prelude.List.intersperse : a -> List a -> List a
Insert some separator between the elements of a list.

> Prelude.List.delete : Eq a => a -> List a -> List a


< assert_smaller : a -> b -> b
Assert to the totality checker than y is always structurally
smaller than x (which is typically a pattern argument)

< Prelude.Basics.const : a -> b -> a
Constant function. Ignores its second argument.
```

The best results are listed first. As we can see, `(::)` and `intersperse` are exact matches; the `=` symbol to the left of those results tells us the types of `(::)` and `intersperse` are effectively the same as the type that was searched.

The next result is `delete`, whose type is more specific than the type that was searched; that's indicated by the `>` symbol. If we had a function with the signature `e -> List e -> List e`, we could have given it the type `Eq a => a -> List a -> List a`, but not necessarily the other way around.

The final two results, `assert_smaller` and `const`, have types more general than the type that was searched, and so they have `<` symbols to their left. For example, `e -> List e -> List e` would be a valid type for `assert_smaller`. The correspondence for `const` is more complicated than any of the four previous results. `:search` shows this result because we could change the order of the arguments! That is, the following definition would be legal:

```
f : e -> List e -> List e
f x xs = const xs x
```

About :search results

:search's functionality is based on the notion of type isomorphism. Informally, two types are isomorphic if we can identify terms of one type exactly with terms of the other. For example, we can consider the types `Nat -> a -> List a` and `a -> Nat -> List a` to be isomorphic, because if we have `f : Nat -> a -> List a`, then `flip f : a -> Nat -> List a`. Similarly, if `g : a -> Nat -> List a`, then `flip g : Nat -> a -> List a`.

With :search, we create a partial order on types; that is, given two types `A` and `B`, we may choose to say that `A <= B`, `A >= B`, or both (in which case we say `A == B`), or neither. For :search, we say that `A >= B` if all of the terms inhabiting `A` correspond to terms of `B`, but it need not necessarily be the case that *all* the terms of `B` correspond to terms of `A`. Here's an example:

```
a -> a            >=          Nat -> Nat
```

The left-hand type has just a single inhabitant, `id`, which corresponds to the term `id {a = Nat}`, which has the right-hand type. However, there are various terms inhabiting the right-hand type (such as `S`) which cannot correspond with terms of type `a -> a`.

We can consider the partial order for `:search` to be, in some sense, inductively generated by several classes of "edits" which are described below.

Possible edits

Here is a simple approximate list of the edits that are possible in `:search`. They are not entirely formal, and do not necessarily reflect the `:search` command's actual behavior. For example, the *argument application* rule may be used directly on arguments that are bound after other arguments, without using several applications of the *argument transposition* rule.

- **Argument transposition**

  ```
     A : Type          B : Type           a : A, b : B  |-  M : Type
  ------------------------------------------------------------------------
  (x : A) -> (y : B) -> [x,y/a,b]M      ==    (y : B) -> (x : A) -> [x,y/a,b]M
  ```

  Score: 1 point

Example:

```
a -> Vect n a -> Vect (S n) a    ==    Vect n a -> a -> Vect (S n) a
```

Note that in order for it to make sense to change the order of arguments, neither of the arguments' types may depend on the value bound by the other argument!

- **Symmetry of equality**

```
A = B : Type      t : Type  |-  M : Type
---------------------------------------
 [A = B/t]M        ==       [B = A/t]M
```

  Score: 1 point

Example:

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

Note that this rule means that we can flip equalities anywhere they occur (i.e., not only in the return type).

- **Argument application**

```
  e : A  |-  M : Type        y1 : T1, ..., yn : Tn  |-  x : A
-------------------------------------------------------------------
(z : A) -> [z/e]M     >=   (y1 : T1) -> ... -> (yn :Tn) -> [x/e]M
```

  Score: <= : 3 points, >= : 9 points

Examples:

```
a -> a            >=    Nat -> Nat
a -> a            >=    List e -> List e
Vect k (Fin k)    >=    Vect 5 (Fin 5)
```

Note that the `n` shown in the scheme above may be 0; that is, there are no Pi terms to be added on the right side. For example, that's the case for the first example shown above. This is probably the most important, and most widely used, rule of all.

- **Type class application**

```
C : Type -> TypeClass
,   y1 : T1, ..., yn : Tn  |-  A : Type, instance : C A
,   t : Type  |- M : Type
-------------------------------------------------------------------
C a => [a/t]M     >=      (y1 : T1) -> ... -> (yn : Tn) -> [A/t]M
```

  Score: <= : 4 points, >= : 12 points

Examples

```
Ord a => a                          >=    Int
Show (List e) => List e -> String   >=    Show a => List a -> String
```

This rule is used by looking at the instances for a particular type class. While the scheme is shown only for single-parameter type classes, it naturally generalizes to multi-parameter type classes. This rule is particularly useful in conjunction with argument application. Again, note that the `n` in the scheme above may be 0.

- **Type class introduction**

```
t : Type |- M : Type      C : Type -> TypeClass
-----------------------------------------------
(t : Type) -> M      >=      C t => M
```

Score: <= : 2 points, >= : 6 points

Example:

```
a -> a -> Bool    >=    Eq a => a -> a -> Bool
```

Scoring and listing search results

When a type `S` is searched, the type is compared against the types of all of the terms which are currently in context. When :search compares two types `S` and `T`, it essentially tries to find a chain of inequalities

```
   R1    R2     Rn    Rn+1
S  <= A1 <= ... <= An <= T
```

using the edit rules listed above. It also tries to find chains going the other way (i.e., showing `S >= T`) as well. Each rule has an associated score which indicates how drastic of a change the rule corresponds to. These scores are listed above. Note that for the rules which are not symmetric, the score depends on the direction in which the rule is used. Finding types which are more general that the searched typed (`S <= T`) is preferred to finding types which are less general.

The score for the entire chain is, at minimum, the sum of the scores of the individual rules (some non-linear interactions may be added). The :search function tries to find the chain between `S` and `T` which results in the lowest score, and this is the score associated to the search result for `T`.

Search results are listed in order of ascending score. The symbol which is shown along with the search result reflects the type of the chain which resulted in the minimum score.

Implementation of :search

Practically, naive and undirected application of the rules enumerated above is not possible; not only is this obviously inefficient, but the two application rules (particularly *argument application*) are really impossible to use without context given by other types. Therefore, we use a heuristic algorithm that is meant to be practical, though it might not find ways to relate two types which may actually be related by the rules listed above.

Suppose we wish to match two types, `S` and `T`. We think of the problem as a non-deterministic state machine. There is a `State` datatype which keeps track of how well we've matched `S` and `T` so far. It contains:

- Names of argument variables (Pi-bound variables) in either type which have yet to be matched
- A directed acyclic graph (DAG) of arguments (Pi-bindings) for `S` and `T` which have yet to be matched
- A list of typeclass constraints for `S` and `T` which have yet to be matched
- A record of the rules which have been used so far to get to this point

A function `nextSteps : State -> [State]` finds the next states which may follow from a given state. Some states, where everything has been matched, are considered final. The algorithm can be roughly broken down into multiple stages; if we start from having two types, `S` and `T`, which we wish to match, they are as follows:

1. For each of `S` and `T`, split the types up into their return types and directed acyclic graphs of the arguments, where there is an edge from argument A to argument B if the term bound in A appears in the type of B. The topological sorts of the DAG represent all the possible ways in which the arguments may be permuted.

2. For type `T`, recursively find (saturated) uses of the `=` type constructor and produce a list of modified versions of `T` containing all possible flips of the `=` constructor (this corresponds to the *symmetry of equality rule*).

3. For each modified type for `T`, try to unify the return type of the modified `T` with `S`, considering arguments from both `S` and `T` to be holes, so that the unifier may match pieces of the two types. For each modified version of `T` where this succeeds, an initial `State` can be made. The arguments and typeclasses are updated accordingly with the results of unification. The remainder of the algorithm involves applying `nextSteps` to these states until either no states remain (corresponding to no path from `S` to `T`) or a final state is found. `nextSteps` also has several stages:

4. Try to unify arguments of `S` with arguments of `T`, much like is done with the return types. We work "backwards" through the arguments: we try matching all remaining arguments of `S` which lack outgoing edges in the DAG of remaining arguments (that is, the bound value doesn't appear in the type of any other remaining arguments) with the all of the corresponding remaining arguments of `T`. This is done recursively until no arguments remain for both `S` and `T`; otherwise, we give up at this point. This step corresponds to application of the *argument application rule*, as well as the *argument transposition* rule.

5. Now, we try to match the type classes. First, we take all possible subsets of type class constraints for `S` and `T`. So if `S` and `T` have a total of `n` type class constraints, this produces `2^n` states for every state, and this quickly becomes infeasible as `n` grows large. This is probably the biggest bottleneck of the algorithm at the moment. This step corresponds to applications of the *type class introduction* rule.

6. Try to match type class constraints for `S` with those for `T`. We attempt to unify each type class constraint for `S` with each constraint for `T`. This may result in applications of the *type class application* rule. Once we are unable to match any more type class constraints between `S` and `T`, we proceed to the final step.

7. Try instantiating type classes with their instances (in either `S` or `T`). This corresponds to applications of the *type class application* rule. After instantiating a type class, we hopefully open up more opportunities to match typeclass constraints of `S` with those of `T`, so we return to the previous step.

The code for :search is located in the Idris.TypeSearch module.

Aggregating results

The search for chains of rules/edits which relate two types can be viewed as a shortest path problem where nodes correspond to types and edges correspond to rules relating two types. The weights or distances on each edge correspond to the score of each rule. We then may imagine that we have a single start node, our search type `S`, and several final nodes: all of the types for terms which are currently in context. The problem, then, is to find the shortest paths (where they exist) to all of the final nodes. In particular, we wish to find the "closest" types (those with the minimum score) first, as we'd like to display them first.

This problem nicely maps to usage of Dijkstra's algorithm. We search for all types simultaneously so we can find the closest ones with the minimum amount of work. In practice, this results in using a priority queue of priority queues. We first ask "which goal type should we work on next?", and then ask "which state should we expand upon next?" By using this strategy, the best results can be shown quickly, even if it takes a bit of time to find worse results (or at least rule them out).

Miscellaneous Notes

Whether arguments are explicit or implicit does not affect search results.

Static Arguments and Partial Evaluation

As of version 0.9.15, Idris has support for *partial evaluation* of statically known arguments. This involves creating specialised versions of functions with arguments annotated as `[static]`.

(This is an implementation of the partial evaluator described in this ICFP 2010 paper. Please refer to this for more precise definitions of what follows.)

Partial evaluation is switched on by default. It can be disabled with the `--no-partial-eval` flag.

Introductory Example

Consider the power function over natural numbers, defined as follows (we'll call it `my_pow` since `pow` already exists in the Prelude):

```
my_pow : Nat -> Nat -> Nat
my_pow x Z = 1
my_pow x (S k) = mult x (my_pow x k)
```

This is implemented by recursion on the second argument, and we can evaluate the definition further if the second argument is known, even if the first isn't. For example, we can build a function at the REPL to cube a number as follows:

```
*pow> \x => my_pow x 3
\x => mult x (mult x (mult x 1)) : Nat -> Nat
*pow> it 3
27 : Nat
```

Note that in the resulting function the recursion has been eliminated, since `my_pow` is implemented by recursion on the known argument. We have no such luck if the first argument is known and the second isn't:

```
*pow> \x => my_pow 2 x
\x => my_pow 2 x : Nat -> Nat
```

Now, consider the following definition which calculates x^2 + 1:

```
powFn : Nat -> Nat
powFn x = plus (my_pow x (S (S Z))) (S Z)
```

Since the second argument to `my_pow` here is statically known, it seems a shame to have to make the recursive calls every time. However, Idris will not in general inline recursive definitions, in particular since they may diverge or duplicate work without some deeper analysis.

We can, however, give Idris some hints that here we really would like to create a specialised version of `my_pow`.

Automatic specialisation of `pow`

The trick is to mark the statically known arguments with the `[static]` flag:

```
my_pow : Nat -> [static] Nat -> Nat
my_pow k Z = 1
my_pow k (S j) = mult k (my_pow k j)
```

When an argument is annotated in this way, Idris will try to create a specialised version whenever it accounts a call with a concrete value (i.e. a constant, constructor form, or globally defined function) in a `[static]` position. If `my_pow` is defined this way, and `powFn` defined as above, we can see the effect by typing `:printdef powFn` at the REPL:

```
*pow> :printdef powFn
powFn : Nat -> Nat
powFn x = plus (PE_my_pow_3f3e5ad8 x) 1
```

What is this mysterious `PE_my_pow_3f3e5ad8`? It's a specialised power function where the statically known argument has been specialised away. The name is generated from a hash of the specialised arguments, and we can see its definition with `:printdef` too:

```
*petest> :printdef PE_my_pow_3f3e5ad8
PE_my_pow_3f3e5ad8 : Nat -> Nat
PE_my_pow_3f3e5ad8 (0arg) = mult (0arg) (mult (0arg) (PE_fromInteger_7ba9767f 1))
```

The `(0arg)` is an internal argument name (programmers can't give variable names beginning with a digit after all). Notice also that there is a specialised version of `fromInteger` for `Nat`s, since type class dictionaries are themselves a particularly common case of statically known arguments!

Specialising Type Classes

Type class dictionaries are very often statically known, so Idris automatically marks any type class constraint as `[static]` and builds specialised versions of top level functions where the class is instantiated. For example, given:

```
calc : Int -> Int
calc x = (x * x) + x
```

If we print this definition, we'll see a specialised version of `+` is used:

```
*petest> :printdef calc
calc : Int -> Int
calc x = PE_+_954510b4 (PE_*_954510b4 x x) x
```

More interestingly, consider `vadd` which adds corresponding elements in a vector of anything numeric:

```
vadd : Num a => Vect n a -> Vect n a -> Vect n a
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

If we use this on something concrete as follows...

```
test : List Int -> List Int
test xs = let xs' = fromList xs in
              toList $ vadd xs' xs'
```

...then in fact, we get a specialised version of `vadd` in the definition of `test`, and indeed the specialised version of `toList`:

```
test : List Int -> List Int
test xs = let xs' = fromList xs
          in PE_toList_888ae67 (PE_vadd_33f98d3d xs' xs')
```

Here's the specialised version of `vadd`:

```
PE_vadd_33f98d3d : Vect n Int -> Vect n Int -> Vect n Int
PE_vadd_33f98d3d [] [] = []
PE_vadd_33f98d3d (x :: xs) (y :: ys) = ((PE_+_954510b4 x y) ::
                                        (PE_vadd_33f98d3d xs ys))
```

Note that the recursive structure has been preserved, and the recursive call to `vadd` has been replaced with a recursive call to the specialised version. We've also got the same specialised version of `+` that we had above in `calc`.

Specialising Higher Order Functions

Another case where partial evaluation can be useful is in automatically making specialised versions of higher order functions. Unlike type class dictionaries, this is not done automatically, but we might consider writing `map` as follows:

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

Then using `my_map` will yield specialised versions, for example to double every value in a list of `Int`s we could write:

```
doubleAll : List Int -> List Int
doubleAll xs = my_map (*2) xs
```

This would yield a specialised version of `my_map`, used in `doubleAll` as follows:

```
doubleAll : List Int -> List Int
doubleAll xs = PE_my_map_1f8225c4 xs

PE_my_map_1f8225c4 : List Int -> List Int
PE_my_map_1f8225c4 [] = []
PE_my_map_1f8225c4 (x :: xs) = ((PE_*_954510b4 x 2) :: (PE_my_map_1f8225c4 xs))
```

Specialising Interpreters

A particularly useful situation where partial evaluation becomes effective is in defining an interpreter for a well-typed expression language, defined as follows (see the Idris tutorial, section 4 for more details on how this works):

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

dsl expr
    lambda = Lam
    variable = Var
    index_first = stop
    index_next = pop
```

We can write a couple of test functions in this language as follows, using the `dsl` notation to overload

lambdas; first a function which multiplies two inputs:

```
eMult : Expr gamma (TyFun TyInt (TyFun TyInt TyInt))
eMult = expr (\x, y => Op (*) x y)
```

Then, a function which calculates the factorial of its input:

```
eFac : Expr gamma (TyFun TyInt TyInt)
eFac = expr (\x => If (Op (==) x (Val 0))
            (Val 1)
            (App (App eMult (App eFac (Op (-) x (Val 1)))) x))
```

The interpreter's type is written as follows, marking the expression to be evaluated as `[static]`:

```
interp : (env : Env gamma) -> [static] (e : Expr gamma t) -> interpTy t
```

This means that if we write an Idris program to calculate a factorial by calling `interp` on `eFac`, the resulting definition will be specialised, partially evaluating away the interpreter:

```
runFac : Int -> Int
runFac x = interp [] eFac x
```

We can see that the call to `interp` has been partially evaluated away as follows:

```
*interp> :printdef runFac
runFac : Int -> Int
runFac x = PE_interp_ed1429e [] x
```

If we look at `PE_interp_ed1429e` we'll see that it follows exactly the structur of `eFac`, with the interpreter evaluated away:

```
*interp> :printdef PE_interp_ed1429e
PE_interp_ed1429e : Env gamma -> Int -> Int
PE_interp_ed1429e (3arg) = \x =>
                             boolElim (x == 0)
                                      (Delay 1)
                                      (Delay (PE_interp_b5c2d0ff (x :: (3arg))
                                                                 (PE_interp_ed1429e (x ::␣
↪(3arg)) (x - 1)) x))
```

For the sake of readability, I have simplified this slightly: what you will really see also includes specialised versions of `==`, `-` and `fromInteger`. Note that `PE_interp_ed1429e`, which represents `eFac` has become a recursive function following the structure of `eFac`. There is also a call to `PE_interp_b5c2d0ff` which is a specialised interpeter for `eMult`.

These definitions arise because the partial evaluator will only specialise a definition by a specific concrete argument once, then it is cached for future use. So any future applications of `interp` on `eFac` will also be translated to `PE_interp_ed1429e`.

The specialised version of `eMult`, without any simplification for readability, is:

```
PE_interp_b5c2d0ff : Env gamma -> Int -> Int -> Int
PE_interp_b5c2d0ff (3arg) = \x => \x1 => PE_*_954510b4 x x1
```

Miscellaneous

Things we have yet to classify, or are two small to justify their own page.

The Unifier Log

If you're having a hard time debugging why the unifier won't accept something (often while debugging the compiler itself), try applying the special operator `%unifyLog` to the expression in question. This will cause the type checker to spit out all sorts of informative messages.

Namespaces and type-directed disambiguation

Names can be defined in separate namespaces, and disambiguated by type. An expression `with NAME EXPR` will privilege the namespace `NAME` in the expression `EXPR`. For example:

```
Idris> with List [[1,2],[3,4],[5,6]]
[[1, 2], [3, 4], [5, 6]] : List (List Integer)

Idris> with Vect [[1,2],[3,4],[5,6]]
[[1, 2], [3, 4], [5, 6]] : Vect 3 (Vect 2 Integer)

Idris> [[1,2],[3,4],[5,6]]
Can't disambiguate name: Prelude.List.::, Prelude.Stream.::, Prelude.Vect.::
```

Alternatives

The syntax `(| option1, option2, option3, ... |)` type checks each of the options in turn until one of them works. This is used, for example, when translating integer literals.

```
Idris> the Nat (| "foo", Z, (-3) |)
0 : Nat
```

This can also be used to give simple automated proofs, for example: trying some constructors of proofs.

```
syntax Trivial = (| oh, refl |)
```

Totality checking assertions

All definitions are checked for *coverage* (i.e. all well-typed applications are handled) and either for *termination* (i.e. all well-typed applications will eventually produce an answer) or, if returning codata, for productivity (in practice, all recursive calls are constructor guarded).

Obviously, termination checking is undecidable. In practice, the termination checker looks for *size change* - every cycle of recursive calls must have a decreasing argument, such as a recursive argument of a strictly positive data type.

There are two built-in functions which can be used to give the totality checker a hint:

- `assert_total x` asserts that the expression `x` is terminating and covering, even if the totality checker cannot tell. This can be used for example if `x` uses a function which does not cover all inputs, but the caller knows that the specific input is covered.
- `assert_smaller p x` asserts that the expression `x` is structurally smaller than the pattern `p`.

For example, the following function is not checked as total:

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

This is because the checker cannot tell that `filter` will always produce a value smaller than the pattern `x :: xs` for the recursive call to `qsort`. We can assert that this will always be true as follows:

```
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))))
```

C heap

Idris has two heaps where objects can be allocated:

FP heap	C heap
Cheney-collected	Mark-and-sweep-collected
Garbage collections touches only live objects.	Garbage collection has to traverse all registered items.
Ideal for FP-style rapid allocation of lots of small short-lived pieces of memory, such as data constructors.	Ideal for C-style allocation of a few big buffers.
Finalizers are impossible to support reasonably.	Items have finalizers that are called on deallocation.
Data is copied all the time (when collecting garbage, modifying data, registering managed pointers, etc.)	Copying does not happen.
Contains objects of various types.	Contains C heap items: `(void *)` pointers with finalizers. A finalizer is a routine that deallocates the resources associated with the item.
Fixed set of object types.	The data pointer may point to anything, as long as the finalizer cleans up correctly.
Not suitable for C resources and arbitrary pointers.	Suitable for C resources and arbitrary pointers.
Values form a compact memory block.	Items are kept in a linked list.
Any Idris value, most notably `ManagedPtr`.	Items represented by the Idris type `CData`.
Data of `ManagedPtr` allocated in C, buffer then copied into the FP heap.	Data allocated in C, pointer copied into the C heap.
Allocation and reallocation not possible from C code (without having a reference to the VM). Everything is copied instead.	Allocated and reallocate freely in C, registering the allocated items in the FFI.

The FP heap is the primary heap. It may contain values of type `CData`, which are references to items in the C heap. A C heap item contains a `(void *)` pointer and the corresponding finalizer. Once a C heap item is no longer referenced from the FP heap, it is marked as unused and the next GC sweep will call its finalizer and deallocate it.

There is no Idris interface for `CData` other than its type and FFI.

Usage from C code

- Although not enforced in code, `CData` is meant to be opaque and non-RTS code (such as libraries or C bindings) should access only its `(void *)` field called `data`.
- Feel free to mutate both the pointer `data` (eg. after calling `realloc`) and the memory it points to. However, keep in mind that this must not break Idris's referential transparency.

- **WARNING!** If you call `cdata_allocate` or `cdata_manage`, the resulting `CData` object *must* be returned from your FFI function so that it is inserted in the C heap by the RTS. Otherwise the memory will be leaked.

```
some_allocating_fun : Int -> IO CData
some_allocating_fun i = foreign FFI_C "some_allocating_fun" (Int -> IO CData) i

other_fun : CData -> Int -> IO Int
other_fun cd i = foreign FFI_C "other_fun" (CData -> Int -> IO Int) cd i
```

```
#include "idris_rts.h"

static void finalizer(void * data)
{
    MyStruct * ptr = (MyStruct *) data;
    free_something(ptr->something);
    free(ptr);
}

CData some_allocating_fun(int arg)
{
    void * data = (void *) malloc(...);
    // ...
    return cdata_manage(data, finalizer);
}

int other_fun(CData cd, int arg)
{
    int result = foo(cd->data);
    return result;
}
```

Preorder reasoning

This syntax is defined in the module `Syntax.PreorderReasoning` in the `base` package. It provides a syntax for composing proofs of reflexive-transitive relations, using overloadable functions called `step` and `qed`. This module also defines `step` and `qed` functions allowing the syntax to be used for demonstrating equality. Here is an example:

```
import Syntax.PreorderReasoning
multThree : (a, b, c : Nat) -> a * b * c = c * a * b
multThree a b c =
  (a * b * c) ={ sym (multAssociative a b c) }=
  (a * (b * c)) ={ cong (multCommutative b c) }=
  (a * (c * b)) ={ multAssociative a c b }=
  (a * c * b) ={ cong {f = (* b)} (multCommutative a c) }=
  (c * a * b) QED
```

Note that the parentheses are required – only a simple expression can be on the left of `={ }=` or `QED`. Also, when using preorder reasoning syntax to prove things about equality, remember that you can only relate the entire expression, not subexpressions. This might occasionally require the use of `cong`.

Finally, although equality is the most obvious application of preorder reasoning, it can be used for any reflexive-transitive relation. Something like `step1 ={ just1 }= step2 ={ just2 }= end QED` is translated to `(step step1 just1 (step step2 just2 (qed end)))`, selecting the appropriate definitions of `step` and `qed` through the normal disambiguation process. The standard library, for example, also contains an implementation of preorder reasoning on isomorphisms.

Pattern matching on Implicit Arguments

Pattern matching is only allowed on implicit arguments when they are referred by name, e.g.

```
foo : {n : Nat} -> Nat
foo {n = Z} = Z
foo {n = S k} = k
```

or

```
foo : {n : Nat} -> Nat
foo {n = n} = n
```

The latter could be shortened to the following:

```
foo : {n : Nat} -> Nat
foo {n} = n
```

That is, `{x}` behaves like `{x=x}`.

Existence of an implementation

In order to show that an implementation of some interface is defined for some type, one could use the `%implementation` keyword:

```
foo : Num Nat
foo = %implementation
```

'match' application

`ty <== name` applies the function `name` in such a way that it has the type `ty`, by matching `ty` against the function's type. This can be used in proofs, for example:

```
plus_comm : (n : Nat) -> (m : Nat) -> (n + m = m + n)
-- Base case
(Z + m = m + Z) <== plus_comm =
    rewrite ((m + Z = m) <== plusZeroRightNeutral) ==>
            (Z + m = m) in refl

-- Step case
(S k + m = m + S k) <== plus_comm =
    rewrite ((k + m = m + k) <== plus_comm) in
    rewrite ((S (m + k) = m + S k) <== plusSuccRightSucc) in
        refl
```

Reflection

Including `%reflection` functions and `quoteGoal x by fn in t`, which applies `fn` to the expected type of the current expression, and puts the result in `x` which is in scope when elaborating `t`.

Bash Completion

Use of `optparse-applicative` allows Idris to support Bash completion. You can obtain the completion script for Idris using the following command:

```
idris --bash-completion-script `which idris`
```

To enable completion for the lifetime of your current session, run the following command:

```
source <(idris --bash-completion-script `which idris`)
```

To enable completion permenatly you must either:

- Modify your bash init script with the above command.
- Add the completion script to the appropriate `bash_completion.d/` folder on your machine.

CHAPTER 7

Tutorials on the Idris Language

Tutorials submitted by community members.

Type Providers in Idris

Type providers in Idris are simple enough, but there are a few caveats to using them that it would be worthwhile to go through the basic steps. We also go over foreign functions, because these will often be used with type providers.

The use case

First, let's talk about *why* we might want type providers. There are a number of reasons to use them and there are other examples available around the net, but in this tutorial we'll be using them to port C's `struct stat` to Idris.

Why do we need type providers? Well, Idris's FFI needs to know the types of the things it passes to and from C, but the fields of a `struct stat` are implementation-dependent types that cannot be relied upon. We don't just want to hard-code these types into our program... so we'll use a type provider to find them at compile time!

A simple example

First, let's go over a basic usage of type providers, because foreign functions can be confusing but it's important to remember that providers themselves are simple.

A type provider is simply an IO action that returns a value of this type:

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

Looks familiar? `Provider` is just `Either a String`, given a slightly more descriptive name.

Remember though, type providers we use in our program must be IO actions. Let's write a simple one now:

```
module Provider
-- Asks nicely for the user to supply the size of C's size_t type on this
-- machine
getSizeT : IO (Provider Int)
getSizeT = do
  putStrLn "I'm sorry, I don't know how big size_t is. Can you tell me, in bytes?"
  resp <- getLine
  case readInt resp of
     Just sizeTSize => pure (Provide sizeTSize)
     Nothing => pure (Error "I'm sorry, I don't understand.")
-- the readInt function is left as an exercise
```

We assume that whoever's compiling the library knows the size of `size_t`, so we'll just ask them! (Don't worry, we'll get it ourselves later.) Then, if their response can be converted to an integer, we present `Provide sizeTSize` as the result of our IO action; or if it can't, we signal a failure. (This will then become a compile-time error.)

Now we can use this IO action as a type provider:

```
module Main
-- to gain access to the IO action we're using as a provider
import Provider

-- TypeProviders is an extension, so we'll enable it
%language TypeProviders

-- And finally, use the provider!
-- Note that the parentheses are mandatory.
%provide (sizeTSize : Int) with getSizeT

-- From now on it's just a normal program where `sizeTSize` is available
-- as a top-level constant
main : IO ()
main = do
  putStr "Look! I figured out how big size_t is! It's "
  putStr (show sizeTSize)
  putStr " bytes!"
```

Yay! We... asked the user something at compile time? That's not very good, actually. Our library is going to be difficult to compile! This is hardly a step up from having them edit in the size of `size_t` themselves!

Don't worry, there's a better way.

Foreign Functions

It's actually pretty easy to write a C function that figures out the size of `size_t`:

```
int sizeof_size_t() { return sizeof(size_t); }
```

(Why an int and not a `size_t`? The FFI needs to know how to receive the return value of this function and translate it into an Idris value. If we knew how to do this for values of C type `size_t`, we wouldn't need to write this function at all! If we really wanted to be safe from overflow, we could use an array of multiple integers, but the SIZE of `size_t` is never going to be a 65535 byte integer.)

So now we can get the size of `size_t` as long as we're in C code. We'd like to be able to use this from Idris. Can we do this? It turns out we can.

`foreign`

With foreign, we can turn a C function into an IO action. It works like this:

```
getSizeT : IO Int
getSizeT = foreign FFI_C "sizeof_size_t" (IO Int)
```

Pretty simple. `foreign` takes a specification of what function it needs to call and that function's return type.

Running foreign functions

This is all well and good for writing code that will typecheck. To actually run the code, we'll need to do just a bit more work. Exactly what we need to do depends on whether we want to interpret or compile our code.

In the interpreter

If we want to call our foreign functions from interpreted code (such as the REPL or a type provider), we need to dynamically link a library containing the symbols we need. This is pretty easy to do with the `%dynamic` directive:

```
%dynamic "./filename.so"
```

Note that the leading "./" is important: currently, the string you provide is interpreted as by `dlopen()`, which on Unix does not search in the current directory by default. If you use the "./", the library will be searched for in the directory from which you run idris (*not* the location of the file you're running!). Of course, if you're using functions from an installed library rather than something you wrote yourself, the "./" is not necessary.

In an executable

If we want to run our code from an executable, we can statically link instead. We'll use the `%include` and `%link` directives:

```
%include C "filename.h"
%link C "filename.o"
```

Note the extra argument to the directive! We specify that we're linking a C header and library. Also, unlike `%dynamic`, these directives search in the current directory by default. (That is, the directory from which we run idris.)

Putting it all together

So, at the beginning of this article I said we'd use type providers to port `struct stat` to Idris. The relevant part is just translating all the mysterious typedef'd C types into Idris types, and that's what we'll do here.

First, let's write a C file containing functions that we'll bind to.

```
/* stattypes.c */
#include <sys/stat.h>

int sizeof_dev_t() { return sizeof(dev_t); }
int sizeof_ino_t() { return sizeof(ino_t); }
/* lots more functions like this */
```

Next, an Idris file to define our providers:

```
-- Providers.idr
module Providers

%dynamic "./stattypes.so"

sizeOfDevT : IO Int
sizeOfDevT = foreign FFI_C "sizeof_dev_t" (IO Int)
{- lots of similar functions -}

-- Indicates how many bits are used to represent various system
-- stat types.
data BitWidth = B8 | B16 | B32 | B64

implementation Show BitWidth where
  show B8 = "8 bits"
  show B16 = "16 bits"
  show B32 = "32 bits"
  show B64 = "64 bits"

-- Now we have an integer, but we want a Provider BitWidth.
-- Since our sizeOf* functions are ordinary IO actions, we
-- can just map over them.
bytesToType : Int -> Provider BitWidth
bytesToType 1 = Provide B8  -- "8 bit value"
bytesToType 2 = Provide B16
bytesToType 4 = Provide B32
bytesToType 8 = Provide B64
bytesToType _ = Error "Unrecognised integral type."

getDevT : IO (Provider BitWidth)
getDevT = map bytesToType sizeOfDevT
{- lots of similar functions -}
```

Finally, we'll write one more idris file where we use the type providers:

```
-- Main.idr
module Main
import Providers
%language TypeProviders
```

```
%provide (DevTBitWidth : BitWidth) with getDevT

-- We can now use DevTBitWidth in our program!
main : IO ()
main = putStrLn $ "size of dev_t: " ++ show DevTBitWidth
```

The Interactive Theorem Prover

This short guide contributed by a community member illustrates how to prove associativity of addition on `Nat` using the interactive theorem prover.

First we define a module `Foo.idr`

```
module Foo

plusAssoc : plus n (plus m o) = plus (plus n m) o
plusAssoc = ?rhs
```

We wish to perform induction on `n`. First we load the file into the Idris `REPL` as follows:

```
$ idris Foo.idr
```

We will be given the following prompt, in future releases the version string will differ:

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9.18.1
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./Foo.idr
Metavariables: Foo.rhs
*Foo>
```

Explore the Context

We start the interactive session by asking Idris to prove the hole `rhs` using the command `:p rhs`. Idris by default will show us the initial context. This looks as follows:

```
*Foo> :p rhs
----------                 Goal:                 ----------
{ hole 0 }:
 (n : Nat) ->
 (m : Nat) ->
 (o : Nat) ->
 plus n (plus m o) = plus (plus n m) o
```

Application of Intros

We first apply the `intros` tactic:

```
-Foo.rhs> intros
----------              Other goals:              ----------
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
{ hole 3 }:
 plus n (plus m o) = plus (plus n m) o
```

Induction on `n`

Then apply `induction` on to `n`:

```
-Foo.rhs> induction n
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus Z (plus m o) = plus (plus Z m) o
```

Compute

```
-Foo.rhs> compute
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus m o = plus m o
```

Trivial

```
-Foo.rhs> trivial
----------              Other goals:              ----------
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
```

```
 n : Nat
 m : Nat
 o : Nat
----------                  Goal:                  ----------
elim_S0:
 (n__0 : Nat) ->
 (plus n__0 (plus m o) = plus (plus n__0 m) o) ->
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

Intros

```
-Foo.rhs> intros
----------              Other goals:              ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                  Goal:                  ----------
{ hole 5 }:
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

Compute

```
-Foo.rhs> compute
----------              Other goals:              ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                  Goal:                  ----------
{ hole 5 }:
 S (plus n__0 (plus m o)) = S (plus (plus n__0 m) o)
```

Rewrite

```
-Foo.rhs> rewrite ihn__0
----------              Other goals:              ----------
{ hole 5 }
{ hole 4 }
elim_S0
{ hole 2 }
```

```
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                 Goal:                  ----------
{ hole 6 }:
 S (plus n__0 (plus m o)) = S (plus n__0 (plus m o))
```

Trivial

```
-Foo.rhs> trivial
rhs: No more goals.
-Foo.rhs> qed
Proof completed!
Foo.rhs = proof
  intros
  induction n
  compute
  trivial
  intros
  compute
  rewrite ihn__0
  trivial
```

Two goals were created: one for `Z` and one for `S`. Here we have proven associativity, and assembled a tactic based proof script. This proof script can be added to `Foo.idr`.

Bibliography

[BMM04] Edwin Brady, Conor McBride, James McKinna: Inductive families need not store their indices