# Programming in IDRIS: A Tutorial

The IDRIS Community

28th November 2013

## Contents

# 1 Introduction

In conventional programming languages, there is a clear distinction between *types* and *values*. For example, in Haskell [9], 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 (e.g. the length of a list) the type of a function can begin to describe its own properties. For example, concatenating two lists 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 *systems* programming. To this end, IDRIS is a compiled language which aims to generate efficient executable code. It also has a lightweight foreign function interface which allows easy interaction with external C libraries.

## 1.1 Intended Audience

This tutorial is intended as a brief introduction to the language, and is aimed at readers already familiar with a functional language such as Haskell[2] or OCaml[3]. 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.

## 1.2 Example Code

This tutorial includes some example code, which has been tested with IDRIS version 0.9.10. The files are available in the IDRIS distribution, and provided along side the tutorial source, so that you can try them out easily, under `tutorial/examples`. However, it is strongly recommended that you can type them in yourself, rather than simply loading and reading them.

# 2 Getting Started

## 2.1 Prerequisites

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

- A fairly recent Haskell platform. Version 2010.1.0.0.1 is currently sufficiently recent.
- The *GNU Multiple Precision Arithmetic Library* (GMP) is available from MacPorts and all major Linux distributions.

[1] Typically, and perhaps confusingly, referred to in the dependently typed programming literature as "vectors"
[2] `http://www.haskell.org`
[3] `http://ocaml.org`

## 2.2 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 on GitHub at: `https://github.com/edwinb/Idris-dev`.

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
```

Note that the `$` 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.
- `-help` display usage summary and command line options

## 2.3 The Interactive Environment

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

```
$ idris
      _____     __      _
     /  _/___/ /_____(_)_____
     / // __  / ___/ / ___/      Version  0.9.10
   _/ // /_/ / /  / (__  )       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, as shown in Listing 1. Listing 2 shows 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`.

Listing 1: Interactive environment commands

```
Idris version  0.9.10
--------------------
```

```
Command         Arguments   Purpose

<expr>                      Evaluate an expression
:t              <expr>      Check the type of an expression
:miss :missing  <name>      Show missing clauses
:doc            <name>      Show internal documentation
:i :info        <name>      Display information about a type class
: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
:m :module      <module>    Import an extra module
:e :edit                    Edit current file using $EDITOR or $VISUAL
:m :metavars    <metavar>   Show remaining proof obligations (metavariables)
:p :prove       <metavar>   Prove a metavariable
: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 interpreter
:c :compile     <filename>  Compile to an executable <filename>
:js :javascript <filename>  Compile to JavaScript <filename>
:exec :execute              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)
:unset          <option>    Unset an option
:colour :color  <option>    Turn REPL colours on or off; set a specific colour
:q :quit                    Exit the Idris system
```

Listing 2: Sample Interactive Run

```
$ idris hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version  0.9.10
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

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

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.

# 3 Types and Functions

## 3.1 Primitive Types

IDRIS defines several primitive types: `Int`, `Integer` and `Float` 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 (none here, however IDRIS programs can consist of several modules, and the definitions in each module each have their own namespace, as we will discuss in Section 5) and a collection of declarations and definitions. The order of definitions is significant — functions and data types must be defined before use. Each definition must have a type declaration, for example see `x : Int`, `foo : String`, from the above listing. Indentation is significant — a new declaration begins at the same level of indentation as the preceding declaration. Alternatively, declarations may be terminated with a semicolon.

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 : Int
*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 type classes, as we will discuss in Section 4 and can be extended to work on user defined types. Boolean expressions can be tested with the `if...then...else` construct:

```
*prims> if x == 6 * 6 + 6 then "The␣answer!" else "Not␣the␣answer"
"The␣answer!" : String
```

## 3.2 Data Types

Data types are declared in a similar way to Haskell data types, with a similar syntax. 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:

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

## 3.3 Functions

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

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

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

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

We can test these functions at the IDRIS prompt:

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

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

```
Idris> plus 2 2
S (S (S (S Z))) : Nat
Idris> mult 3 (plus 2 2)
S (S (S (S (S (S (S (S (S (S (S (S Z)))))))))))) : 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, so optimises 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.

**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
```

## 3.4 Dependent Types

### 3.4.1 Vectors

A standard example of a dependent type is the type of "lists with length", conventionally called vectors in the dependent type literature. In IDRIS, we declare vectors as follows[4]:

[4]Note that prior to IDRIS 0.9.9, the order of the arguments to `Vect` was reversed.

```
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 in the library, appends two `Vect`s:

```
(++) : 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

$ idris vbroken.idr --check
vbroken.idr:3:Can't unify Vect n a with Vect m a

Specifically:
  Can't unify n with m
```

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

### 3.4.2 The Finite Sets

Finite sets, as the name suggests, are sets with a finite number of elements. They are declared as follows (again, in the prelude):

```
data Fin : Nat -> Type where
   fZ : Fin (S k)
   fS : Fin k -> Fin (S k)
```

`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. Obviously we can't construct an element of an empty set, so neither constructor targets `Fin Z`.

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 natural numbers bounded by `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.

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`.

### 3.4.3 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 with a *lower case initial letter* which appears as a parameter or index in a type declaration, but which is otherwise free, will 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.

### 3.4.4 "`using`" notation

Sometimes it is necessary to provide types of implicit arguments where the type checker can not work them out itself. This can happen if there is a dependency ordering — obviously, `a` and `n` must be given as arguments above before being used — or if an implicit argument has a complex type. For example, we will need to state the types of the implicit arguments in the following definition, which defines a predicate on vectors:

```
data Elem : a -> Vect n a -> Type where
   here :  {x:a} ->   {xs:Vect n a} -> Elem x (x :: xs)
   there : {x,y:a} -> {xs:Vect n a} -> Elem x xs -> Elem x (y :: xs)
```

An instance of `Elem 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 : Elem 5 testVec
inVect = there (there here)
```

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 Elem : a -> Vect n a -> Type where
     here  : Elem x (x :: xs)
     there : Elem x xs -> Elem 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 their reduction behaviour to affect type checking. However, this restriction can be relaxed by using a `mutual` block, which allows data types and functions to be defined simultaneously:

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

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

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

## 3.5 I/O

Computer programs are of little use if they do not interact with the user or the system in some way. The difficulty in a pure language such as IDRIS — that is, a language where expressions do not have side-effects — is that I/O is inherently side-effecting. Therefore in IDRIS, such interactions are encapsulated in the type `IO`:

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

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

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

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

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

We can also read strings from user input:

```
getLine : IO String
```

A number of other I/O operations are defined in the prelude, for example for reading and writing files, including:

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

```
openFile  : String -> Mode -> IO File
closeFile : File -> IO ()

fread  : File -> IO String
fwrite : File -> String -> IO ()
feof   : File -> IO Bool

readFile : String -> IO String
```

## 3.6 "do" notation

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

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

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

```
return : a -> IO a
```

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

## 3.7 Useful Data Types

IDRIS includes a number of useful data types and library functions (see the `libs/` directory in the distribution). This chapter describes a few of these. The functions described here are imported automatically by every IDRIS program, as part of `Prelude.idr`.

### 3.7.1 `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 5), 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, to double every element in a vector of integers:

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

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

You'll find these examples in `usefultypes.idr` in the `examples/` directory:

```
*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, in `libs/prelude/list.idr` and `libs/prelude/vect.idr` respectively. Functions include filtering, appending, reversing, and so on. Also remember that IDRIS is still in development, so if you don't see the function you need, please feel free to add it and submit a patch!

**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`.

### 3.7.2 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 : Maybe a -> |(default:b) -> (a -> b) -> b
```

The vertical bar | before the default value is a laziness annotation. Normally expressions are evaluated before being passed to a function. This is typically the most efficient behaviour. However, in this case, the default value might not be used and if it is a large expression, evaluating it will be wasteful. The | annotation tells the compiler not to evaluate the argument until it is needed.

### 3.7.3 Tuples and Dependent Pairs

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")
```

#### Dependent Pairs

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

```
data Exists : (A : Type) -> (P : A -> Type) -> Type where
   Ex_intro : {P : A -> Type} -> (a : A) -> P a -> Exists 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])
```

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.

## 3.8 so

The `so` data type is a predicate on `Bool` which guarantees that the value is true:

```
data so : Bool -> Type where
    oh : so True
```

This is most useful for providing a static guarantee that a dynamic check has been made. For example, we might provide a safe interface to a function which draws a pixel on a graphical display as follows, where `so (inBounds x y)` guarantees that the point `(x,y)` is within the bounds of a $640 \times 480$ window:

```
inBounds : Int -> Int -> Bool
inBounds x y = x >= 0 && x < 640 && y >= 0 && y < 480

drawPoint : (x : Int) -> (y : Int) -> so (inBounds x y) -> IO ()
drawPoint x y p = unsafeDrawPoint x y
```

## 3.9 More Expressions

### `let` bindings

Intermediate values can be calculated using `let` bindings:

```
mirror : List a -> List a
mirror xs = let xs' = rev 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 any numeric type, using the `count` 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 : Int
*usefultypes> lookup_default 4 [3,4,5,6] (-1)
-1 : Int
```

**Restrictions:** The `case` construct is intended for simple analysis of intermediate expressions to avoid the need to write auxiliary functions, and is also used internally to implement pattern matching `let` and lambda bindings. It will *only* work if:

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

### 3.10 Dependent 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. For example, we can represent a person's name and age in a record:

```
record Person : Type where
    MkPerson : (name : String) ->
               (age : Int) -> Person

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

Record declarations are like `data` declarations, except that they are introduced by the `record` keyword, and can only have one constructor. The names of the binders in the constructor type (`name` and `age`) here are the field names, which we can use to access the field values:

```
*record> name fred
"Fred" : String
*record> age fred
30 : Int
*record> :t name
name : Person -> String
```

We can also use the field names to update a record (or, more precisely, produce a new record with the given fields updates).

```
*record> record { name = "Jim" } fred
MkPerson "Jim" 30 : Person
*record> record { name = "Jim", age = 20 } fred
MkPerson "Jim" 20 : Person
```

The syntax `record { field = val, ... }` generates a function which updates the given fields in a record.

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, and the result does not affect the type of the record as a whole. For example:

```
record Class : Type where
    ClassInfo : (students : Vect n Person) ->
                (className : String) ->
                Class
```

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 (prelude.vect.:: (MkPerson "Fred" 30) (prelude.vect.Nil)) "CS"
  : Class
```

# 4 Type Classes

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 `Float` 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 a feature which has proved to be effective in Haskell, namely *type classes*. To define a type class, we provide a collection of overloaded operations which describe the interface for *instances* of that class. A simple example is the `Show` type class, which is defined in the prelude and provides an interface for converting values to `String`:

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

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

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

We can read this as: "under the constraint that `a` is an instance of `Show`, take an input `a` and return a `String`." An instance of a class is defined with an `instance` declaration, which provides implementations of the function for a specific type. For example, the `Show` instance for `Nat` could be defined as:

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

Idris> show (S (S (S Z)))
"sssO" : String
```

Only one instance of a class can be given for a type — instances may not overlap. Instance declarations can themselves have constraints. For example, to define a `Show` instance for vectors, we need to know that there is a `Show` instance for the element type, because we are going to use it to convert each element to a `String`:

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

## 4.1 Default Definitions

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

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

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

```
instance 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 class declaration, in terms of the other method:

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

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

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

## 4.2 Extending Classes

Classes 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` class which inherits methods from `Eq` as well as defining some of its own:

```
data Ordering = LT | EQ | GT

class 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` class 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` class. 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, classes and instances can have multiple constraints. Multiple constaints are written in brackets in a comma separated list, for example:

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

## 4.3 Monads and do-notation

So far, we have seen single parameter type classes, where the parameter is of type `Type`. In general, there can be any number (greater than 0) of parameters, 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:

```
class Monad (m : Type -> Type) where
    return : a -> m a
    (>>=)  : m a -> (a -> m b) -> m b
```

The `Monad` class allows us to encapsulate binding and computation, and is the basis of `do`-notation introduced in Section 3.6. 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` is an instance of `Monad`, defined using primitive functions. We can also define an instance for `Maybe`, as follows:

```
instance Monad Maybe where
    return = Just

    Nothing  >>= k = Nothing
    (Just x) >>= k = k x
```

Using this we can, for example, define a function which adds two `Maybe Int`s, 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
               return (x' + y') -- Add them
```

This function will extract the values from `x` and `y`, if they are available, or return `Nothing` if they are not. Managing the `Nothing` cases is achieved by the »= operator, hidden by the `do` notation.

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

#### Monad comprehensions

The list comprehension notation we saw in Section 3.9 is more general, and applies to anything which is an instance of `MonadPlus`:

```
class Monad m => MonadPlus (m : Type -> Type) where
    mplus : m a -> m a -> m a
    mzero : m 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 : MonadPlus m => Bool -> m ()
```

Then the comprehension is converted to `do` notation:

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

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

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

## 4.4 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" [8].

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

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

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

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

Rather than having to insert `m_app` everywhere there is an application, we can use *idiom brackets* to do the job for us. To do this, we use the `Applicative` class, which captures the notion of application for a data type:

```
infixl 2 <$>

class Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<$>) : f (a -> b) -> f a -> f b
```

`Maybe` is made an instance of `Applicative` as follows, where `<$>` is defined in the same way as `m_app` above:

```
instance Applicative Maybe where
    pure = Just

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

Using *idiom brackets* we can use this instance 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 |]
```

#### 4.4.1 An error-handling interpreter

Idiom notation is commonly useful when defining evaluators. McBride and Paterson describe such an evaluator [8], 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 integer 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 make it an instance of a type class 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 make `Eval` an instance of `Applicative`. Before `Eval` can be an instance of `Applicative` it is necessary to make `Eval` an instance of `Functor`:

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

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

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

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

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

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

## 4.5 Named Instances

It can be desirable to have multiple instances of a type class, for example to provide alternative methods for sorting or printing values. To achieve this, instances can be *named* as follows:

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

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

...we can sort it using the default `Ord` instance, then the named instance `myord` as follows, at the IDRIS prompt:

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

# 5 Modules and Namespaces

An IDRIS program consists of a collection of modules. Each module includes an optional `module` declaration giving the name of the module, a list of `import` statements giving the other modules which are to be imported, and a collection of declarations and definitions of types, classes and functions. For example, Listing 3 gives a module which defines a binary tree type `BTree` (in a file `btree.idr`) and Listing 4 gives a main program (in a file `bmain.idr` which uses the `bst` module to sort a list.

Listing 3: Binary Tree module

```
module btree

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) = toList l ++ (v :: toList r)

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

Listing 4: Binary Tree main program

```
module Main

import btree

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

The same names can be defined in multiple modules. This is possible because in practice names are *qualified* with the name of the module. The names defined in the `btree` module are, in full:

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

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

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

## 5.1 Export Modifiers

By default, all names defined in a module are exported for use by other modules. However, it is good practice only to export a minimal interface and keep internal details abstract. IDRIS allows functions, types and classes to be marked as: `public`, `abstract` or `private`:

- `public` means that both the name and definition are exported. For functions, this means that the implementation is exported (which means, for example, it can be used in a dependent type). For data types, this means that the type name and the constructors are exported. For classes, this means that the class name and method names are exported.
- `abstract` means that only the name is exported. For functions, this means that the implementation is not exported. For data types, this means that the type name is exported but not the constructors. For classes, this means that the class name is exported but not the method names.
- `private` means that neither the name nor the definition is exported.

If any definition is given an export modifier, then all names with no modifier are assumed to be `private`. For our `btree` module, it makes sense for the tree data type and the functions to be exported as `abstract`, as we see in Listing 5.

Listing 5: Binary Tree module, with export modifiers

```
module btree

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

abstract
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))

abstract
toList : BTree a -> List a
toList Leaf = []
toList (Node l v r) = app (toList l) (v :: toList r)

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

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

```
%access abstract
```

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

## 5.2 Explicit Namespaces

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

```
module foo

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

namespace y
  test : String -> String
  test x = x ++ x
```

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

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

### 5.3 Parameterised blocks

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

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

The effect of a `parameters` block is to add the declared parameters to every function, type and data constructor within the block. Outside the block, the parameters must be given explicitly:

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

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

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

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

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

## 6 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. 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       = Int
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), which we'll be using regularly as an implicit argument, so we define everything in a `using` block:

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

The full representation of expressions is given in Listing 6. They are indexed by the types of the local variables, and the type of the expression itself:

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

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.

Listing 6: Expression representation

```
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 : Int) -> 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 -> Expr G a -> Expr G a -> Expr G a
```

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 : Int) -> 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:

```
If  : Expr G TyBool -> Expr G a -> Expr G a -> Expr G a
```

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
```

Listing 7: Intepreter definition

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

Given this, an interpreter (Listing 7) 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
```

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 interpreters 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
```

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, we can write a factorial function. First, we write a *lazy* version of the `App` constructor, so that the recursive branch will only be evaluated if necessary:

```
app : |(f : Expr G (TyFun a t)) -> Expr G a -> Expr G t
app = \f, a => App f a
```

The vertical bar annotation before the `f` argument indicates that this argument should be evaluated lazily when `app` is applied — that is, the argument will be evaluated only when needed. Then `fact` (i.e. `\x. if (x == 0) then 1 else (fact (x-1) * x)`) is written as follows:

```
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)))
```

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
          print (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 shown in Listing 8.

Listing 8: Running the well-typed interpreter

```
$ idris interp.idr
      _____     __      _
     /  _/___/ /_____(_)_____
     / // __  / ___/ / ___/     Version  0.9.10
   _/ // /_/ / /  / (__  )      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 a type class `Cast` which allows conversion between types:

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

It is a *multi-parameter* type class, 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.

### 6.1 Unit testing

Recall the `so` data type:

```
data so : Bool -> Type where
  oh : so True
```

This requires its parameter to be a `True` value. One simple application of this is to perform unit testing of functions at compile time. For example, we know that `4! = 24` so we can set up a test case as follows:

```
unitTestFac : so (interp [] fact 4 == 24)
unitTestFac = oh
```

For the program to compile, the test case must pass. If we change `24` to `25`, say, we will get an error message like the following:

```
Type checking ./interp.idr
interp.idr:64:Can't unify so True with so False

Specifically:
   Can't unify True with False
```

## 7 Views and the "`with`" rule

### 7.1 Dependent pattern matching

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

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

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

### 7.2 The `with` rule — matching intermediate values

Very often, we need to match on the result of an intermediate computation. IDRIS provides a construct for this, the `with` rule, inspired by views in EPIGRAM [7], 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`s. Otherwise, it is the sum of two equal `Nat`s 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 Section 9 to complete the definition of `parity`.

# 8 Theorem Proving

## 8.1 Equality

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

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

Equalities can be proposed between any values of any types, but the only way to construct a proof of equality is if values actually are equal. For example:

```
fiveIsFive : 5 = 5
fiveIsFive = refl

twoPlusTwo : 2 + 2 = 4
twoPlusTwo = refl
```

## 8.2 The Empty Type

There is an empty type, $\bot$, which has no constructors. It is therefore impossible to construct an element of the empty type, at least without using a partially defined or general recursive function (see Section 8.5 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 -> _|_
disjoint n p = replace {P = disjointTy} p ()
  where
    disjointTy : Nat -> Type
    disjointTy Z = ()
    disjointTy (S k) = _|_
```

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. `FalseElim` is defined in the library, to assist with proofs by contradiction.

```
FalseElim : _|_ -> a
```

## 8.3 Simple Theorems

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

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

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

We won't go into details here, but the Curry-Howard correspondence [6] 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 trival 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 therefore provides an interactive proof mode.

## 8.4 Interactive theorem proving

Instead of writing the proof in one go, we can use IDRIS's interactive proof mode. To do this, we write the general *structure* of the proof, and use the interactive mode to complete the details. 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 *metavariables* 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 IDRIS, two global names are created, `plusredZ_O` and `plusredZ_S`, with no definition. We can use the `:m` command at the prompt to find out which metavariables 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 metavariables:
        [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 metavariable from the unsolved metavariables list. This also outputs a trace of the proof:

```
-plusredZ_Z> qed
plusredZ_Z = proof {
    compute;
    refine refl;
}
```

```
*theorems> :m
Global metavariables:
        [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 premisses using the `intro` tactic twice (or `intros`, which introduces all arguments as premisses). This gives:

```
  k : Nat
  ih : k = plus k Z
---------------------------------- (plusredZ_S) --------
{hole2} : S k = plus (S k) Z
```

Since plus is defined 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.

## 8.5 Totality Checking

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

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

-- not terminating
empty2 : _|_
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 : _|_
empty2 = empty2

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

Reassuringly, our proof in Section 8.2 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

### 8.5.1 Directives and Compiler Flags for Totality

By default, IDRIS allows all 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 IDRIS to print a warning for any undeclared partial function.

# 9 Provisional Definitions

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

```
data Parity : Nat -> Type where
   even : Parity (n + n)
   odd  : Parity (S (n + n))

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

We'd like to implement this as follows:

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

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

```
views.idr:12:Can't unify Parity (plus (S j) (S j)) with
Parity (S (S (plus j j)))
```

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

## 9.1 Provisional definitions

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

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

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

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

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

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

The first of these has the following type:

```
*views> :p views.parity_lemma_1

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

-views.parity_lemma_1>
```

The two arguments are `j`, the variable in scope from the pattern match, and `value`, which is the value we gave in the right hand side of the provisional definition. Our goal is to rewrite the type so that we can use this value. We can achieve this using the following theorem from the prelude:

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

After applying `intro` twice, we have:

```
-views.parity_lemma_1> intro

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

We need to use `compute` again to unfold the definition of `(+)`.

```
-views.parity_lemma_1> intro

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

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

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

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

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

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

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

We can now test the `natToBin` function from Section 7.2 at the prompt. The number 42 is 101010 in binary. The binary digits are reversed:

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

## 9.2 Suspension of Disbelief

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

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

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

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

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

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

## 9.3 Example: Binary numbers

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

```
data Binary : Nat -> Type where
   bEnd : Binary Z
   bO : Binary n -> Binary (n + n)
   bI : Binary n -> Binary (S (n + n))
```

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

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

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

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

```
natToBin : (n:Nat) -> Binary n
natToBin Z = bEnd
natToBin (S k) with (parity k)
   natToBin (S (j + j))     | even  = bI (natToBin j)
   natToBin (S (S (j + j))) | odd  ?= bO (natToBin (S j))
```

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

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

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

```
main : IO ()
main = do putStr "Enter a number: "
          x <- getLine
          print (natToBin (fromInteger (cast x)))
```

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

```
instance Show (Binary n) where
    show (bO x) = show x ++ "0"
    show (bI x) = show x ++ "1"
    show bEnd = ""
```

# 10 Syntax Extensions

IDRIS supports the implementation of *Embedded Domain Specific Languages* (EDSLs) in several ways [4]. 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.

## 10.1 `syntax` rules

We have seen `if...then...else` expressions, but these are not built in. Instead, we can define a function in the prelude as follows:

```
boolElim : (x:Bool) -> |(t : a) -> |(f : a) -> a;
boolElim True  t e = t;
boolElim False t e = e;
```

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

```
syntax if [test] then [t] else [e] = boolElim 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 : Float) -> (upper : Float) ->
                so (lower < upper) -> Interval
```

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

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

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

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

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

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

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

## 10.2 `dsl` notation

The well-typed interpreter in Section 6 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 [1] or running concurrent processes safely [2].

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 [4] to make it easier to program in such object languages:

```
dsl expr
    lambda      = Lam
    variable    = Var
    index_first = stop
    index_next  = pop
```

A `dsl` block describes how each syntactic construct is represented in an object language. Here, in the `expr` language, any IDRIS lambda is translated to a `Lam` constructor; 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). It is also possible to overload `let` 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:

```
(<$>) : |(f : Expr G (TyFun a t)) -> Expr G a -> Expr G t
(<$>) = \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 instance 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 type class instances, 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)
```

# 11 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.

## 11.1 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 using the `trivial` tactic, which searches through the context for a proof, and tries to solve with `refl` if a proof is not found. Now when `head` is applied, the proof can be omitted. In the case that a proof is not found, it can be provided explicitly as normal:

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

More generally, we can fill in implicit arguments with a default value by annotating them with `default`. The definition above is equivalent to:

```
head : (xs : List a) ->
       {default proof { trivial; } p : isCons xs = True} -> a
head (x :: xs) = x
```

## 11.2 Implicit conversions

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

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

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

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

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

## 11.3 Literate programming

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

```
> module literate

This is a comment. The main program is below

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

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

## 11.4 Foreign function calls

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

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

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

```
interpFTy : FTy -> Type
interpFTy FInt    = Int
interpFTy FFloat  = Float
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 function `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
```

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 : Float
```

## 11.5 Type Providers

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

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

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

```
%language TypeProviders
```

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

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

fromFile : String -> IO (Provider Type)
fromFile fname = do str <- readFile fname
                    return (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[5].

## 11.6 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.

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

### 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.

### 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.

### 11.7 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 [5], 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.

## 12 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://idris-lang.org/`) and by asking questions on the mailing list.
- The IRC channel `#idris`, on `chat.freenode.net`.
- 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, 3, 4]). Although these mostly describe older versions of IDRIS.

# References


[1] E. Brady. Idris — systems programming meets full dependent types. In *Programming Languages meets Program Verification (PLPV 2011)*, pages 43–54, 2011.

[2] E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010.

[3] E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ICFP '10: Proceedings of the 15th acm SIGPLAN International Conference on Functional Programming*, pages 297–308, New York, NY, USA, 2010. acm.

[4] E. Brady and K. Hammond. Resource-safe systems programming with embedded domain specific languages. In *Practical Applications of Declarative Languages 2012*, LNCS. Springer, 2012. To appear.

[5] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[6] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B.Curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[7] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[8] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18:1–13, January 2008.

[9] S. Peyton Jones et al. Haskell 98 language and libraries — the revised report. Available from `http://www.haskell.org/`, December 2002.