# Contents

1 Introduction 3
   1.1 Intended Audience ................................................. 3
   1.2 Example Code ..................................................... 3

2 Getting Started 3
   2.1 Prerequisites ..................................................... 3
   2.2 Downloading and Installing ....................................... 4
   2.3 The Interactive Environment ..................................... 4

3 Types and Functions 6
   3.1 Primitive Types .................................................. 6
   3.2 Data Types ........................................................ 7
   3.3 Functions .......................................................... 7
   3.4 Dependent Types .................................................. 9
      3.4.1 Vectors ......................................................... 9
      3.4.2 The Finite Sets .............................................. 10
      3.4.3 Implicit Arguments ......................................... 10
      3.4.4 "using" notation ............................................ 11
   3.5 I/O ................................................................. 12
   3.6 "do" notation ...................................................... 12
   3.7 Laziness ............................................................ 13
   3.8 Useful Data Types ................................................ 13
      3.8.1 List and Vect ............................................... 13
      3.8.2 Maybe ......................................................... 15
      3.8.3 Tuples and Dependent Pairs ................................ 15
   3.9 So ................................................................. 16
   3.10 More Expressions ................................................ 16
   3.11 Dependent Records ............................................... 18

4 Type Classes 19
   4.1 Default Definitions .............................................. 20
   4.2 Extending Classes ................................................ 20
   4.3 Monads and do-notation ......................................... 21
   4.4 Idiom brackets .................................................... 23
      4.4.1 An error-handling interpreter ................................. 23
   4.5 Named Instances .................................................. 24

5 Modules and Namespaces 25
   5.1 Export Modifiers .................................................. 26
   5.2 Explicit Namespaces ............................................... 27
   5.3 Parameterised blocks .............................................. 27
6 Example: The Well-Typed Interpreter 28

7 Views and the “with” rule 31
7.1 Dependent pattern matching 31
7.2 The with rule — matching intermediate values 32

8 Theorem Proving 32
8.1 Equality 32
8.2 The Empty Type 33
8.3 Simple Theorems 33
8.4 Interactive theorem proving 34
8.5 Totality Checking 36
8.5.1 Directives and Compiler Flags for Totality 37
8.5.2 Totality checking issues 37
8.5.3 Hints for totality 37

9 Provisional Definitions 38
9.1 Provisional definitions 38
9.2 Suspension of Disbelief 40
9.3 Example: Binary numbers 40

10 Interactive Editing 41
10.1 Editing at the REPL 41
10.2 Editing Commands 42
10.2.1 :addclause 42
10.2.2 :casesplit 42
10.2.3 :addmissing 43
10.2.4 :proofsearch 43
10.2.5 :makewith 43
10.3 Interactive Editing in Vim 44

11 Syntax Extensions 44
11.1 syntax rules 45
11.2 dsl notation 46

12 Miscellany 47
12.1 Auto implicit arguments 47
12.2 Implicit conversions 48
12.3 Literate programming 48
12.4 Foreign function calls 48
12.5 Type Providers 50
12.6 C Target 51
12.7 JavaScript Target 51
12.8 Cumulativity 53

13 Further Reading 53
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, \( \text{Vect} \ n \ a \), 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:

\[
\text{app} : \text{Vect} \ n \ a \rightarrow \text{Vect} \ m \ a \rightarrow \text{Vect} \ (n + m) \ a
\]

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 0.9.15. 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 2010.1.0.1 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, as well as build instructions, 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:

```idris
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 `$` indicates the shell prompt! Should the Idris executable not be found please ensure that you have added `~/.cabal/bin` to your `$PATH` environment variable. Mac OS X users may find they need to use `~/Library/Haskell/bin` instead. 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 0.9.15
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`. 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.
### Listing 1: Interactive environment commands

**Idris version 0.9.15**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr&gt;</td>
<td></td>
<td>Evaluate an expression</td>
</tr>
<tr>
<td>:t</td>
<td>&lt;expr&gt;</td>
<td>Check the type of an expression</td>
</tr>
<tr>
<td>:miss :missing &lt;name&gt;</td>
<td></td>
<td>Show missing clauses</td>
</tr>
<tr>
<td>:doc :name</td>
<td></td>
<td>Show internal documentation</td>
</tr>
<tr>
<td>:i :info :name</td>
<td></td>
<td>Display information about a type class</td>
</tr>
<tr>
<td>:total :name</td>
<td></td>
<td>Check the totality of a name</td>
</tr>
<tr>
<td>:r :reload</td>
<td></td>
<td>Reload current file</td>
</tr>
<tr>
<td>:l :load &lt;filename&gt;</td>
<td></td>
<td>Load a new file</td>
</tr>
<tr>
<td>:cd</td>
<td>&lt;filename&gt;</td>
<td>Change working directory</td>
</tr>
<tr>
<td>:m :module &lt;module&gt;</td>
<td></td>
<td>Import an extra module</td>
</tr>
<tr>
<td>:e :edit</td>
<td></td>
<td>Edit current file using $EDITOR or $VISUAL</td>
</tr>
<tr>
<td>:m :metavars &lt;metavar&gt;</td>
<td></td>
<td>Show remaining proof obligations (metavariables)</td>
</tr>
<tr>
<td>:p :prove &lt;metavar&gt;</td>
<td></td>
<td>Prove a metavariable</td>
</tr>
<tr>
<td>:a :addproof &lt;name&gt;</td>
<td></td>
<td>Add proof to source file</td>
</tr>
<tr>
<td>:rmproof</td>
<td>&lt;name&gt;</td>
<td>Remove proof from proof stack</td>
</tr>
<tr>
<td>:showproof</td>
<td>&lt;name&gt;</td>
<td>Show proof</td>
</tr>
<tr>
<td>:proofs</td>
<td></td>
<td>Show available proofs</td>
</tr>
<tr>
<td>:x</td>
<td>&lt;expr&gt;</td>
<td>Execute IO actions resulting from an expression using the interpreter</td>
</tr>
<tr>
<td>:c :compile &lt;filename&gt;</td>
<td></td>
<td>Compile to an executable &lt;filename&gt;</td>
</tr>
<tr>
<td>:js :javascript &lt;filename&gt;</td>
<td></td>
<td>Compile to JavaScript &lt;filename&gt;</td>
</tr>
<tr>
<td>:exec :execute</td>
<td></td>
<td>Compile to an executable and run</td>
</tr>
<tr>
<td>:dynamic :filename</td>
<td></td>
<td>Dynamically load a C library (similar to % dynamic)</td>
</tr>
<tr>
<td>:dynamic</td>
<td></td>
<td>List dynamically loaded C libraries</td>
</tr>
<tr>
<td>:? :h :help</td>
<td></td>
<td>Display this help text</td>
</tr>
<tr>
<td>:set :option</td>
<td></td>
<td>Set an option (errorcontext, showimplicits)</td>
</tr>
<tr>
<td>:unset :option</td>
<td></td>
<td>Unset an option</td>
</tr>
<tr>
<td>:colour :color :option</td>
<td></td>
<td>Turn REPL colours on or off; set a specific colour</td>
</tr>
<tr>
<td>:q :quit</td>
<td></td>
<td>Exit the Idris system</td>
</tr>
</tbody>
</table>
Listing 2: Sample Interactive Run

$ idris hello.idr

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

3 Types and Functions

3.1 Primitive Types

IDRIS defines several primitive types: Int, Integer and 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:

```idris
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
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:

```plaintext
data Nat = Z | S Nat -- Natural numbers

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: 

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

### 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:

```plaintext
-- 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))
4 : Nat
Idris> mult (S (S (S Z))) (plus (S (S Z)) (S (S Z)))
12 : Nat
```
Note: IDRIS automatically desugars the Nat representation into a more human readable format. The result of \( \text{plus} \ (S \ (S \ Z)) \ (S \ (S \ Z)) \) is actually \( (S \ (S \ (S \ (S \ Z)))) \) which is the Integer 4. This can be checked at the IDRIS prompt:

\[
\text{Idris} > \ (S \ (S \ (S \ (S \ Z))))
\]

4 : Nat

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

\[
\begin{align*}
\text{Idris} & > \ \text{plus} \ 2 \ 2 \\
4 & : \text{Nat} \\
\text{Idris} & > \ \text{mult} \ 3 \ \text{plus} \ 2 \ 2 \\
12 & : \text{Nat}
\end{align*}
\]

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

**where clauses**

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

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

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

**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:

```idris
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)]

```idris
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 follows⁴:

```idris
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 `Vects`:

```idris
(++) : 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:

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

Which when run through the Idris type checker results in the following:

```
$ idris vbroken.idr --check
vbroken.idr:3:23:When elaborating right hand side of vapp:
  When elaborating argument xs to constructor Prelude.Vect.::::
```

⁴Note that prior to Idris 0.9.9, the order of the arguments to `Vect` was reversed.
Can’t unify
   Vect (n + n) a
with
   Vect (plus n m) a

Specifically:
Can’t unify
   plus n n
with
   plus n m

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

### 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):

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

\( \text{FZ} \) is the zeroth element of a finite set with \( S \ k \) elements; \( \text{FS} \) \( n \) is the \( n+1 \)th element of a finite set with \( S \ k \) elements. \( \text{Fin} \) is indexed by a \( \text{Nat} \), which represents the number of elements in the set. Obviously we can’t construct an element of an empty set, so neither constructor targets \( \text{Fin Z} \).

A useful application of the \( \text{Fin} \) family is to represent bounded natural numbers. Since the first \( n \) natural numbers form a finite set of \( n \) elements, we can treat \( \text{Fin} n \) as the set of natural numbers bounded by \( n \).

For example, the following function which looks up an element in a \( \text{Vect} \), by a bounded index given as a \( \text{Fin} \ n \), is defined in the prelude:

```haskell
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 \( \text{Nil} \) here. This is because it is impossible. Since there is no element of \( \text{Fin Z} \), and the location is a \( \text{Fin} n \), 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 \( \text{index} \):

```haskell
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 \textit{implicit} arguments to \( \text{index} \). We could also write the type of \( \text{index} \) as:

```haskell
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 \texttt{Fin n} and \texttt{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:

\[
\text{index \{a=\text{Int}\} \{n=2\} FZ (2 :: 3 :: \text{Nil})}
\]

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

\[
\text{index : (i:Fin n) -> (xs:Vect n a) -> a}
\]

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 useful to provide types of implicit arguments, particularly where there is a dependency ordering, or where the implicit arguments themselves have dependencies. For example, we may wish to state the types of the implicit arguments in the following definition, which defines a predicate on vectors:

\[
\text{data Elem : a -> Vect n a -> Type where}
\]

- \texttt{Here} : \{x:a\} \rightarrow \{xs:Vect n a\} \rightarrow \text{Elem x (x :: xs)}
- \texttt{There} : \{x,y:a\} \rightarrow \{xs:Vect n a\} \rightarrow \text{Elem x xs} \rightarrow \text{Elem x (y :: xs)}

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

\[
\text{testVec : Vect 4 \text{Int}}
\text{testVec = 3 :: 4 :: 5 :: 6 :: \text{Nil}}
\]

\[
\text{inVect : Elem 5 testVec}
\text{inVect = There (There Here)}
\]

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

\[
\text{using (x:a, y:a, xs:Vect n a)}
\text{data Elem : a -> Vect n a -> Type where}
\]

- \texttt{Here} : \texttt{Elem x (x :: xs)}
- \texttt{There} : \texttt{Elem x xs} \rightarrow \text{Elem x (y :: xs)}

Note: Declaration Order and \texttt{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 \texttt{mutual} block, which allows data types and functions to be defined simultaneously:

\[
\text{mutual}
\]

- \texttt{even : Nat -> Bool}
- \texttt{even Z = True}
- \texttt{even (S k) = odd k}

11
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:

```haskell
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:

```haskell
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:

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

We can also read strings from user input:

```haskell
getLine : IO String
```

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

```haskell
data File -- abstract
data Mode = Read | Write | ReadWrite
openFile : String -> Mode -> IO File
closeFile : File -> IO ()
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 Laziness

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

    boolCase : Bool -> a -> a -> a;  
    boolCase True t e = t;  
    boolCase False t e = e;

This function uses one of the t or e arguments, but not both (in fact, this is used to implement the if...then...else construct as we will see later. We would prefer if only the argument which was used was evaluated. To achieve this, IDRIS provides a Lazy data type, which allows evaluation to be suspended:

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

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

    boolCase : Bool -> Lazy a -> Lazy a -> a;  
    boolCase True t e = t;  
    boolCase False t e = e;

3.8 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.

3.8.1 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.

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

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

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

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

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

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

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

You’ll find these examples in usefultypes.idr in the examples/ directory. For more details of the functions available on List and Vect, look in the library files:

- libs/prelude/Prelude/List.idr
- libs/prelude/Prelude/Vect.idr

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:

```idr
*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:

```idr
*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.
### 3.8.2 Maybe

Maybe describes an optional value. Either there is a value of the given type, or there isn’t:

```haskell
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:

```haskell
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:

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

Note that the type of the first argument is `Lazy b` rather than simply `b`. Since the default value might not be used, we mark it as `Lazy` in case it is a large expression where evaluating it then discarding it would be wasteful.

### 3.8.3 Tuples and Dependent Pairs

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

```haskell
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:

```haskell
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. Traditionally, these are referred to as “sigma types”:

```haskell
data Sigma : (A : Type) -> (P : A -> Type) -> Type where
  MkSigma : {P : A -> Type} -> (a : A) -> P a -> Sigma 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.

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

If you like, you can write it out the long way, the two are precisely equivalent.

```haskell
vec : Sigma Nat (\n => Vect n Int)
vec = MkSigma 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:

```haskell
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:

```haskell
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 \texttt{Vect} according to some predicate, we will
not know in advance what the length of the resulting vector will be:

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

If the \texttt{Vect} is empty, the result is easy:

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

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

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

We will see more on \texttt{with} notation later.

### 3.9 So

The \texttt{So} data type is a predicate on \texttt{Bool} which guarantees that the value is true:

```haskell
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
\texttt{So (inBounds x y)} guarantees that the point \((x,y)\) is within the bounds of a \(640 \times 480\) window:

```haskell
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
```

### 3.10 More Expressions

\texttt{let} bindings

Intermediate values can be calculated using \texttt{let} bindings:

```haskell
mirror : List a -> List a
mirror xs = let xs' = reverse xs in
            xs ++ xs'
```
We can do simple pattern matching in `let` bindings too. For example, we can extract fields from a record as follows, as well as by pattern matching at the top level:

```haskell
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:

```haskell
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:

```haskell
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:

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

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

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

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

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

3.11 Dependent Records

Records are data types which collect several values (the record’s fields) together. IDRI S 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:

```idris
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:

```idris
*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 updated).

```idris
*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:

```idris
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:

```idris
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
```
Nested record update

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

\[
\text{record } \{ a \to b \to c = \text{val} \} \ x
\]

This returns a new record, with the field accessed by the path `a \to b \to c` set to `x`. The syntax is first class, i.e. record `\{ a \to b \to c = \text{val} \}` itself has a function type. Symmetrically, the field can also be accessed with the following syntax:

\[
\text{record } \{ a \to b \to c \} \ x
\]

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`:

\[
\text{class Show a where}
\begin{align*}
\text{show} & : a \to \text{String}
\end{align*}
\]

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

\[
\text{show} : \text{Show} \ a \Rightarrow a \to \text{String}
\]

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:

\[
\text{instance Show Nat where}
\begin{align*}
\text{show} \ & Z = "Z" \\
\text{show} \ & (S \ k) = "s" \ ++ \ \text{show} \ k
\end{align*}
\]

\text{Idris} > \text{show (S (S (S Z)))}
\text{"sssZ" : 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`:

\[
\text{instance Show a \Rightarrow Show (Vect n a) where}
\begin{align*}
\text{show} \ & \text{xs} = "[" \ ++ \ \text{show'} \ \text{xs} \ ++ "]" \ \text{where}
\end{align*}
\]

\[
\text{show' : Vect n a \to String}
\begin{align*}
\text{show'} \ & \text{Nil} = ","
\text{show'} \ & (x :: \text{Nil}) = \text{show} \ x
\text{show'} \ & (x :: \text{xs}) = \text{show} \ x \ ++ "," \ ++ \ \text{show'} \ \text{xs}
\end{align*}
\]
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:

```haskell
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`:

```haskell
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:

```haskell
class Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool
    x /= y = not (x == y)
    x == 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:

```haskell
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 \texttt{Ord} class. We give the constraints on the type variables left of the fat arrow \texttt{=>}, and the function type to the right of the fat arrow:\texttt{:}

\[
\text{sort : Ord } a \Rightarrow \text{List } a \Rightarrow \text{List } a
\]

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

\[
\text{sortAndShow : (Ord } a, \text{Show } a) \Rightarrow \text{List } a \Rightarrow \text{String}
\]

\[
\text{sortAndShow } xs = \text{show (sort } xs)
\]

### 4.3 Monads and \texttt{do}-notation

So far, we have seen single parameter type classes, where the parameter is of type \texttt{Type}. In general, there can be any number (greater than 0) of parameters, and the parameters can have \textit{any} type. If the type of the parameter is not \texttt{Type}, we need to give an explicit type declaration. For example:

\[
\text{class Monad } (m : \text{Type } \rightarrow \text{Type}) \text{ where}
\]

\[
\text{return : } a \rightarrow m a
\]

\[
(a \rightarrow m b) \rightarrow m b
\]

The \texttt{Monad} class allows us to encapsulate binding and computation, and is the basis of \texttt{do}-notation introduced in Section 3.6. Inside a \texttt{do} block, the following syntactic transformations are applied:

- \texttt{x <- v; e} becomes \texttt{v >>= \{x => e\}}
- \texttt{v; e} becomes \texttt{v >>= \{}\_\texttt{=> e\}}
- \texttt{let x = v; e} becomes \texttt{let x = v in e}

\texttt{IO} is an instance of \texttt{Monad}, defined using primitive functions. We can also define an instance for \texttt{Maybe}, as follows:

\[
\text{instance Monad } \text{Maybe where}
\]

\[
\text{return } = \text{Just}
\]

\[
\text{Nothing } \gg= k = \text{Nothing}
\]

\[
(\text{Just } x) \gg= k = k \times
\]

Using this we can, for example, define a function which adds two \texttt{Maybe Ints}, using the monad to encapsulate the error handling:

\[
\text{m_add : Maybe Int } \rightarrow \text{Maybe Int } \rightarrow \text{Maybe Int}
\]

\[
\text{m_add } x \ y = \text{do } x' \leftarrow x \ -- \ \text{Extract value from } x
\]

\[
y' \leftarrow y \ -- \ \text{Extract value from } y
\]

\[
\text{return } (x' + y') \ -- \ \text{Add them}
\]

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

\[
*\text{classes}> m_add \ (\text{Just } 20) \ (\text{Just } 22)
\]

\[
\text{Just } 42 : \text{Maybe Int}
\]

\[
*\text{classes}> m_add \ (\text{Just } 20) \ \text{Nothing}
\]

\[
\text{Nothing} : \text{Maybe Int}
\]
!-notation

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

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

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

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

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

For example, the expression...

```haskell
let y = 42 in f !(g !(print y) !x)
```

...is lifted to:

```haskell
let y = 42 in do y' <- print y
        x' <- x
        g' <- g y' x'
        f g'
```

Monad comprehensions

The list comprehension notation we saw in Section 3.10 is more general, and applies to anything which is an instance of `MonadPlus`:

```haskell
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:

```haskell
guard : MonadPlus m => Bool -> m ()
```

Then the comprehension is converted to do notation:

```haskell
do { qual1; qual2; ...; qualn; return exp; }
```
Using monad comprehensions, an alternative definition for `m_add` would be:

```haskell
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:

```haskell
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`:

```haskell
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:

```haskell
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:

```haskell
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 al ...an |]` is translated into `pure f <$> al <$> ... <$> an`:

```haskell
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:
\[
\textbf{data} \ \text{Expr} = \text{Var} \ \text{String} \quad \text{-- variables} \\
| \ \text{Val} \ \text{Int} \quad \text{-- values} \\
| \ \text{Add} \ \text{Expr} \ \text{Expr} \quad \text{-- addition}
\]

Evaluation will take place relative to a context mapping variables (represented as Strings) to integer values, and can possibly fail. We define a data type \(\text{Eval}\) to wrap an evaluator:

\[
\textbf{data} \ \text{Eval} : \text{Type} \to \text{Type} \quad \text{where}
\]

\[
\text{MkEval} : (\text{List} \ (\text{String}, \ \text{Int}) \to \text{Maybe} \ a) \to \text{Eval} \ a
\]

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:

\[
\text{fetch} : \text{String} \to \text{Eval} \ \text{Int}
\]

\[
\text{fetch} \ x = \text{MkEval} \ (\lambda e \Rightarrow \text{fetchVal} \ e) \quad \text{where}
\]

\[
\text{fetchVal} : \text{List} \ (\text{String}, \ \text{Int}) \to \text{Maybe} \ \text{Int}
\]

\[
\text{fetchVal} \ [] = \text{Nothing}
\]

\[
\text{fetchVal} \ ((v, \ \text{val}) :: \ xs) = \text{if} \ (x == v)
\]

\[
\text{then} \ (\text{Just} \ \text{val})
\]

\[
\text{else} \ (\text{fetchVal} \ xs)
\]

When defining an evaluator for the language, we will be applying functions in the context of an \(\text{Eval}\), so it is natural to make \(\text{Eval}\) an instance of \text{Applicative}. Before \(\text{Eval}\) can be an instance of \text{Applicative} it is necessary to make \(\text{Eval}\) an instance of \text{Functor}:

\[
\text{instance} \ \text{Functor} \ \text{Eval} \ \text{where}
\]

\[
\text{map} \ f \ (\text{MkEval} \ g) = \text{MkEval} \ (\lambda \ e \Rightarrow \text{map} \ f \ (g \ e))
\]

\[
\text{instance} \ \text{Applicative} \ \text{Eval} \ \text{where}
\]

\[
\text{pure} \ x = \text{MkEval} \ (\lambda \ e \Rightarrow \text{Just} \ x)
\]

\[
(\langle\$\rangle) \ (\text{MkEval} \ f) \ (\text{MkEval} \ g) = \text{MkEval} \ (\lambda x \Rightarrow \text{app} \ (f \ x) \ (g \ x)) \quad \text{where}
\]

\[
\text{app} : \text{Maybe} \ (a \to b) \to \text{Maybe} \ a \to \text{Maybe} \ b
\]

\[
\text{app} \ (\text{Just} \ fx) \ (\text{Just} \ gx) = \text{Just} \ (fx \ gx)
\]

\[
\text{app} \ _ \ _ = \text{Nothing}
\]

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

\[
\text{eval} : \text{Expr} \to \text{Eval} \ \text{Int}
\]

\[
\text{eval} \ (\text{Var} \ x) = \text{fetch} \ x
\]

\[
\text{eval} \ (\text{Val} \ x) = [\mid x \mid]
\]

\[
\text{eval} \ (\text{Add} \ x \ y) = [\mid \text{eval} \ x + \text{eval} \ y \mid]
\]

\[
\text{runEval} : \text{List} \ (\text{String}, \ \text{Int}) \to \text{Expr} \to \text{Maybe} \ \text{Int}
\]

\[
\text{runEval} \ \text{env} \ \text{e} = \text{case} \ \text{eval} \ e \ \text{of}
\]

\[
\text{MkEval} \ \text{envFn} \Rightarrow \text{envFn} \ \text{env}
\]

\section{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 \textit{named} as follows:

\[
\text{instance} \ [\text{myord}] \ \text{Ord} \ \text{Nat} \ \text{where}
\]

\[
\text{compare} \ Z \ (S \ n) = \text{GT}
\]

\[
\text{compare} \ (S \ n) \ Z = \text{LT}
\]
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 Nats 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)
"[s0, sss0, ssss0]" : String
*named_instance> show (sort @(myord) testList)
"[ssss0, sss0, s0]" : 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) = btree.toList l ++ (v :: btree.toList r)

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

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.
module Main

import btree

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

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 \texttt{import} statement refers to a filename, using dots to separate directories. For example, \texttt{import foo.bar} would import the file \texttt{foo/bar.idr}, which would conventionally have the module declaration \texttt{module foo.bar}. The only requirement for module names is that the main module, with the \texttt{main} function, must be called \texttt{Main}—although its filename need not be \texttt{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. \texttt{IDRIS} allows functions, types and classes to be marked as: \texttt{public}, \texttt{abstract} or \texttt{private}:

- \texttt{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.
- \texttt{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.
- \texttt{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 \texttt{private}. For our \texttt{btree} module, it makes sense for the tree data type and the functions to be exported as \texttt{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) = \texttt{if} \ (x < v) \ \texttt{then} \ (Node (insert x l) v r) \ \texttt{else} \ (Node l v (insert x r))

abstract
toList : BTree a -> List a
toList Leaf = []
toList (Node l v r) = btree.toList l ++ (v :: btree.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 (y : Nat, xs : Vect x a)
    data Vects : Type -> Type where
      MkVects : Vect y a -> Vects a
    
    append : Vects a -> Vect (x + y) a
    append (MkVects ys) = xs ++ ys
To use Vects or append outside the block, we must also give the xs and ys arguments. Here, we can use placeholders for the values which can be inferred by the type checker:

```haskell
*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 

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

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

```haskell
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:

```haskell
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:

```haskell
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.

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:

```haskell
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 nth most recently defined variable is well-typed, so is the n+1th. 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:

```haskell
Var : HasType i G t -> Expr G t
```
Listing 6: Expression representation

```haskell
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 -> Lazy (Expr G a) -> Lazy (Expr G a) -> Expr G a
```

So, in an expression \( \lambda x. \lambda y. x \ y \), 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:

```haskell
Val : (x : Int) -> Expr G TyInt
```

A lambda creates a function. In the scope of a function of type \( a \rightarrow t \), there is a new local variable
of type \( a \), which is expressed by the context index:

```haskell
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 \( a \):

```haskell
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:

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

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

```haskell
If : Expr G TyBool -> Lazy (Expr G a) -> Lazy (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:

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

29
lookup : HasType i G t -> Env G -> interpTy t
lookup Stop (x :: xs) = x
lookup (Pop k) (x :: xs) = lookup k xs

Listing 7: Interpreter 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 IDRS 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 IDRS 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 IDRS constructs. For operators, we apply the function to its operands directly, and for If, we apply the IDRS 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, where fact (i.e. \( x \). if (x == 0) then 1 else (fact (x-1) * x)) is written as follows:
To finish, we write a main program which interprets the factorial function on user input:

```idris
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.

```
$ idris interp.idr
    ___/_______/_____
   /  __/    /  ___/
  /__/       /  __/
 / /          /  __/
 / /          /  __/
    ___\  __/__/\___
    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:

```idris
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.

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:

```idris
(+++) : 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:

\[
\text{filter} : (a -> \text{Bool}) -> \text{Vect} \ n \ a -> (p ** \text{Vect} \ p \ a)
\]
\[
\text{filter} \ p \ (x :: xs) \text{ with } (\text{filter} \ p \ xs)
\]
\[
\mid (\_ ** xs') = \text{if } (p \ x) \text{ then } (\_ ** x :: xs') \text{ else } (\_ ** xs')
\]

Here, the with clause allows us to deconstruct the result of \(\text{filter} \ p \ xs\). 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 Nats. Otherwise, it is the sum of two equal Nats plus one:

```idris
\text{data} \ \text{Parity} : \text{Nat} -> \text{Type} \ \text{where}
\text{Even} : \text{Parity} \ (n + n)
\text{Odd} : \text{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.

\[
\text{parity} : (n:\text{Nat}) -> \text{Parity} \ n
\]

We’ll come back to the definition of \text{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:

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

The value of the result of \text{parity} \ k affects the form of \(k\), because the result of \text{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 \text{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:

```idris
\text{data} \ (\text{=} : a -> b -> \text{Type} \ \text{where}
\text{Refl} : x = x
```

32
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:

\[
\begin{align*}
\text{fiveIsFive} : & \ 5 = 5 \\
\text{fiveIsFive} = & \ \text{Refl} \\
\text{twoPlusTwo} : & \ 2 + 2 = 4 \\
\text{twoPlusTwo} = & \ \text{Refl}
\end{align*}
\]

### 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:

\[
\begin{align*}
\text{disjoint} : & \ (n : \text{Nat}) \rightarrow \ Z = S \ n \rightarrow \ \text{Void} \\
\text{disjoint} \ n \ p = & \ \text{replace} \ {\ P = \text{disjointTy}} \ p \ () \\
\text{where} \\
\text{disjointTy} : & \ \text{Nat} \rightarrow \ \text{Type} \\
\text{disjointTy} \ Z = & \ () \\
\text{disjointTy} \ (S \ k) = & \ \text{Void}
\end{align*}
\]

There is no need to worry too much about how this function works — essentially, it applies the library function \(\text{replace}\), 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. \(\text{void}\) is defined in the library, to assist with proofs by contradiction.

\[
\begin{align*}
\text{void} : & \ \text{Void} \rightarrow \ a
\end{align*}
\]

### 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 \(\text{plus}\):

\[
\begin{align*}
\text{plusReduces} : & \ (n : \text{Nat}) \rightarrow \ \text{plus} \ Z \ n = n \\
\text{plusReduces} \ n = & \ \text{Refl}
\end{align*}
\]

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 \(\text{plus} \ Z \ n\) normalises to \(n\) by the definition of \(\text{plus}\):

\[
\begin{align*}
\text{plusReduces} \ n = & \ \text{Refl}
\end{align*}
\]

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

\[
\begin{align*}
\text{plusReducesZ} : & \ (n : \text{Nat}) \rightarrow \ n = \ \text{plus} \ n \ Z \\
\text{plusReducesZ} \ Z = & \ \text{Refl} \\
\text{plusReducesZ} \ (S \ k) = & \ \text{cong} \ (\text{plusReducesZ} \ k)
\end{align*}
\]
cong is a function defined in the library which states that equality respects function application:

\[
\text{cong} : (f : t \rightarrow u) \rightarrow a = b \rightarrow f \ a = f \ b
\]

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

\[
\begin{align*}
\text{plusReducesS} : (n: \text{Nat}) \rightarrow (m: \text{Nat}) \rightarrow S (\text{plus} \ n \ m) &= \text{plus} \ n \ (S \ m) \\
\text{plusReducesS} \ Z \ m &= \text{Refl} \\
\text{plusReducesS} \ (S \ k) \ m &= \text{cong} \ (\text{plusReducesS} \ k \ m)
\end{align*}
\]

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

\[
\begin{align*}
\text{plusReducesZ'} : (n: \text{Nat}) \rightarrow n &= \text{plus} \ n \ Z \\
\text{plusReducesZ'} \ Z &= ?\text{plusredZ}_Z \\
\text{plusReducesZ'} \ (S \ k) &= \text{let} \ ih = \text{plusReducesZ'} \ k \ \text{in} \\
&?\text{plusredZ}_S
\end{align*}
\]

On running Idris, two global names are created, \( \text{plusredZ}_Z \) and \( \text{plusredZ}_S \), 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) \rightarrow (k = plus k Z) \rightarrow 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 \( \text{hole0} \) 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):
Here, we could also have used the trivial tactic, which tries to refine by \texttt{Refl}, and if that fails, tries to refine by each name in the local context. When a proof is complete, we use the \texttt{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> refine Refl
plusredZ_Z: no more goals
```

```
-plusredZ_Z> qed
plusredZ_Z = proof
  compute
  refine Refl

*theorems> :m

Global metavariables:
  [\texttt{plusredZ_S}]
```

The \texttt{:m} 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, \texttt{plusredZ_S}:

```
*theorems> :p plusredZ_S
```

```
----------------------- (\texttt{plusredZ_S}) -------
{\text{hole0}} : (k : \text{Nat}) \to (k = \text{plus } k \ Z) \to S \ k = \text{plus } (S \ k) \ Z

In this case, the goal is a function type, using \texttt{k} (the argument accessible by pattern matching) and \texttt{ih} — the local variable containing the result of the recursive call. We can introduce these as premisses using the \texttt{intro} tactic twice (or \texttt{intros}, which introduces all arguments as premisses). This gives:

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

Since \texttt{plus} is defined by recursion on its first argument, the term \texttt{plus } (S \ k) \ Z in the goal can be simplified, so we use \texttt{compute}.

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

We know, from the type of \texttt{ih}, that \texttt{k} = \texttt{plus } k \ Z, so we would like to use this knowledge to replace \texttt{plus } k \ Z in the goal with \texttt{k}. We can achieve this with the \texttt{rewrite} tactic:

```
-plusredZ_S> rewrite ih
```

```
k : \text{Nat}
  ih : k = \text{plus } k \ Z
----------------------- (\texttt{plusredZ_S}) -------
{\text{hole3}} : S \ k = S \ k
```

The \texttt{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 {
```

```
```
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:

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

-- not terminating
empty2 : Void
empty2 = empty2
```

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

```
*theorems> :total empty1
possibly not total due to: empty1#hd
not total as there are missing cases

*theorems> :total empty2
possibly not total due to recursive path empty2
```

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

```idris
total empty2 : Void
empty2 = 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, I
DRIS 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 I
DRIS to print a warning for any undeclared partial function.

8.5.2 Totality checking issues

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

8.5.3 Hints for totality

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

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

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

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

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

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

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

In more extreme cases, the function assert_total marks a subexpression as always being total:
assert_total : a -> a
assert_total x = x

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

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:

```idr
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:

```idr
  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}
    | Odd = Odd {n=S j}
```

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

```
viewsbroken.idr:12:10:When elaborating right hand side of ViewsBroken.
parity:
  Can’t unify
    Parity (plus (S j) (S j))
  with
    Parity (S (S (plus j j)))
```

Specifically:

```
  Can’t unify
    plus (S j) (S j)
  with
    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:

```idris
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:

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

The first of these has the following type:

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

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:

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

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

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

After applying `intros` we have:

```idris
-views.parity_lemma_1> intros
j : Nat
value : Parity (S (plus j (S j)))
```

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

```idris
-views.parity_lemma_1> rewrite sym (plusSuccRightSucc j j)
j : Nat
value : Parity (S (plus j (S j)))
```

39
sym is a function, defined in the library, which reverses the order of the rewrite:

\[
\text{sym} : \text{l} = \text{r} \rightarrow \text{r} = \text{l}
\]

\[
\text{sym} \text{ Refl} = \text{Refl}
\]

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 \text{natToBin} function from Section 7.2 at the prompt. The number 42 is 101010 in binary. The binary digits are reversed:

\[
*\text{views}>\text{show (natToBin 42)}
\]

\[
"[False, True, False, True, False, True]" : \text{String}
\]

### 9.2 Suspension of Disbelief

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

\[
\text{believe_me} : \text{a} \rightarrow \text{b}
\]

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 \text{views.parity_lemma_1} using this is:

\[
\text{views.parity_lemma_2} = \text{proof}\{\text{intro; intro; exact believe_me value;}}\}
\]

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 \text{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 \text{Nat} equivalent. This is a common pattern, linking a representation (in this case \text{Binary}) with a meaning (in this case \text{Nat}):

\[
\text{data Binary : Nat} \rightarrow \text{Type where}
\]

\[
\text{bEnd : Binary Z}
\]

\[
\text{bO : Binary n} \rightarrow \text{Binary } (\text{n + n})
\]

\[
\text{bI : Binary n} \rightarrow \text{Binary } (\text{S (n + n)})
\]

\text{bO} and \text{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, \text{n + n} or \text{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 \text{Nat} to binary will state, in the type, that the resulting binary number is a faithful representation of the original \text{Nat}:

\[
\text{natToBin} : (\text{n:Nat}) \rightarrow \text{Binary } \text{n}
\]

40
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:

```idris
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:

```idris
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.

```idris
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`:

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

10 Interactive Editing

By now, we have seen several examples of how Idris’ dependent type system can give extra confidence in a function’s correctness by giving a more precise description of its intended behaviour in its type. We have also seen an example of how the type system can help with EDSL development by allowing a programmer to describe the type system of an object language. However, precise types give us more than verification of programs — we can also exploit types to help write programs which are correct by construction.

The Idris REPL provides several commands for inspecting and modifying parts of programs, based on their types, such as case splitting on a pattern variable, inspecting the type of a metavariable, and even a basic proof search mechanism. In this section, we explain how these features can be exploited by a text editor, and specifically how to do so in Vim\(^5\). An interactive mode for Emacs\(^6\) is also available.

10.1 Editing at the REPL

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

\(^5\)https://github.com/idris-hackers/idris-vim
\(^6\)https://github.com/idris-hackers/idris-emacs
That is, each command acts on a specific source line, at a specific name, and outputs a new program fragment. Each command has an alternative form, which updates the source file in-place:

:command! [line number] [name]

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

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

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

10.2 Editing Commands

10.2.1 :addclause

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

vzipWith : (a -> b -> c) -> Vect n a -> Vect n b -> Vect n c
then :ac 94 vzipWith will give:

vzipWith f xs ys = ?vzipWith_rhs

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

%name Vect xs, ys, zs, ws

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

10.2.2 :casesplit

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

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

then :cs 96 xs will give:

vzipWith f [] ys = ?vzipWith_rhs_1
vzipWith f (x :: xs) ys = ?vzipWith_rhs_2
That is, the pattern variable \( xs \) has been split into the two possible cases \([\]\) and \( x :: xs \). Again, the names are chosen according to the same heuristic. If we update the file (using \( \texttt{:cs!} \)) then case split on \( ys \) on the same line, we get:

\[
vzipWith \ f \ [] \ [] = \ ?vzipWith\_rhs\_3
\]

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

### 10.2.3 :addmissing

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

\[
vzipWith : (a \rightarrow b \rightarrow c) \rightarrow
\Vect n a \rightarrow \Vect n b \rightarrow \Vect n c
vzipWith \ f \ [] \ [] = \ ?vzipWith\_rhs\_1
\]

then \( \texttt{:am} 96 \ vzipWith \) gives:

\[
vzipWith \ f \ (x :: xs) \ (y :: ys) = \ ?vzipWith\_rhs\_2
\]

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

### 10.2.4 :proofsearch

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

\[
vzipWith : (a \rightarrow b \rightarrow c) \rightarrow
\Vect n a \rightarrow \Vect n b \rightarrow \Vect n c
vzipWith \ f \ [] \ [] = \ ?vzipWith\_rhs\_1
\]
\[vzipWith \ f \ (x :: xs) \ (y :: ys) = \ ?vzipWith\_rhs\_2\]

then \( \texttt{:ps} 96 \ vzipWith\_rhs\_1 \) will give:

\[
[]
\]

This works because it is searching for a \( \Vect \) of length 0, of which the empty vector is the only possibility. Similarly, and perhaps surprisingly, there is only one possibility if we try to solve \( \texttt{:ps} 97 \ vzipWith\_rhs\_2 \):

\[f \ x \ y :: (vzipWith \ f \ xs \ ys)\]

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

### 10.2.5 :makewith

The :makewith \( n \ f \) command, abbreviated \( \texttt{:mw} n \ f \), adds a with to a pattern clause. For example, recall parity. If line 10 is:
parity (S k) = ?parity_rhs

then :mw 10 parity will give:

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

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

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

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

10.3 Interactive Editing in Vim

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

- \d adds a template definition for the name declared on the current line (using :addclause).
- \c case splits the variable at the cursor (using :casesplit).
- \m adds the missing cases for the name at the cursor (using :addmissing).
- \w adds a with clause (using :makewith).
- \o invokes a proof search to solve the metavariable under the cursor (using :proofsearch).
- \p invokes a proof search with additional hints to solve the metavariable under the cursor (using :proofsearch).

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

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

Corresponding commands are also available in the Emacs mode. Support for other editors can be added in a relatively straightforward manner by using idris -client.

11 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.
11.1 syntax rules

We have seen if...then...else expressions, but these are not built in. Instead, we can define a
function in the prelude as follows (we have already seen this function in Section 3.7:

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

and then extend the core syntax with a syntax declaration:

```haskell
syntax "if" [test] "then" [t] "else" [e] = boolCase 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:

```haskell
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:

```haskell
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:

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

main : IO ()
main = do x <- [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.

### 11.2 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:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
(<$>) : |(f : Expr G (TyFun a t)) -> Expr G a -> Expr G t
($<$> = \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)

12 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.

12.1 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:

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

### 12.2 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:

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

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

In general, we cannot append a `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.

### 12.3 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:

```haskell
> module literate

This is a comment. The main program is below

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

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

### 12.4 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:

```haskell
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 construct mkForeign converts this description to a function callable by IDRIS:

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

mkForeign : Foreign x -> x

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

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

Include and linker directives

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

• %lib target "x" — include the libx library. If the target is C this is equivalent to passing the -lx option to gcc. If the target is Java the library will be interpreted as a groupId:artifactId:packaging:version dependency coordinate for maven.

• %include target "x" — use the header file or import x for the given back end target.

• %link target "x.o" — link with the object file x.o when using the given back end target.

• %dynamic "x.so" — dynamically link the interpreter with the shared object x.so.

Testing foreign function calls

Normally, the Idris interpreter (used for typechecking and at the REPL) will not perform IO actions. Additionally, as it neither generates C code nor compiles to machine code, the %lib, %include and %link directives have no effect. IO actions and FFI calls can be tested using the special REPL command :x EXPR, and C libraries can be dynamically loaded in the interpreter by using the :dynamic command or the %dynamic directive. For example:
12.5 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 \( \text{IO (Provider } t) \). 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\(^7\).

\(^7\)https://github.com/david-christiansen/idris-type-providers
12.6 C Target

The default target of IDRIS is C. Compiling via:

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

is equivalent to:

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

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

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

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

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

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

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

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

12.7 JavaScript Target

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

**Code Generation**

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

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

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

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

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

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

**Using the FFI**

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

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

One could use the primitive addition of JavaScript like so:
module Main

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

main : IO ()
main = do
  a <- primPlus 1 1
  b <- primPlus 1 2
  print (a, b)

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

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

```javascript
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:

```idris
module Main

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

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

The program outputs 3, just like we expected.

Including external JavaScript files

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

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

For NodeJS:

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

And for use in the browser:

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

The given files will be added to the top of the generated code.
Including NodeJS modules

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

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

This directive compiles into the following JavaScript

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

Shrinking down generated JavaScript

IDRIS can produce very big chunks of JavaScript code. However, the generated code can be minified using the closure-compiler from Google. Any other minifier is also suitable but the 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.

12.8 Cumulativity

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

```plaintext
*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

```plaintext
*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):

```plaintext
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:

```plaintext
myid : (a : Type) -> a -> a
myid _ x = x

idid : (a : Type) -> a -> a
idid = myid _ myid
```

The application of myid to itself leads to a cycle in the universe hierarchy — myid’s first argument is a Type, which cannot be at a lower level than required if it is applied to itself.

13 Further Reading

Further information about IDRIS programming, and programming with dependent types in general, can be obtained from various sources:
• The IDRIS web site (http://idris-lang.org/) and by asking questions on the mailing list.
• The IRC channel #idris, on chat.freenode.net.
• The wiki (https://github.com/idris-lang/Idris-dev/wiki/) has further user provided information, in particular:
  - https://github.com/idris-lang/Idris-dev/wiki/Language-Features
• Examining the prelude and exploring the samples in the distribution. The IDRIS source can be found online at: https://github.com/idris-lang/Idris-dev.
• Existing projects on the Idris Hackers web space: http://idris-hackers.github.io.
• Various papers (e.g. [1, 3, 4]). Although these mostly describe older versions of IDRIS.
References


