Programming in IDRIS: a tutorial

Edwin Brady
eb@cs.st-andrews.ac.uk

January 1, 2012

1 Introduction

In conventional programming languages, there is a clear distinction between *types* and *values*. For example, in Haskell [3], 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. The standard example is the type of lists of a given length[1], `Vect a n`, where `a` is the element type and `n` is the length of the list and can be an arbitrary term.

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

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

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 verifable *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 or OCaml. In particular, a certain amount of familiarity with Haskell syntax is assumed, although most concepts will at least be explained briefly. The reader is also assumed to have some interest in using dependent types for writing and verifying systems software.

1.2 Example Code

This tutorial includes some example code. The files are available in the IDRIS distribution, so that you can try them out easily, under `tutorial/examples`. However, it is strongly recommended that you type them in yourself, rather than simply loading and reading them.

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

2 Getting Started

2.1 Prerequisites

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

- A fairly recent Haskell platform. Version 2010.1.0.0.1 is currently sufficiently recent.
- The Boehm-Demers-Weiser garbage collector library. This is available in all major Linux distributions, or can be installed from source, available from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`. Installing from source is painless on MacOS.
- The GNU multiprecision arithmetic library, GMP, available from MacPorts and all major Linux distributions.

2.2 Downloading and Installing

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

```
cabal install idris
```

This will install the latest version released on Hackage[2], along with any dependencies. If, however, you would like the most up to date development version, you can find it on GitHub at `https://github.com/edwinb/Idris-dev`.

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

```
module main;

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

If you are familiar with Haskell, it should be fairly clear what the program is doing and how it works, but if not, we will explain the details later. You can compile the program to an executable by entering `idris hello.idr -o hello` at the shell prompt. This will create an executable called `hello`, which you can run:

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

Note that the `$` indicates the shell prompt! Some useful options to the `idris` command are:

- `-o prog` to compile to an executable called `prog`.
- `--check` type check the file and its dependencies without starting the interactive environment.

2.3 The interactive environment

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

[2] However version 0.9, which this tutorial describes, is not yet there, so don't do this. Finishing this tutorial is the last requirement before releasing it :-).

```
$ idris
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris>
```

This gives a `ghci`-style interface which allows evaluation of expressions, as well as type checking expressions, theorem proving, compilation, editing and various other operations. `:?` gives a list of supported commands, as shown in Figure 1. Figure 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`.

```
Idris version 0.9
-----------------

   Command          Arguments    Purpose

   <expr>                        Evaluate an expression
   :t               <expr>       Check the type of an expression
   :r :reload                    Reload current file
   :e :edit                      Edit current file using $EDITOR or $VISUAL
   :m :metavars                  Show remaining proof obligations (metavariables)
   :p :prove        <name>       Prove a metavariable
   :a :addproof                  Add last proof to source file
   :c :compile      <filename>   Compile to an executable <filename>
   :exec :execute   <filename>   Compile to an executable <filename> and run
   :? :h :help                   Display this help text
   :q :quit                      Exit the Idris system
```

Figure 1: Interactive environment commands

```
$ idris hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

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

Figure 2: Sample interactive run

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

3 Types and Functions

3.1 Primitive Types

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

```
module prims;

x : Int;
x = 42;

foo : String;
foo = "Sausage machine";

bar : Char;
bar = 'Z';

quux : Bool;
quux = False;
```

An IDRIS file consists of an optional module declaration (here `module prims`) followed by an optional list of imports (none here, however IDRIS programs can consist of several modules, and the definitions in each module each have their own namespace, as we will discuss shortly) and a collection of declarations and definitions. Each definition must have a type declaration (here, `x : Int`, `foo : String`, etc). Each component is separated by a semi-colon.

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

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

All of the usual arithmetic and comparison operators are defined for the primitive types. They are overloaded using type classes, as we will discuss in Section 4 and can be extended to work on user defined types. Boolean expressions can be tested with the `if...then...else` construct:

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

3.2 Data Types

Data types are declared in a similar way to Haskell data types, with a similar syntax. The main difference is that Idris syntax is not whitespace sensitive, and declarations must end with a semi-colon. Natural numbers and lists, for example, can be declared as follows:

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

The above declarations are taken from the standard library. Unary natural numbers can be either zero (`O` - that's a capital letter 'o', not the digit), 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 7 :: ;
```

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

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

3.3 Functions

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

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

-- Unary multiplication
mult : Nat -> Nat -> Nat;
mult O     y = O;
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 (`O`, `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 O)) (S (S O))
S (S (S (S O))) : Nat
Idris> mult (S (S (S O))) (plus (S (S O)) (S (S O)))
S (S (S (S (S (S (S (S (S (S (S (S O)))))))))))) : Nat
```

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

```
Idris> plus 2 2
S (S (S (S O))) : Nat
Idris> mult 3 (plus 2 2)
S (S (S (S (S (S (S (S (S (S (S (S O))))))))))) : Nat
```

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

`where` clauses

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

```
rev : List a -> List a;
rev xs = revAcc [] xs where {
  revAcc : List a -> List a -> List a;
  revAcc acc [] = acc;
  revAcc acc (x :: xs) = revAcc (x :: acc) xs;
}
```

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).

3.4 Dependent Types

3.4.1 Vectors

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

```
data Vect : Set -> Nat -> Set where
    Nil  : Vect a O
  | (::) : a -> Vect a k -> Vect a (S k);
```

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 type and a `Nat` as an argument, where `Set` stands for the type of types. We say that `Vect` is *parameterised* by a type, and *indexed* over `Nat`. Each constructor targets a different part of the family of types. `Nil` can only be used to construct vectors with zero length, and `::` to constructor vectors with non-zero length. In the type of `::`, we state explicitly that an element of type `a` and a tail of type `Vect a k` (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, `app`, defined in the library, appends two `Vect`s:

```
app : Vect A n -> Vect A m -> Vect A (n + m);
app Nil       ys = ys;
app (x :: xs) ys = x :: app xs ys;
```

The type of `app` 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:

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

$ idris vbroken.idr --check
vbroken.idr:3:Can't unify Vect a (S (plus k k)) with Vect a (S (plus k m))
```

This error message suggests that there is a length mismatch between two vectors — we needed a vector of length `(S (k + m))`, but provided a vector of length `(S (k + k))`. Note that the terms in the error message have been *normalised*, so in particular `n + m` has been reduced to `plus n m`.

3.4.2 The Finite Sets

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

```
data Fin : Nat -> Set where
   fO : Fin (S k)
 | fS : Fin k -> Fin (S k);
```

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

A useful application of the `Fin` family is to represent numbers with guaranteed bounds. For example, the following function which looks up an element in a `Vect` is defined in the prelude:

```
lookup : Fin n -> Vect a n -> a;
lookup fO     (x :: xs) = x;
lookup (fS k) (x :: xs) = lookup k xs;
```

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

Note also that there is no case for `Nil` here. This is because it is impossible. Since there is no element of `Fin O`, and the location is a `Fin n`, then `n` can not be `O`. 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 `O`.

3.4.3 Implicit Arguments

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

```
lookup : Fin n -> Vect a n -> 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 explictly. These are *implicit* arguments to `lookup`. We could also write the type of `lookup` as:

```
lookup : {a:Set} -> {n:Nat} -> Fin n -> Vect a n -> a
```

Implicit arguments, given in braces {} in the type declaration, are not given in applications of `lookup`; their values can be inferred from the types of the `Fin n` and `Vect a n` arguments. Any name 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:

```
lookup {a=Int} {n=2} fO (2 :: 3 :: VNil)
```

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

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

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

3.4.4 "`using`" notation

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

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

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

```
testVec : Vect Int 4;
testVec = 3 :: 4 :: 5 :: 6 :: VNil;

inVect : Elem 5 testVec;
inVect = there (there here);
```

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

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

3.5 I/O

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

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

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

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

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

```
getLine : IO String;
```

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

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

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

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

readFile : String -> IO String;
```

3.6 "`do`" notation

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

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

The syntax `x <- iovalue` executes the I/O operation iovalue, of type `IO a`, and puts the result, of type `a` into the variabel `x`. In this case, `getLine` returns an `IO String`, so `name` has type `String`. The `return` operation allows us to inject a value directly into an IO operation:

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

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

3.7 Useful Data Types

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

3.7.1 `List` and `Vect`

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

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

data Vect : Set -> Nat -> Set where
   Nil  : Vect a O
 | (::) : a -> Vect a k -> Vect a (S k);
```

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 `Cons 1 (Cons 2 (Cons 3 Nil))`

The library also defines a number of functions for manipulating these types. `map` is overloaded both for `List` and `Vect` and applies a function to every element of the list or vector.

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

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

For example, to double every element in a vector of integers:

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

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

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

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

For more details of the functions available on `List` and `Vect`, look in the library, in `prelude/list.idr` and `prelude/vect.idr` respectively. Functions include filtering, appending, reversing, and so on. Also remember that IDRIS is still in development, so if you don't see the function you need, please feel free to add it and submit a patch!

Aside: Anonymous functions and operator sections

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

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

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

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

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

3.7.2 Maybe

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

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

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

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

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

```
maybe : Maybe a -> |(default:b) -> (a -> b) -> b;
```

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

3.7.3 Tuples and Dependent Pairs

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

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

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

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

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

Dependent Pairs

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

```
data Exists : (A : Set) -> (P : A -> Set) -> Set where
   Ex_intro : {P : A -> Set} -> (a : A) -> P a -> Exists A P;
```

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

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

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

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

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

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

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

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

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

```
vfilter p VNil = (_ , []);
```

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

```
filter p (x :: xs) with filter p xs {
   | (_ , xs') = if (p x) then (_ , x :: xs') else (_ , xs');
}
```

We will see more on `with` notation later.

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 `Strings`:

```
class Show a where {
    show : a -> String;
}
```

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

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

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

```
instance Show Nat where {
    show O = "O";
    show (S k) = "s" ++ show k;
}

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

Instance declarations can themselves have constraints. For example, to define a `Show` instance for vectors, we need to know that there is a `Show` instance for the element type, because we are going to use it to convert each element to a `String`:

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

4.1 Default Definitions

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

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

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

```
instance Eq Nat where {
    O     == O     = True;
    (S x) == (S y) = x == y;
    O     == (S y) = False;
    (S x) == O     = False;

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

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

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

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

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

4.2 Extending Classes

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

```
data Ordering = LT | EQ | GT;

class Eq a => Ord a where {
    compare : a -> a -> Ordering;

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

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

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

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

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

4.3 Monads and do-notation

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

```
class Monad (m : Set -> Set) where {
    return : a -> m a;
    (>>=)  : m a -> (a -> m b) -> m b;
}
```

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

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

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

```
instance Monad Maybe where {
    return = Just;

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

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

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

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

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

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, Figure 3 gives a module which defines a binary tree type `BTree` (in a file `btree.idr`) and Figure 4 gives a main program (in a file `bmain.idr` which uses the `bst` module to sort a list.

```
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) = app (toList l) (v :: toList r);

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

Figure 3: Binary Tree module

```
module main;

import btree;

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

Figure 4: Binary Tree main program

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` and `btree.toTree`.

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

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

5.1 Export Modifiers

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

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

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

```
module btree;

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

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

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

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

Figure 5: Binary Tree module, with export modifiers

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
```

6 Example: The Well-Typed Interpreter

In this chapter, we'll use the features we've seen so far to write a larger example, an interpreter for a simple functional programming language, with variables, function application, binary operators and an `if...then...else` construct. We will use the dependent type system to ensure that any programs which can be represented are well-typed. First, let us define the types in the language. We have integers, booleans, and functions, represented by `Ty`:

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

We can write a function to translate these representations to a concrete IDRIS type:

```
interpTy : Ty -> Set;
interpTy TyInt = Int;
interpTy TyBool = Bool;
interpTy (TyFun A T) = interpTy A -> interpTy T;
```

We're going to define a representation of our language in such a way that only well-typed programs can be represented. We'll index the representations of expressions by their type and the types of local variables (the context), which we'll be using regularly as an implicit argument, so we define everything in a `using` block:

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

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

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

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

```
data HasType : (i : Fin n) -> Vect Ty n -> Ty -> Set where
    stop : HasType fO (t :: G) t
  | pop  : HasType k G t -> HasType (fS k) (u :: G) t;

data Expr : Vect Ty n -> Ty -> Set where
    Var : HasType i G t -> Expr G t
  | Val : (x : Int) -> Expr G TyInt
  | Lam : Expr (a :: G) t -> Expr G (TyFun a t)
  | App : Expr G (TyFun a t) -> Expr G a -> Expr G t
  | Op  : (interpTy a -> interpTy b -> interpTy c) -> Expr G a -> Expr G b ->
          Expr G c
  | If  : Expr G TyBool -> Expr G a -> Expr G a -> Expr G a;
```

Figure 6: Expression representation

`G` has type `T`. This is defined as follows:

```
data HasType : (i : Fin n) -> Vect Ty n -> Ty -> Set where
    stop : HasType fO (t :: G) t
  | pop  : HasType k G t -> HasType (fS k) (u :: G) t;
```

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

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

So, in an expression `\x. \y. x y`, the variable `x` would have a de Bruijn index of 1, represented as `pop stop`, and `y 0`, represented as `stop`. We find these by counting the number of lambdas between the definition and the use.
A value carries a concrete representation of an integer:

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

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

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

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

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

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

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

Finally, if expressions make a choice given a boolean. Each branch must have the same type:

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

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

```
data Env : Vect Ty n -> Set where
    Nil  : Env Nil
  | (::) : interpTy a -> Env G -> Env (a :: G);

lookup : HasType i G t -> Env G -> interpTy t;
lookup stop    (x :: xs) = x;
lookup (pop k) (x :: xs) = lookup k xs;
```

```
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);
```

Figure 7: Intepreter definition

Given this, an interpreter (Figure 7) is a function which translates an `Expr` into a concrete IDRIS value with respect to a specific environment:

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

To translate a variable, we simply look it up in the environment:

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

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

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

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

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

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

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

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

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

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

```
add : Expr G (TyFun TyInt (TyFun TyInt TyInt));
add = Lam (Lam (Op (+) (Var stop) (Var (pop stop))));
```

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

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

Then `fact` (i.e. `\x. if (x == 0) then 1 else (fact (x-1) * x)`) is written as follows:

```
fact : Expr G (TyFun TyInt TyInt);
fact = Lam (If (Op (==) (Var stop) (Val 0))
               (Val 1) (Op (*) (app fact (Op (-) (Var stop) (Val 1)))
                               (Var stop))));
```

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

```
main : IO ();
main = do { putStr "Enter a number: ";
            x <- getLine;
            print (interp [] fact (prim__strToInt x));
          };
```

Here, `prim__strToInt` is a primitive function which 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 Figure 8.

```
$ idris interp.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

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

Figure 8: Running the well-typed interpreter

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 `app`, we'd see that the form of the length argument was determined by whether the vector was empty or not:

```
app : Vect a n -> Vect a m -> Vect a (n + m);
app {n=O}   []        [] = [];
app {n=S k} (x :: xs) ys = x :: app 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 [2], which takes account of the fact that matching on a value in a dependently typed language can affect what we know about the forms of other values. In its simplest form, the `with` rule adds another argument to the function being defined, e.g. we have already seen a vector filter function, defined as follows:

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

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

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

```
data Parity : Nat -> Set where
   even : Parity (n + n)
 | odd  : Parity (S (n + n));
```

We say `Parity` is a *view* of `Nat`. It has a *covering function* which tests whether it is even or odd and constructs the predicate accordingly.

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

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

```
natToBin : Nat -> List Bool;
natToBin O = Nil;
natToBin k with parity k {
   natToBin (j + j)     | even = False :: natToBin j;
   natToBin (S (j + j)) | odd  = True  :: natToBin j;
}
```

The value of the result of `parity k` affects the form of `k`, because the result of `parity k` depends on `k`. So, as well as the patterns for the result of the intermediate computation (`even` and `odd`) right of the |, we also write how the results affect the other patterns left of the |. Note that there is a function in the patterns (+) and repeated occurrences of `j` — this is allowed because another argument has determined the form of these patterns.

We can test this function at the prompt. 42 is 101010 in binary. The binary digits are reversed with natToBin:

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

8 Theorem Proving

8.1 Equality

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

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

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

```
fiveIsFive : 5 = 5;
fiveIsFive = refl;

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

8.2 Simple Theorems

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

```
plusReduces : (n:Nat) -> plus O n = n;
```

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

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

```
plusReduces n = refl;
```

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

```
plusReducesO : (n:Nat) -> n = plus n O;
plusReducesO O = refl;
plusReducesO (S k) = eqRespS (plusReducesO k);
```

`eqRespS` is a function defined in the library which states that equality respects successor:

```
eqRespS : m = n -> S m = S n;
```

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

```
plusReducesS : (n:Nat) -> (m:Nat) -> S (plus n m) = plus n (S m);
plusReducesS O m = refl;
plusReducesS (S k) m = eqRespS (plusReducesS k m);
```

Even for trival theorems like these, the proofs are a little tricky to construct in one go. When things get even slightly more complicated, it becomes too much to think about to construct proofs in this 'batch mode'. IDRIS therefore provides an interactive proof mode.

8.3 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 `O` and `S`, with a recursive call in the `S` case giving the inductive hypothesis, and insert *metavariables* for the rest of the definition:

```
plusReducesO' : (n:Nat) -> n = plus n O;
plusReducesO' O     = ?plusredO_O;
plusReducesO' (S k) = let ih = plusReducesO' k in
                      ?plusredO_S;
```

On running IDRIS, two global names are created, `plusredO_O` and `plusredO_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 we can use the `:t` command to see what their types are:

```
*theorems> :m
Global metavariables:
        [plusredO_S,plusredO_O]

*theorems> :t plusredO_O
plusredO_O : O = plus O O

*theorems> :t plusredO_S
plusredO_S : (k : Nat) -> (k = plus k O) -> S k = S (plus k O)
```

The `:p` command enters interactive proof mode, which can be used to complete the missing definitions.

```
*theorems> :p plusredO_O

---------------------------------- (plusredO_O) --------
{hole0} : O = plus O O
```

This gives us a list of premisses (above the line; there are none here) and the current goal (below the line; named `{hole0}` here). At the prompt we can enter tactics to direct the construction of the proof. In this case, we can normalise the goal with the `compute` tactic:

```
-plusredO_O> compute

---------------------------------- (plusredO_O) --------
{hole0} : O = O
```

Now we have to prove that `O` equals `O`, which is easy to prove by `refl`. To apply a function, such as `refl`, we use the `refine` tactic which introduces subgoals for each of the function's explicit arguments (though `refl` has none):

```
-plusredO_O> refine refl
plusredO_O: no more goals
```

Here, we could also have used the `trivial` tactic, which tries to refine by `refl`, and if that fails, tries to refine by each name in the local context. When a proof is complete, we use the `qed` tactic to add the proof to the global context, and remove the metavariable from the unsolved metavariables list. This also outputs a trace of the proof:

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

*theorems> :m
Global metavariables:
        [plusredO_S]
```

The `:addproof` command, at the interactive prompt, will add the proof to the source file (effectively in an appendix). Let us now prove the other required lemma, `plusredO_S`:

```
*theorems> :p plusredO_S

---------------------------------- (plusredO_S) --------
{hole0} : (k : Nat) -> (k = plus k O) -> S k = S (plus k O)
```

In this case, the goal is a function type, using `k` (the argument accessible by pattern matching) and `ih` (the local variable containing the result of the recursive call). We can introduce these as premisses using the `intro` tactic twice. This gives:

```
  k : Nat
  ih : k = plus k O
---------------------------------- (plusredO_S) --------
{hole2} : S k = S (plus k O)
```

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

```
-plusredO_S> rewrite ih

  k : Nat
  ih : k = plus k O
---------------------------------- (plusredO_S) --------
{hole3} : S k = S k

-plusredO_S>
```

The `rewrite` tactic takes an equality proof as an argument, and tries to rewrite the goal using that proof. Here, it results in an equality which is trivially provable:

```
-plusredO_S> trivial
plusredO_S: no more goals
-plusredO_S> qed
plusredO_S = proof {
    intro;
    intro;
    rewrite ih;
    trivial;
};
```

Again, we can add this proof to the end of our source file using the `:addproof` command at the interactive prompt.

9 Provisional Definitions

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

```
data Parity : Nat -> Set where
   even : Parity (n + n)
 | odd  : Parity (S (n + n));

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

We'd like to implement this as follows:

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

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

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

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

9.1 Provisional definitions

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

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

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

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

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

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

The first of these has the following type:

```
*views> :p views.parity_lemma_1

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

-views.parity_lemma_1>
```

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

```
plusn_Sm : (n : Nat) -> (m : Nat) -> (plus n (S m)) = S (plus n m);
```

After applying `intro` twice, we have:

```
-views.parity_lemma_1> intro

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

Then we apply the `plusn_Sm` function to `j` and `j`, giving:

```
-views.parity_lemma_1> rewrite plusn_Sm j j

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

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

9.2 Suspension of Disbelief

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

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

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

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

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

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

9.3 Example: Binary numbers

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

```
data Binary : Nat -> Set where
   bEnd : Binary O
 | bO : Binary n -> Binary (n + n)
 | bI : Binary n -> Binary (S (n + n));
```

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

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

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

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

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

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

```
natToBin_lemma_1 = proof {
    intro;
    intro;
    rewrite plusn_Sm j j;
    trivial;
};
```

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

```
main : IO ();
main = do { putStr "Enter a number: ";
            x <- getLine;
            let n = prim__strToInt x;
            let b = natToBin (fromInteger n);
            print b;
          };
```

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

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

10 Syntax Extensions

11 Miscellany

Small things which don't quite fit elsewhere:

- Literate programming
- Foreign functions

11.1 Comparison

How does IDRIS compare with other dependently typed languages and proof assistants, such as Coq, Agda and Epigram?

12 Further Reading

References

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

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

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