Programming in IDRIS: a tutorial

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

December 30, 2011

1 Introduction

In conventional programming languages, there is a clear distinction between *types* and *values*. For example, in Haskell [1], 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] 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 Language Overview

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 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`.

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 runtime 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

Lists, syntactic sugar. Maybe, Either. Pairs, dependent pairs, syntactic sugar.

4 Type Classes

5 Modules and Namespaces

6 Example: The Well-Typed Interpreter

7 Views

8 Theorem Proving

9 Provisional Definitions

10 Syntax Extensions

11 Miscellany

Small things which don't quite fit elsewhere:

- Namespaces (other than modules)
- Literate programming
- Totality checking
- Foreign functions

11.1 Comparison

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

References

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