Idris
A Programming Language with Dependent Types

Sumner Evans and Sam Sartor
2018-03-22

Mines Linux Users Group
https://lug.mines.edu

What is Idris?

"Idris is a general purpose pure functional programming language with dependent types."

The Idris Website

- **Version 0.1.3** of Idris was released in December of 2009.
- **Version 1.2.0** is the latest stable release and was released on January 9, 2018.
- Idris was named after the singing dragon in the 1970s UK children's television program *Ivor the Engine*.
- Idris development is led by Edwin Brady at the University of St. Andrews.

The Obligatory Picture of This Madman

Properties of Idris

- Idris can be **interpreted, transpiled, or compiled**
- Idris is **statically typed**
- Idris is **strongly typed**
- Idris is **purely functional** (much like Haskell)
- Idris has **first class types** (types can be treated as data)
- Idris has **dependent types** (the types are all high on something)

Idris Features

Idris is a general purpose language, and thus it has a lot of features. We will focus on the following aspects of the language.

- Dependent Types
- Haskell-like Syntax
- Proof Assistant

Types

Idris has familiar, Haskell-ish types:

- `Nat` - A natural number
- `Bool` - A boolean
- `Char` - A single charecter
- `List Int` - A list of integers
- `Nat -> Bool` - A function that takes a natural number and produces a boolean
- `(Nat, Nat)` - A tuple of two natural numbers
- `Int -> Int -> Int` - A function that takes two arguments

Types as Data

Unlike Haskell, data types can be stored, passed, and constructed like data:

```
an_int : Int
an_int = 5

a_type : Type
a_type = Int
```

We could write a function to choose between an `Int` and a `Nat`:

```
PickInt : Bool -> Type
PickInt True = Int
PickInt False = Nat
```

This is called a **type constructor**.

Dependent Types

Any expression that returns Type can be used as type itself:

```
foo : PickInt (True && False)
foo = 5

bar : case False of
    True => List Char
    False => String
bar = "Hello, World!"
```

These are called **dependent types**, since they *depend* on data.

Useful Dependent Types

`List` and `Vect` are examples of type constructors:

- `List Int` is a dynamically sized list of integers.
- `Vect 10 Int` is a list of exactly 10 integers.

Since type constructors are simply functions, they support things like currying:

```
TwoOf : Type -> Type
TwoOf = Vect 2
```

The Equality Type Constructor

The basis for proofs in Idris is the `(=)` function. It takes two inputs, and the return type is a proof that the two inputs have the same value.

- Any **`Nat`** is a natural number.
- Any **`Vect`** `2` **`Nat`** is a list of two natural numbers.
- Any `(=) (2 + 2) 4` is a proof that 2+2 and 4 have the same value.
- Any `1 = 3` is a proof that 1 and 3 have the same value.
- Any `even x =` **`True`** is a proof that *x* is even

Idris Syntax: Function Signatures

The Idris function signature syntax is *very* similar to the Haskell function signature syntax. Here are a few examples of Idris function signatures:

```
even : Nat -> Bool
add : Nat -> Nat -> Nat
foo : (a:Nat) -> (b:Nat) -> a = b
bar : (a:Nat) -> (b:Nat) -> LTE a b
```

If you are familiar with Haskell, you will note the use of `:` rather than `::`. This makes it look a bit more like a mathematical function definition:

$$f : \mathbb{N} \to \mathbb{N}.$$

You will also note that instead of the `(Type x) => x` syntax, it uses a more concise `(x:Type)` syntax.

Idris Syntax: Currying and Pattern Matching

Because of its foundation in Lambda Calculus, all functions only take a single argument. We can still handle multiple arguments using *currying*. For example, the `plus` operator is defined as follows:

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

Like Haskell, functions are implemented using *pattern matching*.

Idris Syntax: Type Definition Syntax

Idris defines several primitives including `Int`, `Integer`, `Double`, `Char`, `String`, and `Ptr`.

There are a bunch of other data types defined in the standard library including `Nat` and `Bool`.

Idris allows programmers to define their own data types. Again, the syntax is similar to Haskell.

```
data Nat    = Z   | S Nat
data List a = Nil | (::) a (List a)
```

Idris Syntax: Holes

Idris allows you to leave some of your code unfinished. For example, if we write the following code in a file called `even.idr`:

```
even : Nat -> Bool
even Z = True
even (S k) = ?even_rhs
```

And then load it into Idris:

```
:Idris> :l even
Holes: even_rhs
even> :t even_rhs
  k : Nat
--------------------------------------
even_rhs : Bool
Holes: even_rhs
```

Using Idris as a Proof Assistant

A proof assistant is a software tool to assist with the development of formal proofs by human-machine collaboration.

The Idris type system is robust enough that it can be used as a proof assistant.

Recall from above that equality is a type constructor. This means that we can pass equalities in and out of functions. This is the basis for all proofs in Idris.

Take this example function declaration:

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

This is a function which takes any $n \in \mathbb{N}$, and returns a proof that $0 + n = n$. Any successful implementation of this function will prove that $0 + n = n$.

Demo

Warning

LIVE DEMO AHEAD

We are not responsible for any harm done to your brain by viewing the following code.

Evaluation of Idris as a Language

Writability: very expressive, relatively small grammar, many abstractions, *very* orthogonal

Readability: grammar is simple but sometimes too compact, abstractions are common (but sometimes too magical), *very* orthogonal

Reliability: purely functional, strongly and statically typed, uses the `IO` monad model

Feasibility: interpreter/compiler is widely available (there's a `pacman` package) and supports many targets, tooling is good

Evaluation of Idris as a Proof Assistant

Writability: holes are useful, but filling them is hard due to an insane degree of formality

Readability: proof logic is obscure and hard to follow

Reliability: it is *extremely* reliable (too reliable, in fact)

Feasibility: proofs involving large numbers are extremely slow to compile and cannot be multithreaded

Quotes From Our Exploration

"The concept of a programming language in which the possibility of inline assembly is an entirely foreign concept hurts my brain."

"Where do I put it? Do I put it in the type?"

"When your Rust program compiles, you know it won't segfault, or give you any undefined behavior at runtime. When your Idris program compiles, you throw away your executable, and publish your dissertation."

Questions?