

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 has **first class functions**, much like Haskell.
- Idris has **first class types**. This means that types can be treated as data.

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

Dependent Types

Consider these two arrays:

[1, 2, 3]

[1, 2, 3, 4]

What are their types? The first is a 3-dimensional array, the second is 4-dimensional.

The Equality Type

The basis for proofs in Idris is the $(=)$ type constructor. It constructs a type that is dependent on two expressions.

- 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.

It is impossible to create an instance of $1 = 3$

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:

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

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 $0 + n = n$.

Warning

LIVE DEMO AHEAD

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

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?