An Intro to Dependent Types with Idris

Thomas Gebert

August 9, 2016

Problems with regular type systems

Haskell, F#, and OCaml all have great type systems, but still there are still holes that can be improved on.

Problems with regular type systems

Haskell, F#, and OCaml all have great type systems, but still there are still holes that can be improved on.

```
head []
-- *** Exception: Prelude.head: empty list

printf "blah %s %s" "hello"
-- *** Exception: printf: argument list ended
-- prematurely
```

What causes these problems?

There are two main reasons that this happens

What causes these problems?

There are two main reasons that this happens

- Most functional languages have two languages that can't interact: the Type language, and the programming language

What causes these problems?

There are two main reasons that this happens

- Most functional languages have two languages that can't interact: the Type language, and the programming language
  - Types exist only as an enforcement layer

What causes these problems?

There are two main reasons that this happens

- Most functional languages have two languages that can't interact: the Type language, and the programming language
  - Types exist only as an enforcement layer
- Since these languages can't interact, all the types and conditions for them must be known by the programmer ahead of time, and can't can't be deduced from the context of the code.

The solution to these problems (and many others) are dependent types

So what the hell is a dependent type?

A dependently typed language generally means two things

So what the hell is a dependent type?

A dependently typed language generally means two things

- Types, like functions, are first-class citizens that can be built dynamically (without it being dynamic typing)

So what the hell is a dependent type?

A dependently typed language generally means two things

- Types, like functions, are first-class citizens that can be built dynamically (without it being dynamic typing)
- Types (return types, input types, etc) can change depending on values

So what the hell is a dependent type?

A dependently typed language generally means two things

- Types, like functions, are first-class citizens that can be built dynamically (without it being dynamic typing)
- Types (return types, input types, etc) can change depending on values
- Functions can be called inside the type signature

So what the hell is a dependent type?

A dependently typed language generally means two things

- Types, like functions, are first-class citizens that can be built dynamically (without it being dynamic typing)
- Types (return types, input types, etc) can change depending on values
- Functions can be called inside the type signature

Common languages for it are:

- Agda
- Coq
- F*

Enter Idris

What is Idris?

Enter Idris

What is Idris?

- Idris is a Haskell-like language by Edwin Brady
  - Similar syntax, but not lazy

Enter Idris

What is Idris?

- Idris is a Haskell-like language by Edwin Brady
  - Similar syntax, but not lazy
- Dependent types are a main feature, but less dogmatic than Agda
  - Long-term goals are dependently typed system drivers

Enter Idris

What is Idris?

- Idris is a Haskell-like language by Edwin Brady
  - Similar syntax, but not lazy
- Dependent types are a main feature, but less dogmatic than Agda
  - Long-term goals are dependently typed system drivers
- Outputs to C, LLVM, JavaScript, and PHP
  - Making a new backend can usually be done in <500 LOC

A safe list with a length

Lists are annoying for a couple reasons

A safe list with a length

Lists are annoying for a couple reasons

- The aforementioned issue with unexpected empty lists causes runtime errors

A safe list with a length

Lists are annoying for a couple reasons

- The aforementioned issue with unexpected empty lists causes runtime errors

```
head []
-- *** Exception: Prelude.head: empty list
zipWith (\i j -> i + j) [1,2,3,4] [1]
-- [2]
```

A safe list with a length

Lists are annoying for a couple reasons

- The aforementioned issue with unexpected empty lists causes runtime errors

```
head []
-- *** Exception: Prelude.head: empty list
zipWith (\i j -> i + j) [1,2,3,4] [1]
-- [2]
```

- Even getting the length requires O(n) operations.

A safe list with a length

Lists are annoying for a couple reasons

- The aforementioned issue with unexpected empty lists causes runtime errors

```
head []
-- *** Exception: Prelude.head: empty list
zipWith (\i j -> i + j) [1,2,3,4] [1]
-- [2]
```

- Even getting the length requires O(n) operations.
  - You could store the length as a property, but that requires anyone who updates length to make sure it's updated
    - Generally ok for well-audited things, risky for anything else.

A safe list with a length

Let's code it!

A safe printf (or any variadic function)

Variadic functions (in most langauges) are convenient, but terrible.

A safe printf (or any variadic function)

Variadic functions (in most langauges) are convenient, but terrible.

- Mixing types (like in printf) generally eschews compiler type-safety

A safe printf (or any variadic function)

Variadic functions (in most langauges) are convenient, but terrible.

- Mixing types (like in printf) generally eschews compiler type-safety
- Compiler can't check to see if you have the correct number of arguments

A safe printf (or any variadic function)

More coding!

A note about compiler hacks

F#'s printfn works as you would expect due to a special case in the compiler doing static analysis on that particular case.

A note about compiler hacks

F#'s printfn works as you would expect due to a special case in the compiler doing static analysis on that particular case.

This is totes OK, but that only works for that particular case.

Proofs

Idris (and most dependently typed languages) allows for mathematical proofs of program correctness.

Proofs

Idris (and most dependently typed languages) allows for mathematical proofs of program correctness.

- Quick example

```
fiveIsFive : 5 = 5
fiveIsFive = Refl
```

Proofs

Idris (and most dependently typed languages) allows for mathematical proofs of program correctness.

- Quick example

```
fiveIsFive : 5 = 5
fiveIsFive = Refl
```

- Very cool but a bit beyond the scope of this talk

Proofs

Idris (and most dependently typed languages) allows for mathematical proofs of program correctness.

- Quick example

```
fiveIsFive : 5 = 5
fiveIsFive = Refl
```

- Very cool but a bit beyond the scope of this talk
  - (I'm still learning how to use them)

What's the point of any of this?

I don't expect everyone here to convert all their stuff to Idris, so why am I talking?

What's the point of any of this?

I don't expect everyone here to convert all their stuff to Idris, so why am I talking?

- The ideas in Idris (Agda, F*) could conceivably be ported to more mainstream languages

What's the point of any of this?

I don't expect everyone here to convert all their stuff to Idris, so why am I talking?

- The ideas in Idris (Agda, F*) could conceivably be ported to more mainstream languages
  - Scala actually has basic support for dependent types.

What's the point of any of this?

I don't expect everyone here to convert all their stuff to Idris, so why am I talking?

- The ideas in Idris (Agda, F*) could conceivably be ported to more mainstream languages
  - Scala actually has basic support for dependent types.
- Having increased program safety is always a good thing

What's the point of any of this?

I don't expect everyone here to convert all their stuff to Idris, so why am I talking?

- The ideas in Idris (Agda, F*) could conceivably be ported to more mainstream languages
  - Scala actually has basic support for dependent types.
- Having increased program safety is always a good thing
- Because dependent types are super cool, and worth a ton of further research.

Questions

Questions?