Type Driven Development with Idris

Lecture 1: Introducing Idris

Edwin Brady (`ecb10@st-andrews.ac.uk`)
University of St Andrews, Scotland, UK
`@edwinbrady`

MGS, April 9th 2018

Idris

IDRIS is a *Pac-man Complete* functional programming language with *dependent types*

- `cabal update; cabal install idris`
- `http://idris-lang.org/download`

In these talks:

- A *tutorial* on programming with dependent types in Idris (Lectures 1–3)
- Progress towards *implementing* a dependently typed language in Idris (Lecture 4)
- Slides and code: `https://tinyurl.com/idrismgs18`

Outline

1. Introduction
    - Introduction to *Type-driven Development*
    - Interactive editing in Atom
    - Programs and *proofs*
2. Total Functional Programming
    - Termination and Productivity Checking
    - Streams and Interactive Programs
    - Example: Concurrency
3. Working with State
    - State-aware Domain Specific Languages
    - Generic state-handling: `Control.ST`
4. Implementing a Dependently Typed Language
    - The Core Language, *TT*
    - *Elaboration*
    - A better implementation: Idris in Idris . . . ?

Why types?

We can use *type systems* for:

- *Checking* a program has the intended properties
- *Guiding* a programmer towards a correct program
- Building *expressive* and *generic* libraries

Why types?

We can use *type systems* for:

- *Checking* a program has the intended properties
- *Guiding* a programmer towards a correct program
- Building *expressive* and *generic* libraries

Type Driven Development puts *types* first. Three steps:

- *Type*: Write a type for a function
- *Define*: Create a (possibly incomplete) implementation
- *Refine*: Improve/complete the implementation

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .
- *First class* dependent types
    - Functions can compute types, types can contain values

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .
- *First class* dependent types
    - Functions can compute types, types can contain values
- Interfaces
    - Similar to *type classes* in Haskell (but no `deriving`)
    - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .
- *First class* dependent types
  - Functions can compute types, types can contain values
- Interfaces
  - Similar to *type classes* in Haskell (but no `deriving`)
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets
- *Strict* evaluation order, unlike Haskell
  - `Lazy` as a type

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .
- *First class* dependent types
  - Functions can compute types, types can contain values
- Interfaces
  - Similar to *type classes* in Haskell (but no `deriving`)
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets
- *Strict* evaluation order, unlike Haskell
  - `Lazy` as a type
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation

Idris

- Benefits from decades of research in *lambda calculus*, *type theory*, *functional programming* . . .
- *First class* dependent types
  - Functions can compute types, types can contain values
- Interfaces
  - Similar to *type classes* in Haskell (but no `deriving`)
  - `Functor`, `Applicative`, `Monad`, `do` notation, idiom brackets
- *Strict* evaluation order, unlike Haskell
  - `Lazy` as a type
- Compiled (via C, Javascript, . . .)
  - Optimisations: aggressive erasure, inlining, partial evaluation
- Foreign functions, system interaction

Idris

Demonstration: Introductory Examples

sicsa*