Dependently Typed Functional Programming with Idris
Lecture 1: A Tour of Idris

Edwin Brady
University of St Andrews

ecb10@st-andrews.ac.uk
@edwinbrady
Idris is a pure functional programming language with dependent types

- cabal update; cabal install idris
- http://idris-lang.org
Idris is a pure functional programming language with dependent types

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

In these lectures:

1. Today: An introduction to the language
   - Software correctness - why Idris?
   - Language overview; programming idioms; theorem proving

2. Some examples: verification, EDSLs

3. Dealing with effects and resources

4. Language implementation
Idris is a pure functional programming language with dependent types

- cabal update
- http://idris-lang.org

In these lectures:

1. Today: An introduction
   - Software correctness - why Idris?
   - Language overview; programming idioms; theorem proving
2. Some examples: verification, EDSLs
3. Dealing with effects and resources
4. Language implementation
Preview quit unexpectedly.
Click Reopen to open the application again. Click Report to see more detailed information and send a report to Apple.

Ignore  Report...  Reopen
General purpose programming language
  Compiled, supports foreign functions, ...
**Idris** Overview

- *General purpose* programming language
  - Compiled, supports foreign functions, . . .
- Influenced by *Haskell*
  - Pattern matching, *where*, . . .
  - Type classes, *do*-notation, comprehensions, . . .
**Idris Overview**

- **General purpose** programming language
  - Compiled, supports foreign functions, ... 
- Influenced by *Haskell*
  - Pattern matching, `where`, ...
  - Type classes, `do`-notation, comprehensions, ...
- Has **full dependent types**
  - Types may be *predicated* on values
  - Can encode (and check) program *properties*
  - Supports *tactic based* theorem proving
General purpose programming language
- Compiled, supports foreign functions, ...

Influenced by Haskell
- Pattern matching, where, ...
- Type classes, do-notation, comprehensions, ...

Has full dependent types
- Types may be predicated on values
- Can encode (and check) program properties
- Supports tactic based theorem proving

Support for Embedded Domain Specific Languages
- Syntax overloading, dsl notation
Unary natural numbers

data Nat = O | S Nat
**Unary natural numbers**

```idris
data Nat = O | S Nat
```

**Polymorphic lists**

```idris
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```
Dependent Types in Idris

Unary natural numbers

data Nat = O \mid S \text{Nat}

Polymorphic lists

data List : Type \rightarrow Type where
  Nil : List a
  (::) : a \rightarrow List a \rightarrow List a

Vectors — polymorphic lists with length

data Vect : Type \rightarrow Nat \rightarrow Type where
  Nil : Vect a O
  (::) : a \rightarrow Vect a k \rightarrow Vect a (S k)
Append

\((++): Vect \ a \ m \rightarrow Vect \ a \ n \rightarrow Vect \ a \ (m + n)\)

\((++) \ [] \ ys = ys\)

\((++) \ (x::xs) \ ys = x :: xs ++ ys\)
### Append

\[
(++) : \text{Vect } a \text{ m } \rightarrow \text{Vect } a \text{ n } \rightarrow \text{Vect } a \text{ (m + n)}
\]

\[
(++) \; [] \; ys = ys
\]

\[
(++) \; (x::xs) \; ys = x :: xs ++ ys
\]

### Pairwise addition

\[
vAdd : \text{Num } a \Rightarrow \text{Vect } a \text{ n } \rightarrow \text{Vect } a \text{ n } \rightarrow \text{Vect } a \text{ n}
\]

\[
vAdd \; [] \; [] = []
\]

\[
vAdd \; (x :: xs) \; (y :: ys) = x + y :: vAdd \; xs \; ys
\]
Dependent Types — Examples

Append

\[
(++) : \text{Vect } a \ m \to \text{Vect } a \ n \to \text{Vect } a \ (m + n)
\]
\[
(++) \ [] \ ys = ys
\]
\[
(++) \ (x::xs) \ ys = x :: xs ++ ys
\]

Pairwise addition

\[
\text{total}
\]
\[
vAdd : \text{Num } a \to \text{Vect } a \ n \to \text{Vect } a \ n \to \text{Vect } a \ n
\]
\[
vAdd \ [] \ [] = []
\]
\[
vAdd \ (x :: xs) \ (y :: ys) = x + y :: vAdd \ xs \ ys
\]
Why Dependent Types?

Precise types

\[ \text{sort} : \text{List Int} \rightarrow \text{List Int} \]
Precise types

sort : Vect Int n -> Vect Int n
Why Dependent Types?

**Precise types**

\[
\text{sort} : (xs : \text{Vect Int } n) \rightarrow (ys : \text{Vect Int } n \leftrightarrow \text{Permutation } xs \ ys)
\]
Why Dependent Types?

We can make types as precise as we require.

- However, precise types may require complex implementations/proofs.
In this course, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
In this course, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
- Generic programming
  - First class types, so types can be calculated by programs
In this course, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
- Generic programming
  - First class types, so types can be calculated by programs
- Expressivity
  - Using types to make more expressive libraries
In this course, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
- Generic programming
  - First class types, so types can be calculated by programs
- Expressivity
  - Using types to make more expressive libraries
  - (Or: why I will never write a monad transformer again)
There are several dependently typed languages available (e.g. Agda, Coq, Epigram, . . .). Why did I make Idris?

- We are still *learning* about dependent types
  - Plenty of scope for experimenting, still
There are several dependently typed languages available (e.g. Agda, Coq, Epigram, . . .). Why did I make Idris?

- We are still learning about dependent types
  - Plenty of scope for experimenting, still
- Freedom to make language design decisions
  - e.g. partial functions, high level notation, primitive types, evaluation strategy, . . .
Why Idris?

There are several dependently typed languages available (e.g. Agda, Coq, Epigram, ...). Why did I make Idris?

- We are still learning about dependent types
  - Plenty of scope for experimenting, still
- Freedom to make language design decisions
  - e.g. partial functions, high level notation, primitive types, evaluation strategy, ...  
- Desire to experiment with practical aspects
  - Efficient compilation, Operating system interaction, foreign function calls, ...
Demonstration: Basic Usage

EVERYTHING IS OKAY
By default, Idris uses a strict evaluation strategy. But what about...
By default, Idris uses a *strict* evaluation strategy. But what about... 

Control Structures

\[
\text{boolElim : Bool} \rightarrow \text{a} \rightarrow \text{a} \rightarrow \text{a} \\
\text{boolElim True } t \ e = t \\
\text{boolElim False } t \ e = e
\]

\[
\text{foo : t} \\
\text{foo = if expr then largeexpr1 else largeexpr2}
\]

Both largeexpr1 and largeexpr2 have to be evaluated in full!
By default, Idris uses a strict evaluation strategy. But what about... 

Control Structures with Laziness

```
boolElim : Bool -> |(t : a) -> |(e : a) -> a
boolElim True t e = t
boolElim False t e = e
```

```
foo : t
foo = if expr then largeexpr1 else largeexpr2
```

Now only one of largeexpr1 and largeexpr2 will be evaluated, depending which is needed.
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

### Some proofs

```kotlin
data Or a b = Inl a | Inr b
data And a b = And_intro a b

theorem1 : a -> Or a b
 theorem1 x = Inl x

theorem2 : a -> b -> And a b
 theorem2 x y = And_intro x y
```
Equality Proofs

Built-in types

\[
data (\equiv) : a \to b \to \text{Type} \quad \text{where}
\]
\[
\text{refl} : x \equiv x
\]

\[
data \_\_ \quad \text{where} \quad \{- \text{empty type} \} 
\]
Equality Proofs

Built-in types

data (=) : a -> b -> Type where
   refl : x = x

data _|_| where {- empty type -}

Rewriting

replace : {P : a -> Type} -> x = y -> P x -> P y
Equality Proofs

Example

twoPlusTwo : 2 + 2 = 4

twoPlusTwo = refl
Demonstration: Equality Proofs

(http://xkcd.com/285/)
Parity

data Parity : Nat -> Type where
  even : Parity (n + n)
  odd : Parity (S (n + n))
The with rule

Parity

data Parity : Nat -> Type where
  even : Parity (n + n)
  odd : Parity (S (n + n))

Every number has a parity

parity : (n : Nat) -> Parity n
The with rule

Parity

data Parity : Nat -> Type where
  even : Parity (n + n)
  odd : Parity (S (n + n))

Every number has a parity

parity : (n : Nat) -> Parity n

*Demonstration:* Implementing `parity`
Decidable equality

Type

data Dec : Type -> Type where
  Yes : a -> Dec a
  No : (a -> _|_ | _) -> Dec a

Checking Equality

class DecEq t where
  decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
Decidable equality

Type

```haskell
data Dec : Type -> Type where
  Yes : a -> Dec a
  No : (a -> _|_) -> Dec a
```

Checking Equality

```haskell
class DecEq t where
  decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
```

A `Bool` tells us the result of an equality test. `Dec` tells us why.
List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} ->
    Elem x xs -> Elem x (y :: xs)

Example
inList : Elem 2 [1..4]
inList = There Here
Membership predicates

List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} ->
    Elem x xs -> Elem x (y :: xs)

Example

inList : Elem 2 [1..4]
inList = There Here
Membership predicates

List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} ->
          Elem x xs -> Elem x (y :: xs)

Example

inList : Elem 2 [1..4]
inList = There Here

Demonstration: Building membership predicates