Idris: General Purpose Programming with Dependent Types

Edwin Brady
University of St Andrews

ecb10@st-andrews.ac.uk
@edwinbrady

27th June 2013
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 this talk:

- An introduction to the language
- Extended examples
Idris is a pure functional programming language with dependent types

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

In this talk:
- A tour of language features
- Extended examples
- Compilation challenges
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
Software Correctness
Idris Overview

- **General purpose** programming language
  - Compiled, supports foreign functions, . . .
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
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 | S Nat
```

Polymorphic lists

```
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

Vectors — polymorphic lists with length

```
data Vect : Type -> Nat -> Type where
  Nil : Vect a O
  (::) : a -> Vect a k -> Vect a (S k)
```
Append

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

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

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

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

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

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

### Pairwise addition

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

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

\[
vAdd \ (x :: xs) \ (y :: ys) = x + y :: vAdd \ xs \ ys
\]
Append

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

Pairwise addition

**total**

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

**Precise types**

```
sort : List Int -> List Int
```
Why Dependent Types?

Precise types

\[ \text{sort} : \text{Vect Int } n \rightarrow \text{Vect Int } n \]
**Precise types**

\[
\text{sort} : (\text{x}s : \text{Vect} \ \text{Int} \ n) \rightarrow \\
(\text{y}s : \text{Vect} \ \text{Int} \ n \ \ast \ast \ \text{Permutation} \ \text{x}s \ \text{y}s)
\]
Why Dependent Types?

Precise types

\[
\text{sort : } (xs : \text{Vect Int } n) \rightarrow \\
(y : \text{Vect Int } n \leftrightarrow \text{Permutation } xs \ y)
\]

We can make types as precise as we require.

However, precise types may require complex implementations/proofs.
Why Dependent Types?

In this lecture, we will see examples of:

- *Precise* types for supporting *machine-checked* proofs of correctness
In this lecture, we will see examples of:

- *Precise* types for supporting *machine-checked* proofs of correctness
- Generic programming
  - First class types, so types can be calculated by programs
Demonstration

EVERYTHING
IS OKAY
Idris is a work in progress. We’re having lots of fun:

- Implementing *domain specific languages*
- Investigating *scientific programming* applications
- Writing new code generators: Javascript, LLVM, . . .
- Investigating *network* and *security protocol* verification

But there’s lots to do, and you can get involved:

- Libraries (algorithms, data structures, proofs), generic programming, *deriving*, interactive development, automated theorem proving, more C bindings, . . .
- `cabal install idris` and enjoy...
An Effectful Problem (Haskell)

data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val x) = x
eval (Add x y) = eval x + eval y
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val x) = x
eval (Add x y) = eval x + eval y
data Expr = Val Int | Add Expr Expr | Var String
data Expr = Val Int | Add Expr Expr |
| Var String

type Env = [(String, Int)]
data Expr = Val Int | Add Expr Expr | Var String

type Env = [(String, Int)]

eval :: Expr -> ReaderT Env Maybe Int
data Expr = Val Int | Add Expr Expr
           | Var String

type Env = [(String, Int)]

eval :: Expr -> ReaderT Env Maybe Int
eval (Val n) = return n
eval (Add x y) = liftM2 (+) (eval x) (eval y)
eval (Var x) = do env <- ask
                  val <- lift (lookup x env)
                  return val
Evaluator with variables and state

data Expr = Val Int | Add Expr Expr
  | Var String
  | Random Int
Evaluator with variables and state

data Expr = Val Int | Add Expr Expr
  | Var String
  | Random Int

eval :: RandomGen g =>
  Expr -> RandT g (ReaderT Env Maybe) Int
Evaluator with variables and state

data Expr = Val Int | Add Expr Expr
           | Var String
           | Random Int

eval :: RandomGen g =>
       Expr -> RandT g (ReaderT Env Maybe) Int
...
eval (Var x) = do env <- lift ask
                 val <- lift (lift (lookup x env))
                   return val

eval (Random x) = do val <- getRandomR (0, x)
                       return val
Challenge — write the following:

\[
dropReader :: \text{RandomGen } g \Rightarrow \\
\text{RandT } g \text{ Maybe } a \rightarrow \\
\text{RandT } g \text{ (ReaderT Env Maybe) } a
\]

\[
commute :: \text{RandomGen } g \Rightarrow \\
\text{ReaderT} \text{ (RandT } g \text{ Maybe)} a \rightarrow \\
\text{RandT } g \text{ (ReaderT Env Maybe)} a
\]
Instead, we could capture everything in one evaluation monad:

```haskell
Eval monad
EvalState : Type
EvalState = (Int, List (String, Int))

data Eval a
  = MkEval (EvalState -> Maybe (a, EvalState))
```
Eval operations

\[
\text{rndInt} : \text{Int} \to \text{Int} \to \text{Eval} \text{ Int} \\
\text{get} : \text{Eval} \ \text{EvalState} \\
\text{put} : \text{EvalState} \to \text{Eval} \ ()
\]
An Effectful Problem (Idris)

Evaluator

\[
\begin{align*}
\text{eval} & : \text{Expr} \rightarrow \text{Eval} \text{ Int} \\
\text{eval} (\text{Val } i) & = \text{return } i \\
\text{eval} (\text{Var } x) & = \text{do} \ (\text{seed, env}) \leftarrow \text{get} \\
& \quad \text{lift} \ (\text{lookup } x \ \text{env}) \\
\text{eval} (\text{Add } x \ y) & = [\ | \ \text{eval } x + \ \text{eval } y \ |] \\
\text{eval} (\text{Random } \text{upper}) & = \text{do} \ \text{val} \leftarrow \text{rndInt } 0 \ \text{upper} \\
& \quad \text{return} \ \text{val}
\end{align*}
\]
Neither solution is satisfying!

- Composing monads with transformers becomes hard to manage
  - Order matters, but our effects are largely independent
- Building one special purpose monad limits reuse

Instead:

- We will build an *extensible* embedded domain specific language (EDSL) to capture *algebraic effects* and their *handlers*
The rest of this talk is about an EDSL, Effect. We will see:

- How to *use* effects
- How to *implement* new effects and handlers
- How Effect works
Effectful programs

\[
\text{EffM} : (m : \text{Type} \to \text{Type}) \to \\
\text{List EFFECT} \to \text{List EFFECT} \to \text{Type} \to \text{Type}
\]

\[
\text{Eff} : (\text{Type} \to \text{Type}) \to \text{List EFFECT} \to \text{Type} \to \text{Type}
\]

\[
\text{run} : \text{Applicative } m \Rightarrow \\
\text{Env m xs} \to \text{EffM m xs xs'} a \to m a
\]

\[
\text{runPure} : \text{Env id xs} \to \text{EffM id xs xs'} a \to a
\]
# Using Effects

## Some Effects

<table>
<thead>
<tr>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>Type $\rightarrow$ EFFECT</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>Type $\rightarrow$ EFFECT</td>
</tr>
<tr>
<td>STDIO</td>
<td>EFFECT</td>
</tr>
<tr>
<td>FILEIO</td>
<td>Type $\rightarrow$ EFFECT</td>
</tr>
<tr>
<td>RND</td>
<td>EFFECT</td>
</tr>
</tbody>
</table>
Using Effects

Some Effects

<table>
<thead>
<tr>
<th>Type</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>Type -&gt; EFFECT</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>Type -&gt; EFFECT</td>
</tr>
<tr>
<td>STDIO</td>
<td>EFFECT</td>
</tr>
<tr>
<td>FILEIO</td>
<td>Type -&gt; EFFECT</td>
</tr>
<tr>
<td>RND</td>
<td>EFFECT</td>
</tr>
</tbody>
</table>

Examples

get : Eff m [STATE x] x
putM : y -> EffM m [STATE x] [STATE y] ()
raise : a -> Eff m [EXCEPTION a] b
putStr : String -> Eff IO [STDIO] ()
Demonstration — effects
Recent comment on http://www.reddit.com/r/programming:

“There are also all kinds of issues with the complexity and performance of compiling languages that have dependent types.”
Recent comment on http://www.reddit.com/r/programming:

“There are also all kinds of issues with the complexity and performance of compiling languages that have dependent types.”

(http://www.xkcd.com/386 — Duty Calls)
Phase Distinctions

Conventionally seen as separation of *types* and *terms*
- Erase types before executing a program
- Perceived complexity with dependent types: what is a type and what is a term?
Phase Distinctions

- Conventionally seen as separation of *types* and *terms*
  - Erase types before executing a program
  - Perceived complexity with dependent types: what is a type and what is a term?
- Really separation of *compile time* and *run time*
  - Erase *compile time only* terms before executing a program
  - Conventionally compile time only = *types*
Phase Distinctions

- Conventionally seen as separation of *types* and *terms*
  - Erase types before executing a program
  - Perceived complexity with dependent types: what is a type and what is a term?
- Really separation of *compile time* and *run time*
  - Erase *compile time only* terms before executing a program
  - Conventionally compile time only = *types*
  - Distinction is harder to make with dependent types
    - but Idris does!
The compiler erases all values which it can prove are unused at run-time, by:

- **Forcing**
  - Erase constructor arguments with value determined by another
The compiler erases all values which it can prove are unused at run-time, by:

- **Forcing**
  - Erase constructor arguments with value determined by another
- **Collapsing**
  - Erase data type with only one inhabitant
  - Relies on *totality*
  - Typically erases equality proofs, predicates, . . .
The compiler erases all values which it can prove are unused at run-time, by:

- **Forcing**
  - Erase constructor arguments with value determined by another

- **Collapsing**
  - Erase data type with only one inhabitant
  - Relies on *totality*
  - Typically erases equality proofs, predicates, . . .

- **Identifying unused arguments**
  - Erase function and constructor arguments which are never inspected