Idris: Implementing a Dependently Typed Programming Language

Edwin Brady
University of St Andrews

ecb10@st-andrews.ac.uk
@edwinbrady

Type Inference and Automated Proving,
Dundee, 12th May 2015
Idris is a pure functional programming language with first class dependent types

http://idris-lang.org

In this talk:

- The core language (TT) and elaboration, or...
Idris

Idris is a pure functional programming language with first class dependent types

- http://idris-lang.org

In this talk:

- The core language (TT) and elaboration, or...
- ...how I tried to write a type checker, but accidentally wrote a theorem prover instead
Elaboration Example

Vectors, high level \texttt{Idris}

\begin{verbatim}
data \texttt{Vect : Nat \to Type \to Type} where
  \texttt{Nil : Vect Z a}
  \texttt{ (::: : a \to Vect k a \to Vect (S k) a}
\end{verbatim}

Vectors, TT

\begin{verbatim}
\texttt{Nil : (a : Type) \to Vect a Z}
\texttt{ (::: : (a : Type) \to (k : Nat) \to
  a \to Vect k a \to Vect (S k) a}
\end{verbatim}
### Elaboration Example

**Vectors, high level Idris**

```idris
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a
```

**Vectors, TT**

```idris
Nil : (a : Type) -> Vect a Z
(::) : (a : Type) -> (k : Nat) ->
    a -> Vect k a -> Vect (S k) a
```

**Example**

```idris
 (::) Char (S Z) 'a' ((::) Char Z 'b' (Nil Char))
-- ['a', 'b']
```
Pairwise addition, high level \textit{Idris}

\begin{verbatim}
\textbf{vAdd} : \textit{Num} \ a \to \textit{Vect} \ n \ a \to \textit{Vect} \ n \ a \to \textit{Vect} \ n \ a
\textbf{vAdd} \ \textbf{Nil} \ \textbf{Nil} \quad = \quad \textbf{Nil}
\textbf{vAdd} \ (x :: xs) \ (y :: ys) \quad = \quad x + y :: \textbf{vAdd} \ xs \ ys
\end{verbatim}
Elaboration Example

Step 1: Add implicit arguments

\[ \text{vAdd} : (a : _) \to (n : _) \to \]
\[ (\text{Num } a) \to \text{Vect } n \ a \to \text{Vect } n \ a \to \text{Vect } n \ a \]

\[ \text{vAdd } \_ \_ \ c \ (\text{Nil } \_ ) \ (\text{Nil } \_ ) = \text{Nil } \_ \]

\[ \text{vAdd } \_ \_ \ c \ (\_ \_ \ x \ \_ \ _ \ x s) \ (\_ \_ \ y \ y s) \]
\[ = (\_ \_ \ _ \ _ \ _ \_ \ _ \ (\_ \ _ \ _ \ x \ y) \ (\text{vAdd } \_ \_ \ _ \ _ \ x s \ y s) \) \]
Step 2: Solve implicit arguments

\[ v\text{Add} : (a : \text{Type}) \rightarrow (n : \text{Nat}) \rightarrow \\
(\text{Num } a) \rightarrow \text{Vect } n \ a \rightarrow \text{Vect } n \ a \rightarrow \text{Vect } n \ a \]

\[ v\text{Add} \ a \ Z \ c \ (\text{Nil } a) \ (\text{Nil } a) = \text{Nil } a \]

\[ v\text{Add} \ a \ (S \ k) \ c \ ((::) \ a \ k \ x \ xs) \ ((::) \ a \ k \ y \ ys) \\
= (::) \ a \ k \ ((+) \ c \ x \ y) \ (v\text{Add} \ a \ k \ c \ xs \ ys) \]
Step 3: Make pattern bindings explicit

\( \text{vAdd} : (a : \text{Type}) \rightarrow (n : \text{Nat}) \rightarrow \\
(\text{Num } a) \rightarrow \text{Vect } n \ a \rightarrow \text{Vect } n \ a \rightarrow \text{Vect } n \ a \)

\( \text{pat } a : \text{Type}, c : \text{Num } a . \\
\text{vAdd } a \ Z \ c \ (\text{Nil } a) \ (\text{Nil } a) = \text{Nil } a \)

\( \text{pat } a : \text{Type}, k : \text{Nat}, c : \text{Num } a . \\
\text{pat } x : a, xs : \text{Vect } a \ k, y : a, ys : \text{Vect } a \ k . \\
\text{vAdd } a \ (S \ k) \ c \ ( (::) \ a \ k \ x \ xs) \ ( (::) \ a \ k \ y \ ys) \\
= ( (::) \ a \ k \ (+(+) \ c \ x \ y) \ (\text{vAdd } a \ k \ c \ xs \ ys) ) \)
**Implementing Elaboration**

IDRIS programs may contain several high level constructs not present in TT:

- Implicit arguments, type classes
- where clauses, with and case structures, pattern matching let, ...
- Incomplete terms (metavariates)
- Types often left locally implicit

We want the high level language to be as expressive as possible, while remaining translatable to TT.
Consider Coq style theorem proving (with tactics) and Agda style (by pattern matching).

- **Pattern matching** is a convenient abstraction for humans to write programs
- **Tactics** are a convenient abstraction for building programs by refinement
  - i.e. explaining programming to a machine
Consider Coq style theorem proving (with tactics) and Agda style (by pattern matching).

- **Pattern matching** is a convenient abstraction for humans to write programs
- **Tactics** are a convenient abstraction for building programs by refinement
  - i.e. explaining programming to a machine

Idea: High level program structure directs tactics to build TT programs by refinement
Elaboration

Elaborating terms

build :: Pattern -> PTerm -> Elab Term
runElab :: Name -> Type -> Elab a -> Idris a
Elaboration is *type-directed*

- The **Idris** monad encapsulates system state
- The **Elab** monad encapsulates proof state
- The **Pattern** argument indicates whether this is the left hand side of a definition
- **PTerm** is the representation of the high-level syntax
- **Term** and **Type** are representations of **TT**
The proof state is encapsulated in a monad, \textit{Elab}, and contains:

- Current proof term (including \textit{holes})
  - Holes are incomplete parts of the proof term (i.e. sub-goals)
- Unsolved \textit{unification problems} (e.g. \texttt{f x = g y})
- Sub-goal in \textit{focus}
- Global context (definitions)
Some primitive operations:

- **Type checking**
  
  ```haskell
  check :: Raw -> Elab (Term, Type)
  ```

- **Normalisation**
  
  ```haskell
  normalise :: Term -> Elab Term
  ```

- **Unification**
  
  ```haskell
  unify :: Term -> Term -> Elab [(Name, Term)]
  ```
Implementing Elaboration — Operations

Some primitive operations:

- Type checking
  - \texttt{check :: Raw -> Elab (Term, Type)}

- Normalisation
  - \texttt{normalise :: Term -> Elab Term}

- Unification
  - \texttt{unify :: Term -> Term -> Elab [(Name, Term)]}

Querying proof state

- \texttt{goal :: Elab Type}
- \texttt{get_env :: Elab [(Name, Type)]}
- \texttt{get-proofTerm :: Elab Term}
A **tactic** is a function which updates a proof state, for example by:

- Updating the proof term
- Solving a sub-goal
- Changing focus

For example:

- `focus :: Name -> Elab ()`
- `claim :: Name -> Raw -> Elab ()`
- `forall :: Name -> Raw -> Elab ()`
- `exact :: Raw -> Elab ()`
- `apply :: Raw -> [Raw] -> Elab ()`
Tactics can be combined to make more complex tactics

- By sequencing, with do-notation
- By combinators:
  - `try :: Elab a -> Elab a -> Elab a`
    - If first tactic fails, use the second
  - `tryAll :: [Elab a] -> Elab a`
    - Try all tactics, exactly one must succeed
    - Used to disambiguate overloaded names

Effectively, we can use the `Elab` monad to write proof scripts (c.f. Coq’s `Ltac` language)
Elaborating Applications

**Append**

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

How do we build an application \texttt{Nil ++ (1 :: 2 :: Nil)}?

Elaborating each sub-term (and running \texttt{apply}) also runs the \texttt{unify} operation, which fills in the implicit arguments.
Append

\[(++) : \{a : \text{Type}\} \to \{n : \text{Nat}\} \to \{m : \text{Nat}\} \to \text{Vect \(n\) \(a\)} \to \text{Vect \(m\) \(a\)} \to \text{Vect \((n + m)\) \(a\)}\]

How do we build an application \(\text{Nil} ++ (1 :: 2 :: \text{Nil})\)?

Tactic script

do claim \(a\ \text{Type}\) ; claim \(n\ \text{Nat}\) ; claim \(m\ \text{Nat}\)
claim \(xs\ (\text{Vect \(n\) \(a\)})\) ; claim \(ys\ (\text{Vect \(m\) \(a\)})\)
Elaborating Applications

Append

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

How do we build an application \textbf{Nil ++ (1 :: 2 :: Nil)}?

Tactic script

do claim a \text{Type} ; claim n \text{Nat} ; claim m \text{Nat} \\
claim xs (\text{Vect } n \ a) ; claim ys (\text{Vect } m \ a) \\
apply ((++) a n m xs ys)
Elaborating Applications

Append

\[(++) : \{a : Type\} \to \{n : Nat\} \to \{m : Nat\} \to \text{Vect } n \ a \to \text{Vect } m \ a \to \text{Vect } (n + m) \ a\]

How do we build an application \text{Nil ++ (1 :: 2 :: Nil)}?

Tactic script

do claim a Type ; claim n Nat ; claim m Nat 
claim xs (Vect n a) ; claim ys (Vect m a) 
apply ((++) a n m xs ys) 
focus xs ; elab Nil 
focus ys ; elab (1 :: 2 :: Nil)
Elaborating Applications

Append

$$(++) : \{a : \text{Type}\} \to \{n : \text{Nat}\} \to \{m : \text{Nat}\} \to \text{Vect } n \ a \to \text{Vect } m \ a \to \text{Vect } (n + m) \ a$$

How do we build an application $\text{Nil} ++ (1 :: 2 :: \text{Nil})$?

Tactic script

```latex
\begin{verbatim}
do claim a Type ; claim n Nat ; claim m Nat
  claim xs (Vect n a) ; claim ys (Vect m a)
apply ((++) a n m xs ys)
focus xs; elab Nil
focus ys; elab (1 :: 2 :: Nil)
\end{verbatim}
```

Elaborating each sub-term (and running apply) also runs the unify operation, which fills in the implicit arguments.
Given an Idris application of a function $f$ to arguments $\texttt{args}$:

- Type check $f$
  - Yields types for each argument, $\texttt{ty}_i$
Given an Idris application of a function $f$ to arguments $\text{args}$:

- Type check $f$
  - Yields types for each argument, $\text{ty}_i$
- For each $\text{arg}_i : \text{ty}_i$, invent a name $n_i$ and run the tactic $\text{claim } n_i \text{ ty}_i$
Given an Idris application of a function $f$ to arguments $\text{args}$:

- Type check $f$
  - Yields types for each argument, $\text{ty}_i$
- For each $\text{arg}_i : \text{ty}_i$, invent a name $\text{n}_i$ and run the tactic $\text{claim n}_i \\text{ty}_i$
- Apply $f$ to $\text{ns}$
Given an Idris application of a function \( f \) to arguments \( \text{args} \):

- Type check \( f \)
  - Yields types for each argument, \( \text{ty}_i \)
- For each \( \text{arg}_i : \text{ty}_i \), invent a name \( \text{n}_i \) and run the tactic \text{claim} \( \text{n}_i \ \text{ty}_i \)
- Apply \( f \) to \( \text{ns} \)
- For each \textit{non-placeholder} \( \text{arg} \), \textbf{focus} on the corresponding \( \text{n} \) and elaborate \( \text{arg} \).
Given an Idris application of a function \( f \) to arguments \( \text{args} \):

- Type check \( f \)
  - Yields types for each argument, \( \text{ty}_i \)
- For each \( \text{arg}_i : \text{ty}_i \), invent a name \( n_i \) and run the tactic \( \text{claim } n_i \text{ ty}_i \)
- Apply \( f \) to \( \text{ns} \)
  - For each \text{non-placeholder} \( \text{arg} \), \text{focus} on the corresponding \( n \) and elaborate \( \text{arg} \).

(Note: elaborating an argument may affect the type of another argument!)
Elaborating Bindings

How do we build a function type, e.g. \((n : \text{Nat}) \rightarrow \text{Vect }n\text{ Int}\)?
Elaborating Bindings

How do we build a function type, e.g. \((n : \text{Nat}) \to \text{Vect } n \text{ Int}\)?

**Tactic script**

```plaintext
do claim \text{n\_S Type}
    forall \text{n n\_S}
focus \text{n\_S}; \text{elab Nat}
elab (\text{Vect } n \text{ Int})
```
Elaborating Bindings

In general, given a binder and its scope, say \((x : S) \rightarrow T\)

- Check that the current goal type is a `Type`
- Create a hole for `S`
  - `claim n_S Type`
- Create a binder with `forall x n_S`
- Elaborate `S` and `T`
### Top level declarations

<table>
<thead>
<tr>
<th>f</th>
<th>:</th>
<th>S1 -&gt; ... -&gt; Sn -&gt; T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x1 ... xn = e</td>
<td></td>
</tr>
</tbody>
</table>
Elaborating Declarations

Top level declarations

\[ f : S_1 \to \ldots \to S_n \to T \]
\[ f \ x_1 \ \ldots \ \ x_n = e \]

- Elaborate the type, and add \( f \) to the context
- Elaborate the lhs
  - Any out of scope names are assumed to be \textit{pattern} variables
- Elaborate the rhs \textit{in the scope of the pattern variables from the lhs}
- Check that the lhs and rhs have the same type
Elaborating where

Function with where block

\[ f : S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T \]
\[ f \ x_1 \ldots \ x_n = e \]
where
\[ f_{aux} = \ldots \]
### Function with `where` block

$$ f : S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T $$

$$ f \ x_1 \ \ldots \ \ x_n = e $$

**where**

$$ f_{\text{aux}} = \ldots $$

- Elaborate the lhs of $f$
- Lift the auxiliary definitions to top level functions *by adding the pattern variables from the lhs*
- Elaborate the auxiliary definitions
- Elaborate the rhs of $f$ as normal
Elaborating Type Classes

High level Idris

class Show a where
    show : a -> String

instance Show Nat where
    show Z = "Z"
    show (S k) = "s" ++ show k
Elaborating Type Classes

Elaborated TT

data Show : (a : Set) -> Set where
  ShowInstance : (show : a -> String) -> Show a

show : (Show a) -> a -> String
show (ShowInstance show’) x = show’ x

instanceShowNat : Show Nat
instanceShowNat = ShowInstance show where
  show : Nat -> String
  show Z = "Z"
  show (S k) = "s" ++ show k
Type class constraints are a special kind of implicit argument (c.f. Agda’s *instance arguments*)

- Ordinary implicit arguments solved by *unification*
- Constraint arguments solved by a tactic
  - `resolveTC :: Elab ()`
  - Looks for a local solution first
  - Then looks for globally defined instances
    - May give rise to further constraints
Idris is a high level language, elaborating to TT via:

- Tactics to build TT terms
- A top level monad for adding declarations

Everything is translated to a top-level declaration

- Add a high level feature (e.g. classes) by translating to declarations
- with and case constructs, records, type providers...

Adding new features has proved straightforward!

Full details: JFP 23(5): Idris, a general-purpose dependently typed programming language: Design and implementation
For more information

- http://idris-lang.org/documentation
- The mailing list idris-lang@groups.google.com
- The IRC channel, #idris, on irc.freenode.net