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

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

sicsa*

Elaboration Example

Vectors, high level IDRIS

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

Vectors, TT

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

Elaboration Example

Vectors, high level Idris

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

Vectors, TT

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

Example

```
(::) Char (S Z) 'a' ((::) Char Z 'b' (Nil Char))
      -- ['a', 'b']
```

sicsa*

Elaboration Example

Pairwise addition, high level IDRIS

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Elaboration Example

Step 1: Add implicit arguments

```
vAdd : (a : _) -> (n : _) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
       = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

Elaboration Example

Step 2: Solve implicit arguments

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
vAdd a Z c (Nil a) (Nil a) = Nil a
vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
       = (::) a k ((+) c x y) (vAdd a k c xs ys)
```

sicsa*

Elaboration Example

Step 3: Make pattern bindings explicit

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
pat a : Type, c : Num a .
  vAdd a Z c (Nil a) (Nil a) = Nil a
pat a : Type, k : Nat, c : Num a .
pat x : a, xs : Vect a k, y : a, ys : Vect a k .
  vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
       = (::) a k ((+) c x y) (vAdd a k c xs ys)
```

sicsa*

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 (metavariables)
- Types often left locally *implicit*

We want the high level language to be as *expressive* as possible, while remaining translatable to TT.

An observation

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

An observation

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

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

Implementing Elaboration — Proof State

The proof state is encapsulated in a monad, Elab, and contains:

- Current proof term (including *holes*)
  - Holes are incomplete parts of the proof term (i.e. sub-goals)
- Unsolved *unification problems* (e.g. `f x = g y`)
- Sub-goal in *focus*
- Global context (definitions)

sicsa*

Implementing Elaboration — Operations

Some primitive operations:

- Type checking
  - `check :: Raw -> Elab (Term, Type)`
- Normalisation
  - `normalise :: Term -> Elab Term`
- Unification
  - `unify :: Term -> Term -> Elab [(Name, Term)]`

Implementing Elaboration — Operations

Some primitive operations:

- Type checking
  - check :: Raw -> Elab (Term, Type)
- Normalisation
  - normalise :: Term -> Elab Term
- Unification
  - unify :: Term -> Term -> Elab [(Name, Term)]

Querying proof state

- goal :: Elab Type
- get_env :: Elab [(Name, Type)]
- 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 ()`

Implementing Elaboration — Tactics

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} -> {n : Nat} -> {m : Nat} ->
       Vect n a -> Vect m a -> Vect (n + m) a
```

How do we build an application `Nil ++ (1 :: 2 :: Nil)`?

Elaborating Applications

Append

```
(++) : {a : Type} -> {n : Nat} -> {m : Nat} ->
       Vect n a -> Vect m a -> Vect (n + m) a
```

How do we build an application `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)
```

Elaborating Applications

Append

```
(++) : {a : Type} -> {n : Nat} -> {m : Nat} ->
       Vect n a -> Vect m a -> Vect (n + m) a
```

How do we build an application `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)
```

Elaborating Applications

Append

```
(++) : {a : Type} -> {n : Nat} -> {m : Nat} ->
       Vect n a -> Vect m a -> Vect (n + m) a
```

How do we build an application `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 : Type} -> {n : Nat} -> {m : Nat} ->
       Vect n a -> Vect m a -> Vect (n + m) a
```

How do we build an application `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 each sub-term (and running **apply**) also runs the **unify** operation, which fills in the implicit arguments.

Elaborating Applications

Given an IDRIS application of a function `f` to arguments `args`:

- Type check `f`
  - Yields types for each argument, `ty_i`

Elaborating Applications

Given an IDRIS application of a function `f` to arguments `args`:

- Type check `f`
  - Yields types for each argument, `ty_i`
- For each `arg_i : ty_i`, invent a name `n_i` and run the tactic **claim** `n_i ty_i`

Given an IDRIS application of a function `f` to arguments `args`:

- Type check `f`
  - Yields types for each argument, `ty_i`
- For each `arg_i : ty_i`, invent a name `n_i` and run the tactic claim `n_i ty_i`
- Apply `f` to `ns`

Elaborating Applications

Given an IDRIS application of a function `f` to arguments `args`:

- Type check `f`
  - Yields types for each argument, `ty_i`
- For each `arg_i : ty_i`, invent a name `n_i` and run the tactic **claim** `n_i ty_i`
- Apply `f` to `ns`
- For each *non-placeholder* `arg`, **focus** on the corresponding `n` and elaborate `arg`.

sicsa*

Elaborating Applications

Given an Idris application of a function `f` to arguments `args`:

- Type check `f`
  - Yields types for each argument, `ty_i`
- For each `arg_i : ty_i`, invent a name `n_i` and run the tactic **claim** `n_i ty_i`
- Apply `f` to `ns`
- For each *non-placeholder* `arg`, **focus** on the corresponding `n` and elaborate `arg`.

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

How do we build a function type,
e.g. `(n : Nat) -> Vect n Int`?

Elaborating Bindings

How do we build a function type,
e.g. `(n : Nat) -> Vect n Int`?

Tactic script

```
do claim n_S Type
   forall n n_S
   focus n_S; elab Nat
   elab (Vect n Int)
```

Elaborating Bindings

In general, given a binder and its scope, say `(x : S) -> 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`

Elaborating Declarations

Top level declarations

```
f : S1 -> ... -> Sn -> T
f x1 ... xn = e
```

Elaborating Declarations

Top level declarations

```
f : S1 -> ... -> Sn -> T
f x1 ... xn = e
```

- Elaborate the type, and add `f` to the context
- Elaborate the lhs
    - Any out of scope names are assumed to be *pattern* variables
- Elaborate the rhs *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 : S1 -> ... -> Sn -> T
f x1 ... xn = e
  where
    f_aux = ...
```

Elaborating where

Function with where block

```
f : S1 -> ... -> Sn -> T
f x1 ... xn = e
  where
    f_aux = ...
```

- 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
```

sicsa*

Elaborating Type Classes

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

Summary

- 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