

Type checking by theorem proving in IDRIS

Scottish Theorem Proving, 10th February 2012

ecb10@st-andrews.ac.uk

University of St Andrews

Edwin Brady



Introduction

This talk is about IDRIS, a programming language with *dependent types*.

- `cabal update; cabal install idris`
- <http://idris-lang.org/>
- <http://idris-lang.org/documentation/>

In particular:

- How I tried to write a type checker, and accidentally wrote a theorem prover instead

The IDRIS Programming Language

IDRIS is a general purpose pure functional programming language, with support for theorem proving. Features include:

- Full *dependent types*, dependent pattern matching
- Dependent *records*
- *Type classes* (Haskell style)
 - Numeric overloading, Monads, `do`-notation, idiom brackets, . . .
- *Tactic* based theorem proving
- High level constructs: `where`, `case`, `with`, monad comprehensions, syntax overloading
- *Totality* checking, cumulative universes
- Interfaces for *systems* programming (e.g. C libraries)

Dependent Types in IDRIS

IDRIS syntax is influenced by Haskell. Some data types:

```
data Nat = 0 | S Nat
```

```
infixr 5 :: -- Define an infix operator
```

```
data Vect : Set -> Nat -> Set where -- List with size
```

```
Nil : Vect a 0
```

```
(::) : a -> Vect a k -> Vect a (1 + k)
```

Dependent Types in IDRIS

IDRIS syntax is influenced by Haskell. Some data types:

```
data Nat = 0 | S Nat
```

```
infixr 5 :: -- Define an infix operator
```

```
data Vect : Set -> Nat -> Set where -- List with size
```

```
  Nil : Vect a 0
```

```
  (::) : a -> Vect a k -> Vect a (1 + k)
```

For example:

```
S (S (S 0))           : Nat
```

```
Nil                  : Vect a 0
```

```
2 :: 3 :: 5 :: Nil : Vect Int 3 -- overloaded literal
```

Dependent Types in IDRIS

IDRIS syntax is influenced by Haskell. Some data types:

```
data Nat = 0 | S Nat
```

```
infixr 5 :: -- Define an infix operator
```

```
data Vect : Set -> Nat -> Set where -- List with size
```

```
  Nil : Vect a 0
```

```
  (::) : a -> Vect a k -> Vect a (1 + k)
```

The type of a function over vectors describes invariants of the input/output lengths, e.g..

```
append : Vect a n -> Vect a m -> Vect a (n + m)
```

```
append Nil      ys = ys
```

```
append (x :: xs) ys = x :: append xs ys
```

Dependent Types in IDRIS

We can use Haskell style type classes to constrain polymorphic functions, e.g., pairwise addition a vectors of numbers:

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

(Aside: The `total` keyword means that it is an error if the totality checker cannot determine that the function is total)

The Core Language, TT

High level IDRIS programs *elaborate* to a core language, TT:

- TT allows *only* data declarations and top level pattern matching definitions
- Limited syntax:
 - Variables, application, binders (λ , \forall , let, patterns), constants
- All terms *fully explicit*
- Advantage: type checker is small (\approx 500 lines) so less chance of errors
- Challenge: how to build TT programs from IDRIS programs?

Elaboration example

Consider again pairwise addition:

```
vAdd : Num a => Vect a n -> Vect a n -> Vect a n
```

```
vAdd Nil Nil = Nil
```

```
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Elaboration example

Consider again pairwise addition:

```
vAdd : Num a => Vect a n -> Vect a n -> Vect a n
```

```
vAdd Nil Nil = Nil
```

```
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Step 1: Add implicit arguments

```
vAdd : (a : _) -> (n : _) ->
```

```
Num a -> Vect a n -> Vect a n -> Vect a n
```

```
vAdd _ _ c (Nil _) (Nil _) = (Nil _)
```

```
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
```

```
= (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

Elaboration example

Consider again pairwise addition:

```
vAdd : Num a => Vect a n -> Vect a n -> Vect a n
```

```
vAdd Nil Nil = Nil
```

```
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Step 2: Solve implicit arguments

```
vAdd : (a : Set) -> (n : Nat) ->
```

```
Num a -> Vect a n -> Vect a n -> Vect a n
```

```
vAdd a 0 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)
```

Elaboration example

Consider again pairwise addition:

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

Step 3: Make pattern bindings explicit

```
vAdd : (a : Set) -> (n : Nat) ->
      Num a -> Vect a n -> Vect a n -> Vect a n
pat a : Set, c : Num a .
  vAdd a 0 c (Nil a) (Nil a) = (Nil a)
pat v : Set, 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)
```

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`, ...
- 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/Lego 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

- The Elaborator is implemented as an *Embedded Domain Specific Language* in Haskell

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)
- Sub-goal in *focus*
- Global context (definitions)

We distinguish terms which have been typechecked from those which have not:

- `RawTerm` has not been type checked (and may contain placeholders, `_`)
- `Term` has been type checked (`Type` is a synonym)

Implementing Elaboration — Operations

Some primitive operations:

- Type checking
 - `check :: RawTerm -> Elab (Term, Type)`
- Normalisation
 - `normalise :: Term -> Elab Term`
- Unification
 - `unify :: Term -> Term -> Elab ()`

Querying proof state

- Get the local environment
 - `get_env :: Elab [(Name, Type)]`
- Get the current proof term
 - `get_proofTerm :: Elab Term`

Implementing Elaboration — Tactics

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 -> RawTerm -> Elab ()`
- `forall :: Name -> RawTerm -> Elab ()`
- `exact :: RawTerm -> Elab ()`
- `apply :: RawTerm -> [RawTerm] -> 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

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 .

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

Elaborating Applications

For example, recall `append`

```
append : (a : Set) -> (n : Nat) -> (m : Nat) ->
         Vect a n -> Vect a m -> Vect a (n + m)
```

To build an application `append _ _ _ Nil (1 :: 2 :: Nil)`

```
do claim a Set ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
   apply (append 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 `_`

Elaborating Bindings

Given a binder and its scope, say $(x : S) \rightarrow T$

- Check that the current goal type is a `Set`
- Create a hole for `S`
 - `claim n_S Set`
- Create a binder with `forall x n_S`
- Elaborate `S` and `T`

Elaborating Bindings

For example, to build `(n : Nat) -> Vect Int n`

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

Running the Elaborator

```
runElab :: Name -> Type -> Elab a -> Idris a
build   :: Pattern -> PTerm -> Elab Term
```

- Elaboration is *type-directed*
- The `Idris` monad encapsulates system state
- The `Pattern` argument indicates whether this is the left hand side of a definition
 - Free variables on the lhs in IDRIS become pattern bindings in TT
 - `patbind :: Name -> Elab ()` — convert current goal to pattern binding
- `PTerm` is the representation of the high-level syntax

Elaborating Declarations

Given a function declaration of the following form:

```
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

Given a function declaration with auxiliary local definitions:

```
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

Adding Type Classes

IDRIS has type classes, for example:

```
class Show a where
```

```
  show : a -> String
```

```
instance Show Nat where
```

```
  show 0 = "0"
```

```
  show (S k) = "S" ++ show k
```

Type classes translate directly into data types; instances translate directly into functions.

Adding Type Classes

This type class translates to:

```
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 0 = "0"
  show (S k) = "s" ++ show k
```

Adding 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

Conclusions

Given basic components (type checking, unification, evaluation) we have built an *elaborator* for a dependently typed language.

- Building a DSL for elaboration first: flexible, expressive
 - Access to environment, proof-term, ability to create sub-goals, . . .
- High level features have proved easy to add
 - Type classes, records, overloading, tactic scripts, . . .
- Efficient enough (meaning: outperforms previous version!)
 - Prelude: \approx 3000 lines in 6.5 secs
 - Profiling suggests some easy efficiency gains to be had
- Don't forget to `cabal install idris :-)`