Dependently Typed Functional Programming with Idris

Lecture 4: Implementing Idris

Edwin Brady
University of St Andrews

ecb10@st-andrews.ac.uk
@edwinbrady

In This Lecture

IDRIS internals:

- The core language, TT
- *Elaboration* from high level programs to TT
- Compilation challenges

Code is available to browse at
`https://github.com/edwinb/Idris-dev`

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

Vectors, high level IDRIS

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

Elaboration Example

Vectors, high level IDRIS

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

Vectors, TT

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

Elaboration Example

Vectors, high level Idris

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

Vectors, TT

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

Example

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

Elaboration Example

Pairwise addition, high level Idris

```
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

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

Step 2: Solve implicit arguments

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect a n -> Vect a n -> Vect a n
vAdd a O 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

Step 3: Make pattern bindings explicit

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect a n -> Vect a n -> Vect a n
pat a : Type, c : Num a .
  vAdd a O 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)
```

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 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

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

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

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

- `Raw` 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 :: Raw -> Elab (Term, Type)`

Implementing Elaboration — Operations

Some primitive operations:

- Type checking
  - `check :: Raw -> Elab (Term, Type)`
- Normalisation
  - `normalise :: Term -> Elab Term`

Implementing Elaboration — Operations

Some primitive operations:

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

Implementing Elaboration — Operations

Some primitive operations:

- Type checking
  - `check :: Raw -> 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 -> 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

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`

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`

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`.

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

Append

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

Elaborating Applications

Append

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

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
```

Elaborating Applications

Append

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

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
```

Elaborating Applications

Append

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

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
   apply ((++) a n m xs ys)
```

Elaborating Applications

Append

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

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
   apply ((++) a n m xs ys)
   focus xs; elab Nil
```

Elaborating Applications

Append

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

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
   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 a n -> Vect a m -> Vect a (n + m)
```

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

Tactic script

```
do claim a Type ; claim n Nat ; claim m Nat
   claim xs (Vect a n) ; claim ys (Vect a m)
   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 _

Elaborating Bindings

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 Bindings

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

Tactic script

```
do claim n_S Type
```

Elaborating Bindings

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

Tactic script

```
do claim n_S Type
   forall n n_S
```

Elaborating Bindings

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

Tactic script

```
do claim n_S Type
   forall n n_S
   focus n_S; elab Nat
```

Elaborating Bindings

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

Tactic script

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

Running the Elaborator

Elaborating terms

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

Running the Elaborator

Elaborating terms

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

- Elaboration is *type-directed*

Running the Elaborator

Elaborating terms

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

- Elaboration is *type-directed*
- The `Idris` monad encapsulates system state

Running the Elaborator

Elaborating terms

```
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

Running the Elaborator

Elaborating terms

```
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

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

High level IDRIS

```
class Show a where
    show : a -> String

instance Show Nat where
    show O = "O"
    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 O = "O"
    show (S k) = "s" ++ show k
```

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

Compilation

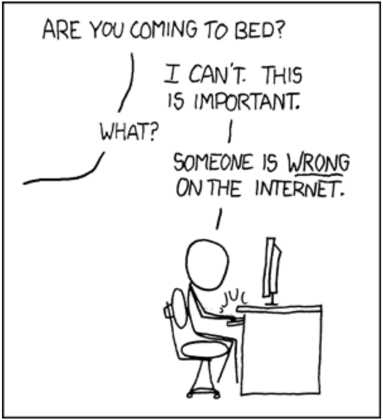
Recent comment on `http://www.reddit.com/r/programming`:

> *"There are also all kinds of issues with the complexity and performance of compiling languages that have dependent types."*

Compilation

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!

Phase Distinctions in IDRIS

The compiler erases all values which it can prove are unused at run-time, by:

- *Forcing*
    - Erase constructor arguments with value determined by an index

Phase Distinctions in IDRIS

The compiler erases all values which it can prove are unused at run-time, by:

- *Forcing*
  - Erase constructor arguments with value determined by an index
- *Collapsing*
  - Erase data type with only one inhabitant
  - Relies on *totality*
  - Typically erases equality proofs, predicates, ...

Phase Distinctions in Idris

The compiler erases all values which it can prove are unused at run-time, by:

- *Forcing*
  - Erase constructor arguments with value determined by an index
- *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

Forcing Example

Vectors, TT

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

vAdd

```
vAdd : (a : Type) -> (n : Nat) ->
          (Num a) -> Vect a n -> Vect a n -> Vect a n
vAdd a O 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)
```

Forcing Example

Vectors, TT

```
Nil  :                 Vect a O
(::) :
       a -> Vect a k -> Vect a (S k)
```

vAdd

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

Forcing Example

Vectors, TT

```
Nil  :                    Vect a O
(::) :
         a -> Vect a k -> Vect a (S k)
```

vAdd

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

Collapsing Example

Less than predicate, high level IDRIS

```
data LE : Nat -> Nat -> Type where
     ltO : LE O m
     ltS : LE n m -> LE (S n) (S m)
```

sicsa*

Collapsing Example

Less than predicate, TT

```
LE  : Nat -> Nat -> Type
ltO : (m : Nat) -> LE O m
ltS : (n : Nat) -> (m : Nat) ->
      LE n m -> LE (S n) (S m)
```

minus

```
minus : (x : Nat) -> (y : Nat) -> LE y x -> Nat
minus x     O     (ltO x)     = x
minus (S x) (S y) (ltS x y p) = minus x y p
```

Collapsing Example

Less than predicate, TT

```
LE   : Nat -> Nat -> Type
ltO  :              LE O m
ltS  :
      LE n m -> LE (S n) (S m)
```

minus

```
minus : (x : Nat) -> (y : Nat) -> LE y x -> Nat
minus x     O     (ltO   )     = x
minus (S x) (S y) (ltS     p) = minus x y p
```

Collapsing Example

Less than predicate, TT

```
LE    : Nat -> Nat -> Type
[lt0] :              LE 0 m
[ltS] :
        LE n m -> LE (S n) (S m)
```

minus

```
minus : (x : Nat) -> (y : Nat) -> LE y x -> Nat
minus x     0     (     )     = x
minus (S x) (S y) (        p) = minus x y p
```

sicsa*

Collapsing Example

Less than predicate, TT

```
LE    : Nat -> Nat -> Type
[lt0] :             LE 0 m
[ltS] :
          -> LE (S n) (S m)
```

minus

```
minus : (x : Nat) -> (y : Nat) -> LE y x -> Nat
minus x     0     _               = x
minus (S x) (S y) _               = minus x y _
```

Logging

Some internal (undocumented) features for examining structure:

- REPL command `:di`
  - For “debug info”
- `%logging` directive
  - Displays elaborator progress
  - e.g. `%logging 5` for a high level of logging info
- `--dumpcases <filename>` compiler flag
  - Outputs compiled case trees

Logging

Some internal (undocumented) features for examining structure:

- REPL command `:di`
  - For "debug info"
- `%logging` directive
  - Displays elaborator progress
  - e.g. `%logging 5` for a high level of logging info
- `--dumpcases <filename>` compiler flag
  - Outputs compiled case trees

Additional exercise: use these features to examine the internal representations of your answers to earlier exercises

The End

On this course, we have covered:

- Introductory programming with dependent types
  - Invariants, predicates, theorem proving
- Embedded Domain Specific Languages
  - Implementing the λ-calculus
  - Verification of extra-functional properties
- Effect management
  - Programming with side effects, implementing new effects
- Implementing a dependently typed language
  - Elaboration overview
  - Type erasure

Future Work

Useful features to be added to IDRIS:

- `deriving` for type classes
- Run-time representation of data
- Decision procedures
  - e.g. Presburger arithmetic solver
- Programming tools
  - Type directed editing
  - Adapt hoogle for IDRIS
- See `http://idris-lang.org/help-required`

For more information

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