Programming and refactoring dependent types

Andreas Reuleaux

Programming Languages and Systems group (PLAS), University of Kent

BCS · December 2016

outline

- programming dependent types (examples from IDRIS) — 2/3 time
- refactoring dependent types (PI-FORALL) — 1/3 time

dependent types: definition and first example

Definition (dependent types)

Dependent types are types that may depend on values.

Example

vectors (as opposed to lists) have a notion size: `Vect (n: Nat) a`

```
Idris> :module  Data.Vect

*Data/Vect> the (List _) [2, 17, 5, 9]
[2, 17, 5, 9] : List Integer

*Data/Vect> the (Vect _ _) [2, 17, 5, 9]
[2, 17, 5, 9] : Vect 4 Integer

*Data/Vect> the (Vect _ _) ["hello", "world"]
["hello", "world"] : Vect 2 String
```

dependent types: some background

- The *λ-calculus* as a mathematical model for the "mechanics of computation" emerged in the 1930's (Alonzo Church)
- *Type systems* were introduced to distinguish certain classes of valid programs, and predict their behaviour.
- *Curry/Howard* discovered the correspondence between programs and proofs:

  $$x : \quad T \quad \begin{cases} x \textit{ is of type } T \\ x \textit{ is a proof of } T \end{cases}$$

  at the cost of *constructive logic* and *terminating programs*
- in *constructive logic* one has to provide witnesses as proofs, as opposed to classical logic whith the law of the excluded middle.
- all programs must terminate, otherwise inconsistencies in the logic are possible. (in IDRIS: *total functions*).

corresponding according to Curry/Howard

The Curry-Howard isomorphism

Conjunction	$\wedge$	Product type
Implication	$\rightarrow$	Function type
Disjunction	$\vee$	Disjoint union type
Trivial	$\top$	One element type
Absurd	$\perp$	Empty type
Universally quantified	$\forall$	Dependent function space
Existentially quantified	$\exists$	Dependent product
Natural numbers / induction	$\mathbb{N}$	Natural numbers / recursion
Inductive predicates	$\mathfrak{T}$	Inductively defined types
Co-inductive predicates	$\mathfrak{C}$	Co-inductively defined types
Identity predicates	$\mathfrak{I}$	Identity types

vectors, natural numbers, and bounded natural numbers, inductively defined

vectors, natural numbers, and bounded natural numbers, inductively defined

vectors (1)

```
*Data/Vect> :doc Vect
Data type Data.Vect.Vect : Nat -> Type -> Type
    Vectors: Generic lists with explicit length in the type

Constructors:
    Nil : Vect 0 a
        Empty vector

    (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
        A non-empty vector of length S k, consisting of a head element
        and the rest of the list, of length k.
        infixr 7
```

vectors (2)

lists and vectors defined ourselves (in a file `MyStuff.idr`):

```
data MyList : (elem : Type) -> Type where
  LNil  : MyList elem
  LCons : (x : elem) -> (xs : MyList elem) -> MyList elem

data MyVect : (len : Nat) -> (elem : Type) -> Type where
  VNil  : MyVect Z elem
  VCons : (x : elem) -> (xs : MyVect len elem) -> MyVect (S len) elem
```

- ask the type of a function with `:t`
- ask for general documention with `:doc`
- browse the IDRIS documentation (for vectors):
  http://www.idris-lang.org/docs/current/base_doc/docs/Data.Vect.html
- have a look at the IDRIS source code (for vectors):
  https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Vect.idr

vectors (3)

usage:

```
Idris> :l MyStuff.idr
*MyStuff> :module Data.Vect

*MyStuff *Data/Vect> the (Vect _ _) [5, 7]
[5, 7] : Vect 2 Integer

*MyStuff *Data/Vect> the (Vect _ _) (5 :: (7 :: Nil))
[5, 7] : Vect 2 Integer

*MyStuff *Data/Vect> 5 `VCons` (7 `VCons` VNil)
VCons 5 (VCons 7 VNil) : MyVect 2 Integer
```

vectors, natural numbers, and bounded natural numbers, inductively defined

natural numbers

```
*Data/Vect> :doc Nat
Data type Prelude.Nat.Nat : Type
    Natural numbers: unbounded, unsigned integers which can be
    pattern matched.

Constructors:
    Z : Nat
        Zero

    S : Nat -> Nat
        Successor
```

bounded natural numbers (1)

bounded natural (or finite) numbers: `Fin (n: Nat)` $= \mathbb{N}_{<n}$
usage:

```
Idris> :module  Data.Fin

*Data/Fin> the (Fin 5) 3
FS (FS (FS FZ)) : Fin 5

*Data/Fin> the (Fin 5) 7
(input):1:14:When checking argument prf to function Data.Fin.fromInteger:
        When using 7 as a literal for a Fin 5
                7 is not strictly less than 5
```

bounded natural numbers (2)

```
*Data/Vect> :doc Fin
Data type Data.Fin.Fin : (n : Nat) -> Type
    Numbers strictly less than some bound. The name comes from
    "finite sets".

    It's probably not a good idea to use Fin for arithmetic, and
    they will be exceedingly inefficient at run time.
    Arguments:
        n : Nat  -- the upper bound

Constructors:
    FZ : Fin (S k)


    FS : Fin k -> Fin (S k)
```

safe head function (1)

`head` on IDRIS vectors:

```
*MyStuff *Data/Vect> :t Vect.head
head : Vect (S n) a -> a

*MyStuff *Data/Vect> Vect.head $ [5, 7]
5 : Integer

*MyStuff *Data/Vect> Vect.head $ []
(input):1:11-12:When checking an application of function Data.Vect.head:
        Type mismatch between
                Vect 0 a (Type of [])
        and
                Vect (S n) iType (Expected type)

        Specifically:
                Type mismatch between
                        0
                and
                        S n
```

safe head function (2)

`head` on HASKELL lists:

```
Prelude > :t head
head :: [a] -> a

Prelude > head [5, 7]
5

Prelude > head []
*** Exception: Prelude.head: empty list
```

`head` on IDRIS lists:

```
Idris> :t List.head
head : (l : List a) -> {auto ok : NonEmpty l} -> a
```

bounds safe lookup (or `index`) function

```
*Data/Vect> :t Vect.index
index : Fin len -> Vect len elem -> elem

*Data/Vect> Vect.index 2 [1, 4, 23, 17]
23 : Integer

*Data/Vect> Vect.index 6 [1, 4, 23, 17]
(input):1:14:When checking argument prf to function Data.Fin.fromInteger:
        When using 6 as a literal for a Fin 4
                6 is not strictly less than 4
```

common list and vector functions

HASKELL list function	IDRIS vector function (or variation thereof)
head	
`head :: [a] -> a`	`head : Vect (S len) elem -> elem`
lookup or index	
`(!!) :: [a] -> Int -> a`	`index : Fin len -> Vect len elem -> elem`
append	
`(++) :: [a] -> [a] -> [a]`	`(++) : Vect m elem -> Vect n elem -> Vect (m + n) elem`
take	
`take :: Int -> [a] -> [a]`	`take : (n : Nat) -> Vect (n + m) elem -> Vect n elem`
drop	
`drop :: Int -> [a] -> [a]`	`drop : (n : Nat) -> Vect (n + m) elem -> Vect m elem`
map (possible definition)	
`map :: (a -> b) -> [a] -> [b]`	`map : (a -> b) -> Vect n a -> Vect n b`
filter (possible definition)	
`filter :: (a -> Bool) -> [a] -> [a]`	`filter : (a -> bool) -> Vect n a -> Vect (<=n) a`
and more	

typical patterns of vector sizes

pattern	used as
`S m`	one greater than a given size `m` or non-zero, ie. at least one
`m+n`	the sum of two given vector sizes `m` and `n`
`<=m`	less or equal than a given size `m`
etc.	

matrix product

$$(n \times m) \times (m \times p) = (n \times p)$$

$$\begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \times \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\begin{matrix} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \\ \begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{matrix}$$

```
matrixProduct : Num num =>
  Vect n (Vect m num) -> Vect m (Vect p num) ->  Vect n (Vect p num)
```

sorting

sorting lists (vectors), sorting broken down:

- the sorted list (vector) has the same length as the original one
- permutation of the original elements
- any pair of elements is sorted.

dependent pairs

example: reading a vector yields a dependent pair:

```
readVect : IO (len ** Vect len String)
```

as we don't know the size of the vector before hand.

strong types

- Yes, because.../ No, because...
  as opposed to Boolean True/False
- etc.

proofs

append is easily proved

```
(++) : (xs : Vect n elem) -> (ys : Vect m elem) -> Vect (plus n m) elem
(++) []       ys = ys
(++) (x::xs) ys = x :: (xs ++ ys)
```

given `plus`:

```
plus: Nat -> Nat -> Nat
plus Z m = m
plus (S n) m = S (plus n m)
```

but much harder to prove

```
(+++) : Vect n a -> Vect m a -> Vect (plus' n  m) a
```

given a slightly different `plus'` function:

```
plus' n Z = n
plus' n (S m) = S (plus' n m)
```

example from David Christiansen's Master thesis, chapter 4
(the wording is mine, recent IDRIS vector syntax)

dependently typed languages: inner workings (1)

some key ideas:

- expressions and types live in the same realm,
  ie. one common data structure for *expressions* and *types*
  (or otherwise mutually recursive ones):

```
data Expr =
  Type
  | Var...
  | Lam...
  | App...
  | ...
```

dependently typed languages: inner workings (2)

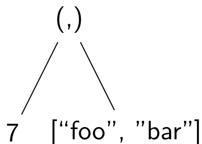
cf. this to classical functional languages, where they are seperate:

```
data Expr =
  Var...
  | Lam...
  | App...
  | ...
```

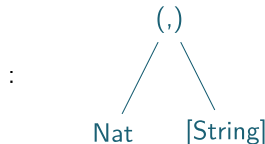
```
data Type =
   | Nat
   | String
   | [Type]
   | (Type, Type)
   | ...
```

types reflecting the structure of expressions, eg.:

- 7 : Nat
- "hello" : String
- (7, ["foo", "bar"]) : (Nat, [String])



:



dependently typed languages: inner workings (3)

key ideas, continued:

- the function type constructor $\rightarrow$ is a binder: $(x : A) \rightarrow B(x)$
  similar to a λ-abstraction: $\lambda x.E(x)$,
  in simple cases we may omit the x, thus just: $A \rightarrow B$
- typing rules come in two flavours each: checking rules, and inference rules
  cf. Stephanie Weirich's Oregon summerschool talks 2014 and 2015

Refactoring PI-FORALL, outline

- refactoring in general
- parsing concrete syntax, in a white space aware manner ie., given the chosen libraries: *Trifecta* and *Bound*, as opposed to *Parsec* and *Unbound*
- navigation in the zipper representation of the syntax tree
- finding syntax elements, given position information only
- simple classical refactorings (in a dependently typed context): renaming, generalisation
- more ambitious efforts, taking dependent types into account: list-to-vector refactorings
- data-type refactorings in general, eg. list-to-tree

Refactoring

- program transformations to improve the design of existing programs
- typically applied in a step by step manner, each step preserving the original meaning of the program
- well established among software practitioners since the early nineties (William Opdyke, Martin Fowler, ..., for OO languages at the time)

Refactoring

Rough analogies

- Whereas as a compiler translates from one language to another *(think eg. of translating from English to French)*,
- refactoring is rewriting some program in its original language *(think eg. of an editor of a newspaper rewriting/improving some text)*

some experience in our group at Kent:

- HaRe for Haskell
- Wrangler for Erlang

building upon that experience: Refactoring dependently typed programs:

- Pi-Forall for now, keeping Idris in mind

Refactoring

Refactorings broken down

- parsing
  - *syntax - structure*
  - carries its *semantics*
  - in refactoring also pragmatics: preserving things beyond that: white-space including comments, desugaring made explicit etc.
- program transformations - *substitutions* (using all the knowledge available)
- ensure that the meaning hasn't changed: type checking, tests.

parsing *if-then-else* (1)

parsing

```
if a then b else c
```

or more generally

```
if{-i-}a{-a'-}then{-t-}b{-b'-}else{-e-}c{-c-}
```

yields

```
If (V "a") (V "b") (V "c") (Annot Nothing)
```

respectively

```
If_ (IfTok "if" (Ws "{-i-}")) (Ws_ (V "a") (Ws "{-a'-}"))
    (ThenTok "then" (Ws "{-t-}")) (Ws_ (V "b") (Ws "{-b'-}"))
    (ElseTok "else" (Ws "{-e-}")) (Ws_ (V "c") (Ws "{-c-}"))
    (Annot_ Nothing (Ws ""))
```

parsing *if-then-else* (2)

syntax definitions:

```
data Expr t a =
  V a
  | Ws_ (Expr t a) (Ws t)
  ...
  | Lam  t       (Scope () (Expr t) a)
  ...
  | If (Expr t a) (Expr t a) (Expr t a) (Annot t a)
  | If_ (IfTok t) (Expr t a) (ThenTok t) (Expr t a)
      (ElseTok t) (Expr t a) (Annot t a)
```

parsing *if-then-else* (3)

```
ifExpr = do
    reserved "if"
    a <- expr
    reserved "then"
    b <- expr
    reserved "else"
    c <- expr
    return (If a b c (Annot Nothing))

ifExpr_ = do
    i <- if_
    a <- expr_
    t <- then_
    b <- expr_
    e <- else_
    c <- expr_
    return (If_ i a t b e c (Annot_ Nothing $ Ws ""))
```

parsing *if-then-else* (4)

```
-- helper if_ (likewise: then_, else_)

-- originally just
-- if_ = do ws <- reserved_ "if"
--          return $ IfTok $ ws

if_ :: (TokParsing m
       , DeltaParsing m
       )
       => m (IfTok T.Text)

if_ = do ws <- reserved_ "if"
         let if' = "if"
         ws <- reserved_ if'
         return $ IfTok (T.pack if') $ ws



-- basic building block
reserved_ :: (TokenParsing m
             , Monad m
             , DeltaParsing m
             )
             => String -> m (Ws T.Text)
reserved_ s = do runUnspaced $ reserved s
```

concrete syntax

can we go back: $\mathit{Absy}_{\mathit{Bound}} \rightarrow \mathit{ConcreteSyntax}_{\mathit{Bound}}$?

- with the help of some lexical info: position etc. maybe
- not in a simple / automatic manner though, difficulties:
  - infix (mixfix) operators, like *if-then-else*
  - parenthesis/brackets (followed by white space possibly)
  - desugaring
- rethink for Idris

desugaring

$\lambda xy.\, body$

$\lambda x.\, \lambda y.\, body$

Lams ["x", "y"] body'

$\downarrow$ *desugar*

Lam "x" (Lam "y" body")

2

Succ (Succ Zero)

Nat 2

$\downarrow$ *desugar*

DCon "Succ" [...(DCon "Succ" [...(DCon "Zero"...)]...)]

binding structure

(*Unbound* vs.) Edward Kmett's *Bound* library:

- clever (fast) representation of bound and free variables, and their scopes: generalized generalized de Bruijn indices (conversion to traditional de Bruijn is available)
- we define our expressions as monads, thus *Exp a* (with free variables in *a*)
- *Scope b Exp a* then keeps track of bound variables in *b* and free variables in *a* (*Scope* is a monad transformer), but lightweight.
- really one more degree of freedom: *Exp t a* with *Scope b (Expr t) a*
- examples:

```
Exp a =
   Lam String (Scope () Exp a)
   ...

Expr t a =
     Lam   t  (Scope ()  (Expr t) a)
   | Lams [t] (Scope Int (Expr t) a)
   ...
```

Bound, example

```
>>> (V "x" :@ V "y") :@ V "x"
```

```
>>> abstract1 "x" $ V "x" :@ V "y" :@ V "x"
Scope ((V (B ()) :@ V (F (V "y"))) :@ V (B ()))
```

Bound, substitution

given some term *t*, say eg.

```
>>> let t = V "x" :@ V "n" :@ V "x"
```

can *substitute t' for n in t* (or read: *substitute n by t'*):

$$t[t'/n]$$

```
>>> substitute "n" (V "t'") t
(V "x" :@ V "t'") :@ V "x"
```

this can be seen as *binding* a function $\lambda n.t'$ to *t*:

$$t >>= \lambda n.t'$$

```
>>> t >>= (\n' -> if  n'=="n" then V "t'" else return n')
(V "x" :@ V "t'") :@ V "x"
```

Bound, substitution (2)

thus taking advantage of our monad, but need to define *bind* (>>=) beforehand:

```
instance Applicative (Exp t) where
  pure = V
  (<*>) = ap

instance Monad (Exp t) where
  return = V
  V a          >>= f = f a
  (x :@ y)     >>= f = (x >>= f) :@ (y >>= f)
  Lam n s      >>= f = Lam n (s >>>= f)
  Type          >>= _ = Type
  TrustMe (Annot ann)  >>= f = TrustMe (Annot $ (>>= f) <$> ann)
  ...
```

thereby also more fine grained control of substitution.

zipper

- navigating up and down (left and right etc.) in the syntax tree
- eg. given a variable, find it's binding occurence up in the tree, do substitution on this subtree, and return the complete tree.
- one tree datatype that contains expressions, declarations, and modules:

```
data Tr a =
    Exp { _exp :: Expr a a }
  | Dcl { _dcl :: Decl a a }
  | Mod { _mod :: Module a a }
  | Aa  { _aa ::  a }
    deriving (Show)
```

- contrary to the usual zipper implementations: a list of functions as breadcrumbs:

```
type Zipper a = (Tr a, [Tr a -> Tr a])

left :: Refactoring m => Zipper a -> m (Zipper a)
left (Exp (l :@ r), bs) =
    rsucceed (Exp l, (\(Exp l') -> Exp $ l' :@ r) : bs)

up :: Refactoring m => Zipper a -> m (Zipper a)
up (e, b:bs) = rsucceed $ (b e, bs)
```

finding syntax elements, given position information only

- rough analogy: given a text, page x, line y, column z, find the corresponding chapter, section, sentence, element in the sentence
- calculate the length of every given token, expression, decl..., and thus exact position information.
- with `Foldable`, `Traversable` derived automatically
- and `Bitraversal` defined once, in a straightforward manner in applicative style:

```
bitraverse f g = bt where

bt (V a)           = V <$> g a
bt (Lam p b)        = Lam <$> f p <*> bitraverseScope f g b
...
bt (Type)           = pure Type
bt (Type_ tt ws) = Type_ <$> f tt <*> traverse f ws
...
```

- caveat: have to take bound variables into account: `wrap`, `unwrap`

list to vector refactorings (1)

list `map` vs vector `map` in Pi-Forall

```
map : [a : Type] -> [b: Type] -> (a -> b) -> List a -> List b
map = \[a] [b] f xs . case xs of
  Nil -> Nil
  Cons y ys -> Cons (f y) (map [a][b] f ys)

map : [A:Type] -> [B:Type] -> [n:Nat] -> (A -> B) -> Vec A n -> Vec B n
map = \[A][B][n] f v.
  case v of
    Nil -> Nil
    Cons [m] x xs -> Cons [m] (f x) (map[A][B] [m] f xs)
```

functions having no implicit arguments, even operations like `Cons` are indexed by a vector size in Pi-Forall: `m` here.

`[arg]` denotes an erasable argument

list to vector refactorings (2)

list `append` vs vector `append` in Pi-Forall

```
append : [a:Type] -> List a -> List a -> List a
append = \[a] xs ys. case xs of
  Nil -> ys
  Cons x xs' -> Cons x (append [a] xs' ys)

append : [A :Type] ->[m:Nat] -> [n:Nat] -> Vec A m -> Vec A n -> Vec A (plus m n)
append = \[A] [m] [n] v1 ys . case v1 of
     Nil -> ys
     Cons [m0] x xs -> Cons [plus m0 n] x (append [A] [m0][n] xs ys)
```

list to vector refactorings (3)

- the structure of the list and vector functions are the same in each case
- but we need additional information in various places, and thus additional (erasable) arguments at times
- essentially we need to know the sizes of all the vectors involved, and their operations
- these vector sizes are given as patterns typically (as mentioned above), relating eg. the sizes of input and output vectors

list to vector refactorings (4)

idea:

- require the user to provide the desired vector signatures, as these would be too difficult to guess
- calculate the body (definition) of a vector function, ie. all the vector sizes involved
- borrowing ideas from Hindley-Milner type inference: treating the vector sizes as unknow initially (similar to type variables)
- and calculate them from the given constraints, taking into account all the available information

list to vector refactorings (5)

`PossiblyVar` allows for various vector sizes, known and unknown, and is essentially an elaborate natural number type, that we use in various places:

```
data PossiblyVar t =
  ZeroV
  | SuccV (PossiblyVar t)
  | Some t
  | VarV t
  | Sum (PossiblyVar t) (PossiblyVar t)
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)
```

- A concrete vector size of m would thus be represented as `Some "m"`.
- the concrete size *succ* m as `SuccV (Some "m")`.
- the size of a vector that we haven't decided upon yet, as `VarV "alpha"`.
- etc.

list to vector refactorings (6)

working already:

```
>>> ((\m -> (\t -> pp $ evalState (vect t) vInit) $ fromRight' $ (ezipper $
    Mod $ nopos $ t2s $ m) >>= forgetZ >>= navigate [Decl 2] >>= focus) <$>) $
     (runExceptT $ getModules ["samples"] "List") >>= return . last .
    fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/List.pi"
map = \[Some a] [Some b] [VarV alpha] (Some f) (Some xs) .
        case xs of
          Nil -> Nil
          (Cons [VarV gamma] y ys) ->
            Cons [VarV beta] ((f y)) ((map [a] [b] f ys))
```

next steps: use *unification* to decide upon these vectors sizes, taking all the given constraints into account.

Unification is the process of finding an answer to the question, if two terms can be made equal, and if so, with what substitutions.