A Dependently-Typed Zipper over GADT-Embedded ASTs

Donovan Crichton

November 2018

Preliminaries

- **Slides and examples available at:**
  `https://github.com/donovancrichton/talkdepzip.git`
- **About me:**
  - Honours 'year' student at Griffith University.
  - Working towards a type-correct genetic program through dependently-typed functional programming.
  - About 18 months experience in FP, just under 12 with dependent types.

A refresher on dependent types 1.

- The most basic definition is a dependent data type (GADT in Haskell).
- Dependent data-types depend on being parameterised over a value for their construction.
- Distinguished from parameterised ADTs by the ability to specify the return type parameter of each data constructor.

A vector dependent on a length value.

```
data Nat = Z | S Nat

data Vec : (n : Nat) -> (e : Type) -> Type where
  Nil : Vec Z e
  (::) : (x : e) -> (xs : Vec n e) -> Vec (S Z) e
```

A refresher on dependent types 2.

Why is this ‘good’?

If our length forms part of our type, we gain the ability to write correct functions with respect to vector length, without having to explicitly check.

Adding some vectors.

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

A refresher on dependent types 3.

Π types.

- The Π type is a family of types that are indexed by a value (hence type families in Haskell).
- Π types are used to calculate correct return types when given a specified value.
- In Idris Π types only fully refine if the functions requiring them are marked as total.
- In Idris functions that return Π types don't always refine in function composition, recursive calls or let bindings.

A refresher on dependent types 4.

An example of using Π types in Idris.

```
Age : Type
Age = Nat

Name : Type
Name = String

data Material = Plastic | Wood | Metal | Cheese
data Person = P Name Age
data Object = O Material

IsPerson : Bool -> Type
IsPerson True = Person
IsPerson False = Object

isPerson : (x : Bool) -> IsPerson x
isPerson True = P "Donovan Crichton" 33
isPerson False = O Cheese
```

A refresher on dependent types 5.

Σ types.

- Σ types are a pairing of a value, and a type that depends on that value (They are also called dependent pairs).
- Σ types are useful when you want some basic type inference around dependent types!
- In Idris it is difficult to extract either side of a Σ type pair. Particularly when that pair is under a constructor.
- Due to the dependent nature of the `fst` and `snd` functions on dependent pairs, they cannot be used with ordinary maps, folds, binds etc.

A refresher on dependent types 6.

An example using Σ types in Idris.

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
  MkDPair : (x : a) -> (pf : P x) -> DPair a P

-- also has some syntactic sugar in Idris.
f : (x : Bool ** IsPerson x)
f = (_ ** isPerson True)

g : Num a => (n : Nat ** Vec n a)
g = (_ ** [1, 2, 3])

-- this is particularly useful if we are passing a vector
-- of unknown length in as an argument.
len : Num a => Vec n a -> (x : Nat ** Vec x a)
len x = (_ ** x)

-- len [1, 2, 3] returns
-- (3 ** [1, 2, 3]) : (x : Nat ** Vec x Integer)
```

A refresher on dependent types 7.

Why are Σ and Π ‘good’?

- It turns out that the curry-howard isomorphsim still holds with the introduction of dependent types.
- Lets say we have some proof $P\ x$, using a dependent-data-type or a Π type is saying $\forall x, P\ x$. Using a Σ type is saying $\exists x, P\ x$.
- This brings the isomorphic logic from propositional logic to first-order or predicate logic, allowing us to write proofs in our code (more on this later).

Dependent data-type embedded DSLs 1.

A quick review of DSLs

- Short for Domain Specific Language.
- Used in lots of places, salary calculations, query and markup languages, business logic, etc.
- Used very much in programming language research!

A DDT (or GADT) embedded DSL

```
data Expr : (a : Type) -> Type where
  Lit   : a -> Expr a
  Add   : Num a => Expr a -> Expr a -> Expr a
  Const : Expr a -> Expr b -> Expr a
```

Dependent data-type embedded DSLs 2.

An embedded DSL and it's interpreter.

```
data Expr : (a : Type) -> Type where
  Lit   : a -> Expr a
  Add   : Num a => Expr a -> Expr a -> Expr a
  Const : Expr a -> Expr b -> Expr a

interp : Expr a -> a
interp (Lit x)     = x
interp (Add x y)   = (interp x) + (interp y)
interp (Const x y) = const (interp x) (interp y)
```

Why is an embedded DSL 'good'?

- It was very easy to write that interpreter.
- The type checker over the meta language takes care of type checking the DSL.
- The meta language also takes care of variable binding.

Zipping over embedded DSLs LYAH style 1.

```
data Expr : Type -> Type where
  Lit   : a -> Expr a
  Add   : Num a => Expr a -> Expr a -> Expr a
  Const : Expr a -> Expr b -> Expr a

data Context = Root
  | L (Expr a) Context
  | R (Expr a) Context

left : (Expr a, Context) -> (Expr a, Context)
left (Lit x, c)       = (Lit x, c)
left (Add x y, c)     = (x, L (Add x y) c)
left (Const x y, c) = (x, L (Const x y) c)

-- the problem comes when trying to write right
right : (Expr a, Context) -> (Expr a, Context)
right (Lit x, c)       = (Lit x, c)
right (Add x y, c)     = (y, R (Add x y) c)
right (Const x y, c) = (y, R (Const x y) c)
```

Zipper over embedded DSLs LYAH style 2.

Why doesn't this work?

- 'left' works fine because the compiler can see that all instances of left result in an 'Expr a'.
- 'right' cannot typecheck because the compiler sees that it returns an 'Expr b' in the 'Const' case.

Where to from here?

- There is nothing stopping us from writing a non-sensical context and pairing it up with some expression. We'd like to avoid this.
- There are methods for traversing these well-typed ASTs via higher-order morphisms, they must touch every element of the tree however.
- Lets see where we can get using some dependent types!

A dependently typed zipper 1.

Let's start with Π types.

- These Π types will allow us to calculate the correct type of the context, and keep us honest when developing the zipper.
- Have a look at the 'Maybe Type'. It doesn't make sense for us to build a type from going left or right on the 'Lit x' case.

```
GoLeft : Expr a -> Maybe Type
GoLeft (Lit x) = Nothing
GoLeft (Add {a} x y) = Just a
GoLeft (Const {a} x y) = Just a

GoRight : Expr a -> Maybe Type
GoRight (Lit x) = Nothing
GoRight (Add {a} x y) = Just a
GoRight (Const {b} x y) = Just b
```

A dependently typed zipper 2.

Let's re-define the context.

- By parameterising the context over our Π types, we ensure that the type checker will fail when we try to build invalid contexts.
- Moving from an ADT to a DDT (GADT) gives us a lot more expressivity here! This is also a strong example of the dependence relationship in the type parameter.

```
data Context : Maybe Type -> Type where
  Root : Context (Just a)
  L : (x : Expr a) -> Context (Just a) -> Context (GoLeft x)
  R : (x : Expr a) -> Context (Just a) -> Context (GoRight x)
```

A dependently typed zipper 3.

Let's re-define the zipper.

- We'd like to parameterise the zipper so that we can perform operations on zippers that have holes (or focii) of the same type.
- 'wrap' is provided to give us an easy way of creating a Σ type from a zipper, we wrap all this up in a 'Maybe' as some zipping operations may fail.

```
data Zipper : Type -> Type where
  Zip : Expr a -> Context (Just a) -> Zipper a

wrap : Zipper a -> Maybe (a : Type ** Zipper a)
wrap x = Just (_ ** x)
```

A dependently typed zipper 4.

Re-defining the direction functions.

```
left : Maybe (a : Type ** Zipper a)
    -> Maybe (b : Type ** Zipper b)
left Nothing = Nothing
left (Just (x ** pf)) =
  case pf of
    (Zip p@(Lit x) c) => Nothing
    (Zip p@(Add x y) c) => Just (_ ** Zip x (L p c))
    (Zip p@(Const x y) c) => Just (_ ** Zip x (L p c))

right : Maybe (a : Type ** Zipper a)
     -> Maybe (b : Type ** Zipper b)
right Nothing = Nothing
right (Just (x ** pf)) =
  case pf of
    (Zip p@(Lit x) c) => Nothing
    (Zip p@(Add x y) c) => Just (_ ** Zip y (R p c))
    (Zip p@(Const x y) c) => Just (_ ** Zip y (R p c))
```

A dependently typed zipper 5.

More direction functions

```
up : Maybe (a : Type ** Zipper a)
  -> Maybe (b : Type ** Zipper b)
up Nothing = Nothing
up (Just (x ** pf)) =
 case pf of
  (Zip e Root) => Just (_ ** Zip e Root)
  (Zip e (R (Lit x) pc)) impossible
  (Zip e (R (Add x y) pc)) => Just (_ ** Zip (Add x e) pc)
  (Zip e (R (Const x y) pc)) => Just (_ ** Zip (Const x e) pc)
  (Zip e (L (Lit x) pc)) impossible
  (Zip e (L (Add x y) pc)) => Just (_ ** Zip (Add e y) pc)
  (Zip e (L (Const x y) pc)) => Just (_ ** Zip (Const e y) pc)
```