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.

Ackknowledgements

A sincere thank you to..:

- Mr Isaac Elliot (LightAndLight)
- Mr Alex Gryzlov (clayrat)
- The kind people from the Idris channel on the discord functional programming server!

A refresher on dependent types 1.

A note on types and values.

- Types can be functions and functions can be types!
- This takes some getting used to!

```
-- n is a value in the type signature
-- n is also used as an argument to refer to that value.
len : {a : Type} -> {n : Nat} -> Vec n a -> Nat
len xs {n} = n
```

A refresnher on dependent types 2.

A note on totality

- Functions can be types.
- Functions need to be evaluated.
- We need a guarantee that our type-checker will eventually stop and give us a type.
- Any functions that are used as a type must be **total**!
- Total functions are defined for all cases and are guaranteed to terminate in some finite time.

A refresher on dependent types 3.

- The most basic definition is a dependent data type (GADT in Haskell).
- Dependent data types (DDT) 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 4.

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.

```
-- Is this total? Defined for all cases and terminating?
(+) : Num a => Vec n a -> Vec n a -> Vec n a
(+) [] [] = []
(+) (x :: xs) (y :: ys) = x + y :: xs + ys
```

A refresher on dependent types 5.

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 evaluate if the functions requiring them are marked as total.
- In Idris functions that return Π types don't always evaluate in function composition, recursive calls or let bindings.

A refresher on dependent types 6.

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 7.

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 calculation around dependent types!
- Idris defines two functions for Σ types: `fst` and `snd` for extracting the first and second elements of the pair. Similar to an ordinary product type (tuple).

A refresher on dependent types 8.

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 9.

The Curry-Howard Isomorphism.

The CH Isomorphism shows the relationship between mathematical proofs under logic, and computer programs. In idris this relationship can be expressed as follows:

Logic Term	**Logic Symbol**	**Idris Symbol**	**Idris Type**
Implication	$p \Rightarrow q$	`p -> q`	Function Arrow
Conjuction	$p \wedge q$	`(p, q)`	Pair (Product)
Disjunction	$p \vee q$	`Either p q`	Enum (Sum)
Negation	$\neg\, p$	`p -> Void`	Void Type
Equivalence	$p \equiv q$	`p = q`	Equality Type
Universal	$\forall$ x. P x	`p -> Type`	Π Type
Existantial	$\exists$ x. P x	`(x ** P x)`	Σ Type

A refresher on dependent types 9.

Why are Σ and Π ‘good’?

- Π types let us map types to values.
- We can now be more precise about function values.
- Σ types let us specify properties of types, even when we may not know the exact return type at compile time.
- ‘properties’ includes both Π types, and types that are parameterised over other types.
- This will become much clearer later!

Motivation

- Came from a research project on the automatic generation of well-typed functions to solve a given problem.
- We'd like to substitute values at specific positions on the expression tree.
- We may not know the value at the position during runtime (but we may know its type).
- A zipper allows the specification of a position in a tree via a path of transformations or rotations.
- We'd like to exchage values of the same type at different positions between two trees.

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.

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)
```

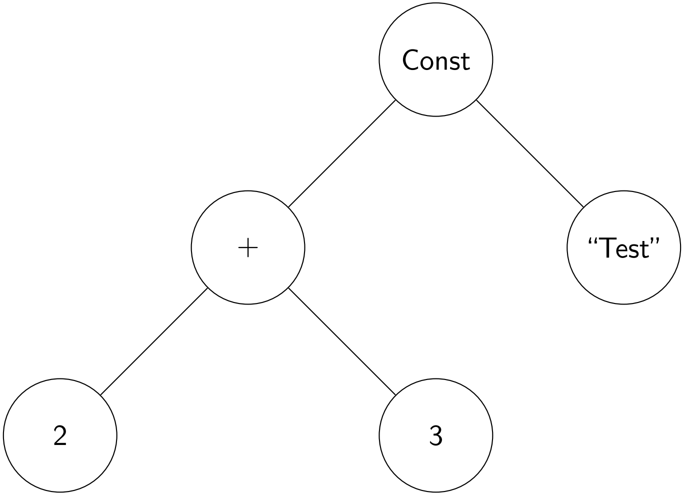
Why is an embedded DSL 'good'?

- 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.

Dependent data type embedded DSLs 3.

Expressions as Trees

```
Const (Add (Lit 2) (Lit 3)) (Lit "Test"))
```



Zipping over embedded DSLs LYAH style (Naive).

```
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 (Naive).

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 type-check because the compiler sees that it returns an ‘Expr b’ in the ‘Const’ case.

Where to from here?

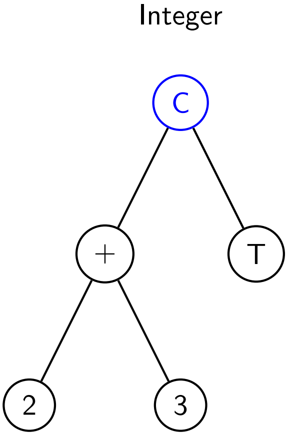
- There is nothing stopping us from writing a nonsensical context and pairing it up with some expression. We’d like some stronger guarantees here.
- Lets see where we can get using some dependent types!

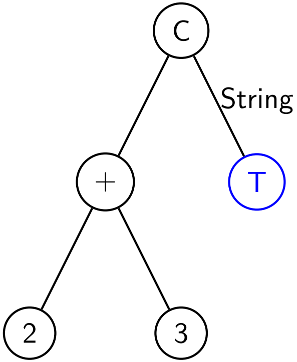
Some Intuition.

Going right down the expression tree.

```
Const (Add (Lit 2) (Lit 3)) (Lit "Test")
```

Let C represent Const. Let T represent "Test".



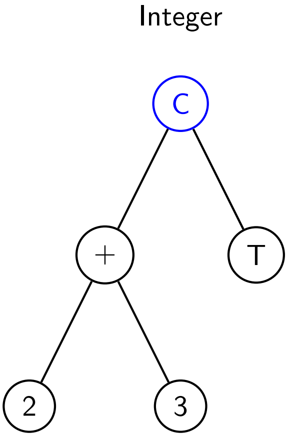


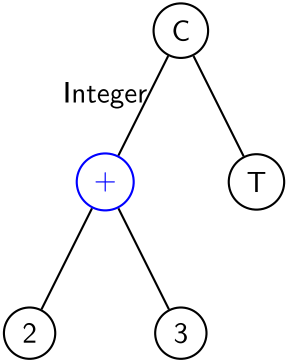
More Intuition.

Going left down the expression tree.

```
Const (Add (Lit 2) (Lit 3)) (Lit "Test")
```

Let C represent Const. Let T represent “Test”.





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.

Notes on the left and right directions.

- Thanks to the ‘GoLeft’ and ‘GoRight’ Π types we defined earlier. It’s not possible to accidentally produce a focus of the incorrect type when implementing ‘left’ and ‘right’.
- We can say that the type is now correct by construction.
- Does the context need to store the full parent expression? How much do we gain by returning a partially applied constructor? There is more to think about here!
- We need the different constructor tags for the next direction case.

A dependently typed zipper 6.

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)
```

A dependently typed zipper 7.

`up` and Σ types.

- Remember, we say a function is total when it is defined for all cases and is guaranteed to terminate in finite time.
- Π and Σ types do not refine if used to define non-total functions in Idris.
- To make ‘up’ total, we must list all the cases, however being able to construct a right and left case of ‘Lit x’ is clearly nonsense! To convince the totality checker we can mark the case as impossible.
- Σ types are ideal for when we do not know the exact type of a function, but we know some properties that the inputs and outputs must adhere to.

A dependently typed zipper 8.

A graphical representation of ‘subst’

A dependently typed zipper 9.

Substitution and evaluation.

```
subst : (x : (a : Type ** Zipper a))
      -> Expr (fst x)
      -> Maybe (a : Type ** Zipper a)
subst (x ** (Zip x' c)) e = Just (_ ** Zip e c)

data NotNothing : Maybe a -> Type where
  IsNotNothing : NotNothing (Just x)

fromMaybe : (input : Maybe (a : Type ** Zipper a))
          -> {auto prf : NotNothing input}
          -> (a : Type ** Zipper a)
fromMaybe (Just z) = z

interp : (x : (a : Type ** Zipper a)) -> (fst x)
interp (x ** (Zip e c)) = eval e
```

A dependently typed zipper 10.

Notes on substitution, interpretation and 'Just' proofs.

- We write 'subst' under the assumption that we have an ordinary Σ type parameterised by 'a' and a further expression also parameterised by 'fst x'. This ensures that the function will calculate the correct type!
- We defined 'NotNothing' and 'fromMaybe' to give us some tools to access the underlying type parameter. We can also achieve this with more Σ types.
- Once we've removed the maybe we can lean on 'fst' again to calculate the correct type for our evaluation.

But...does it actually work?

```
ex1 : Num a => Zipper a
ex1 = Zip (Const (Lit 2) (Lit "Test")) Root
-- Zip (Const (Lit 2) (Lit "Test")) Root

ex2 : Maybe (a : Type ** Zipper a)
ex2 = wrap ex1
-- Just (Integer ** Zip (Const (Lit 2) (Lit "Test")) Root)

ex3 : Maybe (a : Type ** Zipper a)
ex3 = right ex2
-- Just (String **
--   Zip (Lit "Test") (R (Const (Lit 2) (Lit "Test")) Root))

ex4 : Maybe (a : Type ** Zipper a)
ex4 = up (subst (fromMaybe ex3) (Lit "Hello"))
-- Just (Integer ** Zip (Const (Lit 2) (Lit "Hello")) Root)

ex5 : (DPair.fst (fromMaybe Main.ex4))
ex5 = interp (fromMaybe ex4)
-- 2
```

Gotchas! (Further work).

- If we're wrapping out Σ types with 'Maybe' then trying to say something like "There may possibly exist a type, such that we can build a zipper out of it." which reads `Maybe (a : Type ** Zipper a)`
- It'd probably make more sense to say "There definitely exists a type, that we may possibly build a zipper out of it." which would instead read `(a : Type ** Maybe (Zipper a))`. This means we could probably avoid the 'NotNothing' proof and the 'fromMaybe' function.
- How necessary is it to parameterise the context over a 'Maybe Type'? We can probably achieve the same results without this!
- This is in no way generic! Our zipping functions, and our context are both tightly coupled to the structure of our DSL. Is there a way we can do better?

In summary.

- We've shown the implementation of a method to correctly traverse a DDT-embedded DSL, where the types can be calculated at run-time.
- The goal here is to automate this process so that we can perform substitution and other transformations over well-typed expressions automatically.
- While there is still work to be done, hopefully this shows how dependent types can allow for both more rigour when developing software and more expressiveness when defining the types of our functions.