# CLASE: Cursor Library for A Structured Editor

## Functional Brick: Embracing Boilerplate for Yet Another Zipper Library

Tristan O.R. Allwood
Imperial College London
tora@doc.ic.ac.uk

Susan Eisenbach
Imperial College London
s.eisenbach@imperial.ac.uk

## Abstract

The "zipper" is a well known design pattern for providing a cursor-like interface to a data structure. However, the classic treatise by Huet only scratches the surface of some of the potential applications of the zipper. In this paper we take inspiration from Huet, and build a library suitable as an underpinning for a structured editor for programming languages. We consider a zipper structure that is suitable for traversing heterogeneous data types, encoding routes to other places in the tree (for bookmark or quick-jump functionality), expressing lexically bound information using contexts, and traversals for rendering a program indicating where the cursor is currently focused in the whole.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data*]: Data Structures

***General Terms*** language, context, cursor, zipper, GADT, constructor, bookmark, traversal, render

***Keywords*** zipper, cursor, boilerplate, bookmarks, traversal, generalized algebraic data types,

## 1. Introduction

### 1.1 Motivation

The Zipper (1), and its variants (3), (6), (2), are well known in the folklore as an appropriate pattern for providing a cursor-like interface to a data structure. We have started developing an interactive tool that visualises and manipulates $F_C$ (7) (the new GHC intermediate language). As the manipulations are to be done in user-selected focused areas of the tree, a cursor presentation indicating the current focus was decided upon for presenting to the user. Naturally we were drawn to a zipper-style of implementation for our underlying representation.

During implementation we borrowed inspiration from the existing literature, but came across issues that do not seem to be previously addressed; Our expected underlying data type is a programming language with a notion of binding, and we needed several ways of traversing the data type that used both the bound information (e.g. names of in-scope variables) and the location of the cursor. We also wanted to support an arbitary number of bookmarks (saved locations that the user can jump back to) into the program tree that remain valid across changes to unrelated parts of the tree.

We present here a library that has been developed out of our experiences. It is built around the ideas of the traditional Huet Zipper, but extended with the following properties:

- A library component that is agnostic to the underlying data type being traversed, and Template Haskell scripts that generate all the boilerplate to link the underlying data type to the library.
- The ability to traverse heterogeneous data types.
- A way of expressing routes from the current focus to arbitary places in the tree, with easy detection of whether local changes will invalidate these routes, and the ability to move to them - allowing easy creation of a "bookmark" facility.
- If a programming language with lexical binding is being held in the zipper, the ability for the user to express the binding strategy for the language once in an idiomatic way.
- A traversal that uses the binding specification to make available transparantly the current binding information for the local context without traversing the entire tree.
- A second traversal that would be suitable for rendering a cursor in an editor, that transparantly uses binding information, and the current cursor location. The user code to use this traversal is made very idiomatic due to some automatically generated adapter code.

It is our intention that this library could aid in the development of interactive visualisers or editors for small programming languages that need a cursor presentation to the user.

The library and some screenshots are available online at (8). We freely admit it is not a particularly small or elegant solution to our problems; there is a considerable amount of mechanically derived boilerplate involved and it requires a large number of extensions to standard Haskell '98 (GADTs, TypeFamilies, RankNTypes, MultiParamater type classes to name a few). We mitigate this by providing Template Haskell scripts to generate all the boilerplate. For this reason we lovingly consider our work as a Functional Brick, it may not look pretty, but it provides a core that works well as a foundation for a larger application.

Our aim is for a user of our library to need to provide only the data structures describing their language, the strategy for binding in their language, and how to render each constructor and the cursor position in their language. In return they get a cursor library that allows generic movement and update throughout the tree, the ability to easily create bookmarks in the tree, and ways to traverse and render the cursor in the tree. One further advantage of our approach is that the structure of the cursor is entirely done with data. Although we have not exploited this here, it does mean the structure is (in theory) persistable without too much difficulty.

### 1.2 Outline

We proceed as follows; In 2.1 we briefly introduce GADTs, and a utility data type that will be used in the paper. We introduce a small language LAM in 2.2 that will be our running example of something we wish to traverse. We explain the notion of 1 holed contexts for LAM, and introduce the `Path` data type, which, when applied to LAM's contexts and paired with an appropriate value gives us a simple cursor (2.3). Next (2.4) we introduce primitive movements for our cursor.

We then abstract in 2.5, using type classes and associated data type families to create a class of languages that our cursor is parameterised over. We move away from desiging our library tied to LAM and instead make LAM an instance of our new `Language` class.

We extend our simple cursor with a notion of a route to an arbitary place in the tree, and show how this can be used to encode bookmarks that allow easy detection of whether local modifications will invalidate them (2.6). In 2.7 we extend the definition of Language to include some utilities that allow recovery of the traditional zipper moveUp/moveLeft/moveRight/moveDown functions.

We then (2.8) use our context constructors to allow the user to express a binding strategy (should they desire to), and introduce a simple traversal to recover the names of of in scope variables in LAM. Finally we implement a traversal for rendering a Lam program, indicating where the cursor currently is, and reusing our binding code (2.9).

We finish by discussing related work in 3, and some future extensions we would like to make in 4.1.

## 2. Library

### 2.1 GADT Preliminaries

In this work we will be using a lot of extensions to Haskell that are implemented in GHC. One of the most pervasive extensions we use are Generalized Algebraic Data Types (GADTs). GADTs generalise normal Haskell data types by allowing individual constructors to refine (or specify) more specific types for the type parameters of the type.

For example, we can use a GADT to create a simple reflection scheme for a closed set of types with values:

```
data SimpleReflect t where
  IntReflect :: Int → SimpleReflect Int
  TwoBoolReflect :: Bool → Bool →
                      SimpleReflect (Bool, Bool)
```

This introduces a new type `SimpleReflect` that is parameterised by a type `t`. This type has two constructors, `IntReflect`, which carries a single `Int`, and `TwoBoolReflect`, which carries two `Bool`s.

However, both constructors refine the type `t` to more concrete values. This makes it possible to implement a function to extract the contents from either constructor:

```
extract :: SimpleReflect t → t
extract (IntReflect int) = int
extract (TwoBoolReflect b1 b2) = (b1, b2)
```

When the compiler sees a pattern match that brings into scope more information about the type, the extra type information is then available for the programmer. E.g. in the first case the pattern match on `IntReflect` refines the `t` to `Int`, and so the compiler can tell the function is well typed.

One incredibly useful GADT, which we will make much use of later, is one that witnesses type equality:

```
data TyEq a b where
  Eq :: TyEq x x
```

A pattern match on the `Eq` type constructor will bring into scope the knowledge that two type variables are actually the same type. Using this it is easy to produce a simple "cast" like function:

```
simpleCast :: TyEq a b → a → b
simpleCast Eq = id
```

i.e. given a witness that types `a` and `b` are exactly the same type, we can use the identity function to turn a value of type `a` into a value of type `b`.

### 2.2 LAM Example Language

```
data Lam = Lam Exp
data Exp
  = Abs String Type Exp
  | App Exp Exp
  | Var Integer
data Type
  = Unit
  | Arr Type Type
```

**Figure 1.** The LAM Language

In Figure 1 we present a small language, LAM, that we will use as a concrete example for our techniques in this paper. The `Lam` type marks the root of our program, and its sole constructor is a simple wrapper over a LAM expression.

Expressions are either lambda abstractions, (`Abs`) which carry a `String` name for their variable, a `Type` for their variable and an expression (`Exp`) in which the variable is in scope. Application expressions are the familiar application of two expressions to each other. Variable expressions carry a de Bruijn index (9) indicating which enclosing `Abs` binds the variable this `Var` refers to.

Types are either arrow types (`Arr`) or some notional unital type (`Unit`).

For example, we would expect the following Lam program to represent the term $\lambda x :: \tau \rightarrow \tau.(x \circ \lambda y :: \tau.(y \circ x))$:

```
Lam (Abs "x" (Unit `Arr` Unit) $
  (Var 0) `App` (Abs "y" Unit $ (Var 0) `App` (Var 1)))
```

It is our desire to allow a cursor to navigate between `Exp`, `Type` and `Lam` types.

### 2.3 Towards a simple Cursor

In terms of cursor design, the first thing to notice about the LAM language is that it is heterogeneous. At certain points in the design, we may know that we are dealing with some type in the LAM language (`Lam`, `Exp` or `Type`), but we don't know which part. To deal with these cases, we will need some form of reflection scheme to allow us to get back to the current type. There are several libraries that already exist to do this (for example `Data.Typeable`), however here we want a closed scheme that can only let us get back into the LAM language. For this we use a (mechanically derived) simple GADT, and type class:

```
data TypeRepI a where
  ExpT :: TypeRepI Exp
  LamT :: TypeRepI Lam
  TypeT :: TypeRepI Type
class ReifyLam a where
  reifyI :: a → TypeRepI a
```

```
instance ReifyLam Exp where reifyI _ = ExpT
instance ReifyLam Lam where reifyI _ = LamT
instance ReifyLam Type where reifyI _ = TypeT
```

The core of a zipper library is the notion of a context. This represents a constructor in the original language but with a hole in it for the current item in focus. Contexts are then chained together (usually by a 'moveUp' field) to allow focus at an arbitary place in the tree. Here, we have separated the contexts from the chaining, so our contexts only represent a constructor with a hole in them. Because our contexts can 'cross types' (e.g. the cursor could be focused on the `Type` inside an `Abs` constructor (of type `Exp`)), they are GADTs that hold the type the Context goes from (what the type of the hole is) and to (what the type of the constructor that has the hole in it is).

The LAM language contexts are then (mechanically derivable):

```
data ContextI from to where
  TypeToAbs :: String → Exp → ContextI Type Exp
  ExpToAbs :: String → Type → ContextI Exp Exp
  ExpToApp0 :: Exp → ContextI Exp Exp
  ExpToApp1 :: Exp → ContextI Exp Exp
  ExpToLam :: ContextI Exp Lam
  TypeToArr0 :: Type → ContextI Type Type
  TypeToArr1 :: Type → ContextI Type Type
```

Given a `ContextI from to` and an item of type `from`, we can then build something of type `to`, vis:

```
buildOneI :: ContextI from to → from → to
buildOneI (TypeToAbs x0 x1) h = Abs x0 h x1
buildOneI (ExpToAbs x0 x1) h = Abs x0 x1 h
buildOneI (ExpToApp0 x0) h = App h x0
buildOneI (ExpToApp1 x0) h = App x0 h
buildOneI (ExpToLam) h = Lam h
buildOneI (TypeToArr0 x0) h = Arr h x0
buildOneI (TypeToArr1 x0) h = Arr x0 h
```

A single `ContextI from to` represents a constructor with a hole in it, but to represent an arbitary place in the tree with a hole in it, we need to chain the contexts together. If our contexts were ordinary data types we could use a list, however we need to ensure that the `to` parameter of our first `ContextI` matches up with the `from` parameter of the next `ContextI`. To do this we use a new data type called `PathLam`.

```
data PathLam ctr from to where
  Stop :: PathLam ctr anywhere anywhere
  Step :: (ReifyLam middle) ⇒ ctr from middle →
    PathLam ctr middle to → PathLam ctr from to
```

`Stop` is akin to the nil (`[]`) at the end of a list, and `Step` is akin to cons (`:`). Since the intermediate location (`middle`) in `Step` is existentially quantified, we need to provide a way of extracting it's type at a later time, and hence the class constraint on `ReifyLam middle`.

A simple cursor for the LAM language can now be given:

```
data CursorLam here = (ReifyLam here) ⇒ Cursor{
  it :: here,
  ctx :: PathLam ContextI here Lam
}
```

The current point of focus is denoted by `it`, and the context we are in (`ctx`) is a path from `here` up to the root of our language, `Lam`.

### 2.4 Specific Movement

Our simple cursor is not currently very useful as we do not support a way of moving around the tree. Movement in our library is based upon primitive combinators that express a single movement from a specific constructor to a type (for downward movements), or from a type into a constructor (for upward movements). Later (in 2.7) we will recover the more familiar generic move up / down / left / right operations that zipper libraries traditionally provide.

Our combinators are provided as GADT constructors as we will be able to re-use them later when talking about routes (bookmarks) and generic traversals. Having constructors (as opposed to functions) also gives us entities that can (in theory - not implemented yet) be persisted and restored.

As already indicated the primitive movements can either be upward or downward movements. Later we will wish to put constraints on which type of movement is used in some places, so we model in the type system (using an empty data declaration and a type family) the notion of up and down, and how to invert these directions.

```
data Up
data Down
type family Invert d :: *
type instance Invert Up = Down
type instance Invert Down = Up
```

Movements in LAM follow the style of the one hole construtors, they relate a constructor to a type. `Down` movements go from a constructor to a type in that constructor. Since `Up` movements are the exact opposite of a `Down` movement, we re-use our down movements to create an up one.

```
data MovementI direction from to where
  MUp :: MovementI Down to from →
    MovementI Up from to
  MAbsToType :: MovementI Down Exp Type
  MAbsToExp :: MovementI Down Exp Exp
  MAppToExp0 :: MovementI Down Exp Exp
  MAppToExp1 :: MovementI Down Exp Exp
  MLamToExp :: MovementI Down Lam Exp
  MArrToType0 :: MovementI Down Type Type
  MArrToType1 :: MovementI Down Type Type
```

We can also mechanically provide a simple combinator to invert a movement, making use of our `Invert` type family.

```
invertMovementI :: MovementI d a b →
  MovementI (Invert d) b a
invertMovementI (MUp dwn) = dwn
invertMovementI MAbsToType = MUp (MAbsToType)
invertMovementI MAbsToExp = MUp (MAbsToExp)
invertMovementI MAppToExp0 = MUp (MAppToExp0)
invertMovementI MAppToExp1 = MUp (MAppToExp1)
invertMovementI MLamToExp = MUp (MLamToExp)
invertMovementI MArrToType0 = MUp (MArrToType0)
invertMovementI MArrToType1 = MUp (MArrToType1)
```

Our aim for movements is to implement a function that takes a `MovementI d from to` and a `CursorLam from`, and if the movement is appropriate for the cursor, returns a new `CursorLam to`, i.e.

```
applyMovement :: MovementI d from to →
  CursorLam from → Maybe (CursorLam to)
```

We could implement this function mechanically in one big boiler-plated mess. However we have instead split it into a small (fairly idiomatic) core that requires several simple boiler-plate functions implementing. We will give an overview of how we intend to implement this method, introducing the boilerplate dependencies as they arise, before giving it's implementation in those terms.

***Moving Up:*** In the case that `applyMovement` is applied to an `Up` movement, we need to check that the intended `Up` movement co-incides with the first `ContextI` in the `CursorLam ctx` path. If it does, we can use `buildOne` to rebuild the item above us in the tree, and unpeel the `ContextI` from the `ctx` path.

As already mentioned, there is a close correspondence between `ContextI`s and `MovementI Up`s. We make this correspondence clear by providing a function to extract the corresponding `Up` movement from a `ContextI`:

```
ctxToMovementI :: ContextI a b → MovementI Up a b
ctxToMovementI (TypeToAbs _ _) = MUp MAbsToType
ctxToMovementI (ExpToAbs _ _) = MUp MAbsToExp
ctxToMovementI (ExpToApp0 _) = MUp MAppToExp0
ctxToMovementI (ExpToApp1 _) = MUp MAppToExp1
ctxToMovementI (ExpToLam) = MUp MLamToExp
ctxToMovementI (TypeToArr0 _) = MUp MArrToType0
ctxToMovementI (TypeToArr1 _) = MUp MArrToType1
```

If we then add a notion of equality between `MovementI`s that checks whether two `MovementI`s start from and go to the same places (and provides proofs of these equalities should they exist)

```
movEqI :: MovementI d x y → MovementI d a b →
          Maybe (TyEq x a, TyEq y b)
movEqI (MUp a) (MUp b) = fmap (λ(x, y) → (y, x)) $
  movEqI a b
movEqI MAbsToType MAbsToType = Just (Eq, Eq)
movEqI MAbsToExp MAbsToExp = Just (Eq, Eq)
movEqI MAppToExp0 MAppToExp0 = Just (Eq, Eq)
movEqI MAppToExp1 MAppToExp1 = Just (Eq, Eq)
movEqI MLamToExp MLamToExp = Just (Eq, Eq)
movEqI MArrToType0 MArrToType0 = Just (Eq, Eq)
movEqI MArrToType1 MArrToType1 = Just (Eq, Eq)
movEqI _ _ = Nothing
```

Checking the `Up` movement co-incides with the `ContextI` then becomes as simple as:

```
contextMovementEqI :: ContextI a b →
  MovementI Up a c → Maybe (TyEq b c)
contextMovementEqI ctx mov
  = fmap snd ((ctxToMovementI ctx) `movEqI` mov)
```

***Moving down:*** In the case that `applyMovement` is applied to a `Down` movement we essentially need to "unbuild" one layer, providing a context based on the constructor at `it` in the cursor, and a new value based for the new hole. This is the dual to `buildOneI`, and is hence named `unbuildOneI`:

```
unbuildOneI :: MovementI Down a b →
  a → Maybe (ContextI b a, b)
unbuildOneI mov here = case mov of
  MAbsToType → case here of
    (Abs x0 h x1) → Just $ (TypeToAbs x0 x1, h)
    _ → Nothing
  MAbsToExp → case here of
    (Abs x0 x1 h) → Just $ (ExpToAbs x0 x1, h)
    _ → Nothing
  MAppToExp0 → case here of
    (App h x0) → Just $ (ExpToApp0 x0, h)
    _ → Nothing
  MAppToExp1 → case here of
    (App x0 h) → Just $ (ExpToApp1 x0, h)
    _ → Nothing
  MLamToExp → case here of
    (Lam h) → Just $ (ExpToLam, h)
    _ → Nothing
  MArrToType0 → case here of
    (Arr h x0) → Just $ (TypeToArr0 x0, h)
    _ → Nothing
  MArrToType1 → case here of
    (Arr x0 h) → Just $ (TypeToArr1 x0, h)
    _ → Nothing
```

All that is missing to implement `applyMovement` is a way of checking whether we have an `Up` or `Down` movement. In a similar way to the reflection scheme provided for LAM we use a GADT and a projection function (`reifyDirectionT`) to aquire a representation for whether we have an `Up` or `Down` movement.

```
data DirectionT a where
  UpT :: DirectionT Up
  DownT :: DirectionT Down
reifyDirectionI :: MovementI d a b → DirectionT d
reifyDirectionI d = case d of
  (MUp _) → UpT
  MAbsToType → DownT
  MAbsToExp → DownT
  MAppToExp0 → DownT
  MAppToExp1 → DownT
  MLamToExp → DownT
  MArrToType0 → DownT
  MArrToType1 → DownT
```

`applyMovement` is then implemented as follows. Notice that in the `UpT` case we need the `Eq` type equality to prove to GHC that the result location (`to`) indicated by the provided movement really does intersect with the (up to that point) existentially bound `middle` type variable from the variable `up` in the `(Step up ups)` pattern match.

```
applyMovement :: MovementI d from to →
  CursorLam from → Maybe (CursorLam to)
applyMovement mov (Cursor it ctx)
  = case (reifyDirectionI mov) of
    UpT   → case ctx of
      Step up ups →
        case (up `contextMovementEqI` mov) of
          Just Eq → Just $ Cursor (buildOneI up it) ups
          Nothing → Nothing
      Stop → Nothing
    DownT →
      fmap (λ(ctx', it') → Cursor it' (Step ctx' ctx))
           (unbuildOneI mov it)
```

### 2.5 Generalizing LANGUAGE

Thus far we have created a cursor library specifically tied to our LAM language. However the code presented so far falls into three groups:

1. The description of LAM which comes from the user.
2. The data types that are derived directly from the structure of LAM (`ContextI`, `TypeRepI`, `MovementI`) and the functions that explicitly know about their implementation by pat-

tern matching on them (e.g. `buildOneI`, `invertMovementI`, `movEqI`).

3. Functions and data types that do not explictly need to use the structure of the LAM language or the data types derived from it (e.g. `PathLam`, `CursorLam`, `applyMovement`).

In theory, if the description of LAM from the user changes, we should be able to mechanically re-derive the data and functions in the group 2, and not need to change the functions in group 3. Of course should the user change the name of the root type (`Lam`) then group 3 would need to change slightly (`PathLam` would hardly still be an appropriate name!). What we wish to do is capture that change and make it a parameter of the items in 3; we can then provide them as a generic library.

Our approach is to create a Haskell typeclass `Language` that takes a single parameter (hereafter `l`) that represents the users language. Since we have been working with the LAM language, it is an instance of `Language Lam` that we shall use as our example.

Using associated data type families, we can model the need for our `Language` to provide data types akin to `TypeRepI`, `ContextI` and `MovementI`. We can also express the need to provide the functions as mentioned in group 2 above as members of the `Language` type class.

This gives us an initial model of a generic language:

```
class Language l where
  data Context l :: * → * → *
  data Movement l :: * → * → * → *
  data TypeRep l :: * → *
  buildOne :: Context l a b → a → b
  unbuildOne :: Movement l Down a b →
    a → Maybe (Context l b a, b)
  invertMovement :: Movement l d a b →
    Movement l (Invert d) b a
  movEq :: Movement l d a b →
    Movement l d a c → Maybe (TyEq b c)
  reifyDirection :: Movement l d a b →
    DirectionT d
  ctxToMovement :: Context l a b → Movement l Up a b
```

We also need to provide a way of getting from an item in a particular language to a `TypeRep` for that language, generalising `ReifyLam`[1].

```
class Reify l a where
  reify :: a → TypeRep l a
```

Our notion of Paths can also be generalised to rely on our more general `Reify` as opposed to the previous `ReifyLam`. `PathLam` now becomes:

```
data Path l ctr from to where
  Stop :: Path l ctr anywhere anywhere
  Step :: (Reify l middle) ⇒ ctr from middle →
    Path l ctr middle to → Path l ctr from to
```

And this means our `CursorLam` can become more general, using the more general `Path`s and `Reify`.

```
data Cursor l here = (Reify l here) ⇒ Cursor{
  it :: here,
  ctx :: Path l (Context l) here l
}
```

[1] The alert reader will realise that the generic data types up to now called `FooLam` are being generalised to `Foo l`, which, when instantiated by `Lam` will become `Foo Lam`

We also have to update two other functions to become more general (`contextMovementEqI` and `applyMovementI`), that only rely on our `Language` typeclass. The changes are trivial, and result in functions with signatures:

```
applyMovement :: (Language l, Reify l a, Reify l b) ⇒
  Movement l d a b → Cursor l a → Maybe (Cursor l b)
contextMovementEq :: (Language l) ⇒ Context l a b →
  Movement l Up a c → Maybe (TyEq b c)
```

We can also provide the instance for `Language Lam` using what we have already written. Because (at the time of writing) GHC data families do not support instances that are GADTs, we need to make the data instances wrappers around the already written data structures as opposed to giving the implementation directly. E.g. our instance for `Context Lam` is a wrapper constructor around `ContextI` called `CW`.

```
instance Language Lam where
  data Context Lam from to = CW (ContextI from to)
  data Movement Lam d from to
    = MW (MovementI d from to)
  data TypeRep Lam t = TW (TypeRepI t)
  buildOne (CW x) = buildOneI x
  unbuildOne (MW m) a
    = fmap (first CW) (unbuildOneI m a)
  invertMovement (MW x) = MW (invertMovementI x)
  movEq (MW x) (MW y)
    = fmap snd $ movEqI x y
  reifyDirection (MW x) = reifyDirectionI x
  ctxToMovement (CW x)
    = MW (ctxToMovementI x)
```

We also need to provide instances for `Reify Lam`, these need to wrap up the LAM specific `TypeRepI` value in the `TypeRep Lam` wrapper `TW`:

```
instance Reify Lam Exp where
  reify = const $ TW ExpT
instance Reify Lam Lam where
  reify = const $ TW LamT
instance Reify Lam Type where
  reify = const $ TW TypeT
```

### 2.6 Routes and Bookmarks

A `Path Lam (Context Lam) here Lam` will give a simple location in a `Lam` tree, and a way of getting back to the root from it.

However, an editor using our data structure may want to keep track of multiple locations in the tree (e.g. to provide bookmark or quick-jump functionality). Ideally we would like these bookmarks to be persistent across updates to the tree, and where this is not possible, for there to be some way of dealing with the now invalidated bookmarks.

Any position in the tree can be reached from any other by a series of `Up` movements, followed by a series of `Down` movements. This can be made into a unique route by disallowing the last `Up` movement to be the inverse of the first `Down` movement; vis

```
data Route l from to where
  Route :: (Reify l mid) ⇒
    Path l (Movement l Up) from mid →
    Path l (Movement l Down) mid to → Route l from to
```

With the additional invariant that the following predicate always holds:

$$
\begin{aligned}
&route\_invariant :: (Language\ l) \Rightarrow Route\ l\ from\ to \rightarrow Bool\\
&route\_invariant\ (Route\ (Step\ mup\ Stop)\ (Step\ mdown\ \_))\\
&\quad = (\neg \circ isJust)\ res\\
&\quad \textbf{where}\\
&\qquad res = (invertMovement\ mup\ \text{`}movEq\text{`}\ mdown)\\
&route\_invariant\ (Route\ (Step\ \_\ ups)\ downs)\\
&\quad = route\_invariant\ (Route\ ups\ downs)\\
&route\_invariant\ (Route\ Stop\ \_) = True
\end{aligned}
$$

We can now add a `Route` to our cursor so that it can keep a path back to some marked location. We provide an API for extending the current route by a single movement, resetting it, joining two routes together and making a cursor follow a route.

$$
\begin{aligned}
&\textbf{data}\ Cursor\ l\ x\ a = (Reify\ l\ a) \Rightarrow Cursor\{\\
&\quad it :: a,\\
&\quad ctx :: Path\ l\ (Context\ l)\ a\ l,\\
&\quad log :: Route\ l\ a\ x\\
&\}\\
&updateRoute :: (Language\ l, Reify\ l\ a, Reify\ l\ b) \Rightarrow\\
&\quad Movement\ l\ d\ a\ b \rightarrow Route\ l\ a\ c \rightarrow Route\ l\ b\ c\\
&resetLog :: Cursor\ l\ x\ a \rightarrow Cursor\ l\ a\ a\\
&appendRoute :: (Language\ l, Reify\ l\ a,\\
&\qquad\qquad Reify\ l\ b, Reify\ l\ c) \Rightarrow\\
&\quad Route\ l\ a\ b \rightarrow Route\ l\ b\ c \rightarrow Route\ l\ a\ c\\
&followRoute :: (Language\ l) \Rightarrow\\
&\quad Cursor\ l\ x\ a \rightarrow Route\ l\ a\ c \rightarrow Maybe\ (Cursor\ l\ x\ c)
\end{aligned}
$$

We now have to modify the `applyMovement` function to also use `updateRoute` to update the `log` as we navigate around the tree.

This `appendRoute` function allows an application to keep a collection of bookmarks into a tree. It does this by creating empty routes at the appropriate places, and then as the cursor naviages away, the bookmark (the route) is updated using `appendRoute` with the new cursor's log. This log is then reset until the next motion, when `appendRoute` can be used again to keep the bookmark in sync.

Should a bookmark then wish to be jumped to, the appropriate route is looked up, and the cursor moved to it using `followRoute`.

If a local change is to be made to a program, it is easy to check if any `Routes` point inside the current cursor location, and may need to be invalidated. If the up movement path of the `Route` is `Stop` then the route points inside.

It is also worth pointing out that the type parameters on the cursor and any user bookmark routes can be used to ensure bookmarks stay in step with cursor movements. By holding them in a data structure that requires the type parameters to be the same, we enforce the user updates the bookmarks in step with cursor updates. For example, the generic GUI application that comes with the library is based around the following GADT which holds a `Cursor` and a set of bookmarks (map of bookmark id to a route from the cursor to somewhere else).

$$
\begin{aligned}
&\textbf{data}\ CursorHolder\ l\ \textbf{where}\\
&\quad CH :: (LanguageGUI\ l) \Rightarrow Cursor\ l\ a\ a \rightarrow\\
&\qquad\quad Map\ Int\ (ExistsR\ l\ (Route\ l\ a)) \rightarrow\\
&\qquad\quad CursorHolder\ l
\end{aligned}
$$

Since we don't know where the routes end up, we wrap them in an existential wrapper that contains a `Reify` constraint:

$$
\begin{aligned}
&\textbf{data}\ ExistsR\ l\ (r :: * \rightarrow *)\ \textbf{where}\\
&\quad ExistsR :: (Reify\ l\ a) \Rightarrow r\ a \rightarrow ExistsR\ l\ r
\end{aligned}
$$

### 2.7 Recovering Generalized Motions

$$
\begin{aligned}
&\textbf{class}\ Language\ l\ \textbf{where}\\
&\quad ...\ as\ before\ ...\\
&\quad downMoves :: TypeRep\ l\ a \rightarrow\\
&\qquad [ExistsR\ l\ (Movement\ l\ Down\ a)]\\
&\quad moveLeft :: Movement\ l\ Down\ a\ x \rightarrow\\
&\qquad Maybe\ (ExistsR\ l\ (Movement\ l\ Down\ a))\\
&\quad moveRight :: Movement\ l\ Down\ a\ x \rightarrow\\
&\qquad Maybe\ (ExistsR\ l\ (Movement\ l\ Down\ a))
\end{aligned}
$$

**Figure 2.** Additions to Language to support generlized motions

Thus far our cursor is only able to move if given an exact `Movement` that precisely matches the structure of the local tree. However, for general purpose navigation, four generic movement operators (up, down, left and right) are more convinient. Here, we make the move up / down functions take a `Cursor` and, if movement in the direction requested is possible, return an existential wrapper containing the new cursor and the movement that was applied: So

$$
\begin{aligned}
&\textbf{data}\ CursorWithMovement\ l\ d\ x\ from\ \textbf{where}\\
&\quad CWM :: (Reify\ l\ to) \Rightarrow Cursor\ l\ x\ to \rightarrow\\
&\qquad Movement\ l\ d\ from\ to \rightarrow\\
&\qquad CursorWithMovement\ l\ d\ x\ from\\
&genericMoveUp :: (Language\ l) \Rightarrow Cursor\ l\ x\ a \rightarrow\\
&\quad Maybe\ (CursorWithMovement\ l\ Up\ x\ a)\\
&genericMoveDown :: (Language\ l) \Rightarrow Cursor\ l\ x\ a \rightarrow\\
&\quad Maybe\ (CursorWithMovement\ l\ Down\ x\ a)
\end{aligned}
$$

`genericMoveUp` is the simplest operation. We unpeel one context from the context path in the cursor, and we use `buildOne` to get what we are now looking at. However to update the route, and to return in the `CursorWithMovement` we will need to convert the `Context` into the `Movement Up` that it corresponds to using `ctxToMovement`.

$$
\begin{aligned}
&genericMoveUp\ (Cursor\ it\ (Step\ up\ ups)\ log)\\
&\quad = Just\ (CWM\ (Cursor\ (buildOne\ up\ it)\\
&\qquad\qquad\qquad\qquad ups\\
&\qquad\qquad\qquad\qquad (updateRoute\ upMov\ log))\ upMov)\\
&\quad \textbf{where}\\
&\qquad upMov = ctxToMovement\ up\\
&genericMoveUp\ (Cursor\ \_\ Stop\ \_) = Nothing
\end{aligned}
$$

To implement a generic, depth-first downward movement, we require the language to provide (in a depth-first order) all possible down movements for a given type in the language. We then simply try applying all the down motions, appropriate for where the cursor currently is, and take the first that succeeds.

$$
\begin{aligned}
&genericMoveDown\ cursor@Cursor\{\,\}\\
&\quad = msum \circ\\
&\qquad map\ (\lambda(ExistsR\ c) \rightarrow fmap\ (flip\ CWM\ c) \circ\\
&\qquad flip\ applyMovement\ cursor\ \$\ c) \circ\\
&\qquad downMoves \circ reify \circ it\ \$\ cursor
\end{aligned}
$$

Generally moving left or right are compound actions consisting of an up movement followed by a down movement on the sibling either directly left or right of the original location. Conceptually, if you have a down movement, you should be able to derive the movement to use to go down to the sibling left or right of where the original movment would go (`moveLeft` and `moveRight`). We then implement `genericMove(Left/Right)` by using a `genericMoveUp` to move upwards, `invertMovement` to turn the up movement into a down one, `move(Left/Right)` to get our left/right sibling and then `applyMovement` to use it.

$$genericMoveLeft = genericMoveSideways\ moveLeft$$

$$genericMoveRight = genericMoveSideways\ moveRight$$

$$\begin{array}{l} genericMoveSideways :: \forall\ l\ x\ a.(Language\ l) \Rightarrow \\ \quad (\forall\ a\ z.Movement\ l\ Down\ a\ z \rightarrow \\ \qquad\qquad Maybe\ (ExistsR\ l\ (Movement\ l\ Down\ a))) \rightarrow \\ \quad Cursor\ l\ x\ a \rightarrow Maybe\ (ExistsR\ l\ (Cursor\ l\ x)) \\ genericMoveSideways\ fn\ cursor = \mathbf{do} \\ \quad (CWM\ cursor'\ upmov) \leftarrow genericMoveUp\ cursor \\ \quad \mathbf{let}\ downmov = invertMovement\ upmov \\ \quad (ExistsR\ newDownMov) \leftarrow fn\ downmov \\ \quad cursor'' \leftarrow applyMovement\ newDownMov\ cursor' \\ \quad return\ \$\ ExistsR\ cursor'' \end{array}$$

One final generic motion we provide is the ability to move the cursor up to the root of the tree. This just walks the path of contexts, and uses our `ctxToMovement` function for updating the Route in a similar way to `genericMoveUp`.

$$\begin{array}{l} moveToRoot :: (Language\ l) \Rightarrow Cursor\ l\ x\ a \rightarrow Cursor\ l\ x\ l \\ moveToRoot\ cursor@(Cursor\ \_\ Stop\ \_) = cursor \\ moveToRoot\ (Cursor\ it\ (Step\ up\ ups)\ log) \\ \quad = moveToRoot\ (Cursor\ (buildOne\ up\ it) \\ \qquad ups \\ \qquad (updateRoute\ (ctxToMovement\ up)\ log)) \end{array}$$

### 2.8 Binding

One of the major motivations for this library is to provide a cursor library that supports languages requiring lexically bound information for operations at the cursor point. For example in the LAM language, the names of variables in scope are lexically bound by the enclosing `Abs` constructors from the current location.

Abstractly, if we were performing some operations on a LAM subterm, we may wish to do it in the context of a structure (e.g. `Monad`) that provided an API for performing operations in the presence of a new variable added to the most local scope, for looking up the name of a variable given it's deBruijn index, and to get the map containing all the variables in scope:

$$\begin{array}{l} \mathbf{class}\ LamBinder\ c\ \mathbf{where} \\ \quad addBinding :: String \rightarrow c\ a \rightarrow c\ a \\ \quad lookupBinding :: Integer \rightarrow c\ (Maybe\ String) \\ \quad getBindingMap :: c\ (Map\ Integer\ String) \end{array}$$

In the simplest case, if we just want to compute a single value of type `a` with this API, a simple implementation based on a `Data.Map Integer String -> a` would suffice. We make good use of the `GeneralizedNewtypeDeriving` extension to save us from manually creating the instances of `Functor`, `Applicative` and `Monad`.

$$\begin{array}{l} \mathbf{newtype}\ LamBinderImpl\ a \\ \quad = LBI\{fromLBI :: Map\ Integer\ String \rightarrow a\} \\ \quad \mathbf{deriving}\ (Functor, Applicative, Monad) \end{array}$$

$$\begin{array}{l} \mathbf{instance}\ LamBinder\ LamBinderImpl\ \mathbf{where} \\ \quad addBinding\ s\ (LBI\ f1) = (LBI\ f2) \\ \qquad \mathbf{where} \\ \qquad\quad f2 = f1 \circ Map.insert\ 0\ s \circ \\ \qquad\qquad\qquad Map.mapKeysMonotonic\ succ \\ \quad lookupBinding\ i = LBI\ (Map.lookup\ i) \\ \quad getBindingMap = LBI\ id \end{array}$$

Since we are using deBruijn indexes for variable references, adding a new variable to the most local scope (`addBinding`) is implemented by increasing the index (key) for all existing local variables by one, and then adding our new binding at (the now available) index 0.

The `LamBinder` type class, and the implementation `LamBinderImpl` need to be provided by the user as the capabilities required there are completely user-language dependent. Also completley user language dependent is when in a traversal functions like `addBinding` (the functions that do the binding of new information) need to be used. However, for languages that are completely lexically bound, it is possible to express when the functions like `addBinding` need to be called in a traversal without having to inline their calls into all traversals.

The idea is to provide the user with a `Context` from their language, and a value representing the result of the traversal up to the hole in the context. The user then modifies the hole adding any appropriate bound information. Concretely, we provide the following class the user can implement:

$$\begin{array}{l} \mathbf{class}\ (Language\ l) \Rightarrow BoundLanguage\ l\ t\ \mathbf{where} \\ \quad bindingHook :: Context\ l\ from\ to \rightarrow t \rightarrow t \end{array}$$

In the case of LAM, the implementation binds a new variable name in the `Exp` value inside any `Abs` constructors. If we arn't moving from an `Exp` into an `Abs`, then there is no extra binding to add, and we can directly return the sub-value:

$$\begin{array}{l} \mathbf{instance}\ (LamBinder\ c) \Rightarrow BoundLanguage\ Lam\ (c\ a) \\ \mathbf{where} \\ \quad bindingHook\ ctx\ hole = \mathbf{case}\ ctx\ \mathbf{of} \\ \qquad (CW\ (ExpToAbs\ str\ \_)) \rightarrow addBinding\ str\ hole \\ \qquad \_ \rightarrow hole \end{array}$$

We can now use `bindingHook` to provide some binding-aware generic traversals. In 2.9 we will build up a fairly complex traversal suitable for rendering a language with binding information and also indicating where the cursor is. First, we shall build a simple function that allows the computation of a value at the current cursor location, with all the necessary bound information in scope.

Our function takes a `Cursor`, and a function to compute some result from the current focus:

$$\begin{array}{l} inBindingScope :: (BoundLanguage\ l\ t) \Rightarrow \\ \quad (a \rightarrow t) \rightarrow Cursor\ l\ x\ a \rightarrow t \\ inBindingScope\ fn\ (Cursor\ it\ ctx\ \_) = foldUp\ (fn\ it)\ ctx \end{array}$$

The implementation is based upon a helper function that will walk up the `Path` of `Contexts` in the `Cursor`, nesting the underlying result value (`fn it`) in the appropriate `bindingHook` calls for the cursor's location.

$$\begin{array}{l} foldUp :: (BoundLanguage\ l\ t) \Rightarrow \\ \quad t \rightarrow Path\ l\ (Context\ l)\ a\ b \rightarrow t \\ foldUp\ t\ Stop = t \\ foldUp\ t\ (Step\ ctx\ nxt) = foldUp\ (bindingHook\ ctx\ t)\ nxt \end{array}$$

For a LAM cursor, getting hold of the currently in scope variables is then as easy as asking for the binding map in the current scope, unwrapping the `LamBinderImpl`, and calling the underlying function with an empty `Map`.

$$\begin{array}{l} varsInScope :: Cursor\ Lam\ x\ a \rightarrow Map\ Integer\ String \\ varsInScope = (\$Map.empty) \circ fromLBI \circ \\ \qquad\qquad inBindingScope\ (const\ getBindingMap) \end{array}$$

### 2.9 Rendering

Our final contribution is a traversal mechanism for our cursor that computes a single result for the whole tree stored in our cursor.

However this result can be dependant on where the cursor is and any binding information present. The motivating example we provide for this traversal is the ability to render our cursor, so a structured editor application based on our library can present the program being worked on with a marker indicating where the cursor is.

Our library is going to expect the user to implement an API that describes how to compute the resulting value from a complete constructor in the tree (e.g. `Abs`, `App`, `Var`), or from a constructor with a hole in it (i.e a `Context`). The user should also describe how to modify the result when it is conceptually under the point of the cursor. This is specified in a type class by the three functions `visitStep`, `visitPartial` and `cursor` respectively:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{BoundLanguage}\ l\ t) \Rightarrow \mathit{Traversal}\ l\ t\ \textbf{where}\\
\quad \mathit{visitStep} :: (\mathit{Reify}\ l\ a) \Rightarrow a \rightarrow\\
\qquad (\forall\ b \circ \mathit{Reify}\ l\ b \Rightarrow \mathit{Movement}\ l\ \mathit{Down}\ a\ b \rightarrow t) \rightarrow t\\
\quad \mathit{visitPartial} :: \mathit{Context}\ l\ a\ b \rightarrow b \rightarrow t \rightarrow\\
\qquad (\forall\ c \circ \mathit{Reify}\ l\ c \Rightarrow \mathit{Movement}\ l\ \mathit{Down}\ b\ c \rightarrow t) \rightarrow t\\
\quad \mathit{cursor} :: l \rightarrow t \rightarrow t
\end{array}
$$

***visitStep*** An implementation of `visitStep` will take a complete part of the tree (e.g. an `Abs` constructor), and a function (which we will name `recurse`) that provides the results of the traversal from the navigable children of that constructor. `recurse` will also ensure that `bindingHook` gets called before computing the results from the children. In the rendering example, the idea is that an implementation of `visitStep` will use `recurse` to generate the text of the children, and then combine them together with some connectives.

***visitPartial*** An implementation of `visitPartial` will take a `Context`, and the complete part of the tree that would be in the place of the context should it be rebuilt (`b`), plus a value from the traversal up to the "hole" in the context (the `t`). It also carries a `recurse` function as in `visitStep`. As with `visitStep` the intention (in the rendering example) is that the implementation of `visitPartial` will use `recurse` to generate the text of it's children, but use the provided `t` for the text of it's "hole". The reason for this `t` being passed separately is that it's value will have passed through a `cursor` function (since the path of `Context` constructors will end in the value pointed at by the cursor, and the result of the `visitStep` call to that value will be wrapped in a call to `cursor`), whereas requesting the value from `recurse` will not include this `cursor` wrapping.

***cursor*** An implementation of `cursor` will need to be non-strict in it's first argument (it is present to ensure that type-class resolution of uses of the method works properly and is always passed the value `undefined`), and then some result of the traversal. For the rendering example, it is the intention that `cursor` modifies the result to mark that the cursor is pointing here.

In order to concretize how an implementation of this type class will look, we will give part of one for the LAM language. However, we would not expect our user (the LAM implementor) to deal directly with implementing `Traversal`, so we provide a Template Haskell script to generate a class of adapters to make the interface simpler. What we expect the LAM implementer to implement is a set of type classes that specify how to combine values for each of the constructor cases, and how to deal with the cursor position.

$$
\begin{array}{l}
\textbf{class}\ \mathit{LamTraversalAdapterExp}\ t\ \textbf{where}\\
\quad \mathit{visitAbs} :: \mathit{Exp} \rightarrow t \rightarrow t \rightarrow t\\
\quad \mathit{visitApp} :: \mathit{Exp} \rightarrow t \rightarrow t \rightarrow t\\
\quad \mathit{visitVar} :: \mathit{Exp} \rightarrow t\\
\textbf{class}\ \mathit{LamTraversalAdapterLam}\ t\ \textbf{where}\\
\quad \mathit{visitLam} :: \mathit{Lam} \rightarrow t \rightarrow t\\
\textbf{class}\ \mathit{LamTraversalAdapterType}\ t\ \textbf{where}\\
\quad \mathit{visitUnit} :: \mathit{Type} \rightarrow t\\
\quad \mathit{visitArr} :: \mathit{Type} \rightarrow t \rightarrow t \rightarrow t\\
\textbf{class}\ \mathit{LamTraversalAdapterCursor}\ t\ \textbf{where}\\
\quad \mathit{visitCursor} :: \mathit{Lam} \rightarrow t \rightarrow t
\end{array}
$$

For example, the `visitAbs` function is passed an `Exp` (which is guaranteed by our implementation to be an `Abs` constructor) and the results of recursively traversing (with binding hooks correctly called) the traversable components in the `Abs`. It is then expected to produce a new result, presumebly by combining the recursive results together.

We automatically generate an instance of `Traversal` for the user, given an instance of these four classes:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{LamTraversalAdapterLam}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterExp}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterType}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterCursor}\ t,\\
\qquad\qquad \mathit{BoundLanguage}\ \mathit{Lam}\ t) \Rightarrow \mathit{Traversal}\ \mathit{Lam}\ t\\
\textbf{where}\\
\quad \mathit{visitStep}\ \mathit{it}\ \mathit{recurse} = \textbf{case}\ \mathit{reify}\ \mathit{it}\ \textbf{of}\\
\qquad \mathit{TW}\ x \rightarrow \mathit{visitStep'}\ x\ \mathit{it}\ \mathit{recurse}\\
\quad \mathit{visitPartial}\ (\mathit{CW}\ \mathit{ctx}) = \mathit{visitPartial'}\ \mathit{ctx}\\
\quad \mathit{cursor} = \mathit{visitCursor}
\end{array}
$$

`visitStep'` case dispatches on the `TypeRepI` it is passed, and the underlying value to call the appropriate adapter function, with the recurse calls already made:

$$
\begin{array}{l}
\mathit{visitStep'} :: (\mathit{LamTraversalAdapterExp}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterLam}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterType}\ t) \Rightarrow\\
\quad \mathit{TypeRepI}\ a \rightarrow a \rightarrow\\
\quad (\forall\ b.\mathit{Reify}\ \mathit{Lam}\ b \Rightarrow \mathit{Movement}\ \mathit{Lam}\ \mathit{Down}\ a\ b \rightarrow t) \rightarrow\\
\quad t\\
\mathit{visitStep'}\ \mathit{ExpT}\ \mathit{it}\ \mathit{recurse} = \textbf{case}\ \mathit{it}\ \textbf{of}\\
\quad \mathit{Abs}\ \_\ \_\ \_ \rightarrow \mathit{visitAbs}\ \mathit{it}\ (\mathit{recurse}\ (\mathit{MW}\ \mathit{MAbsToType}))\\
\qquad (\mathit{recurse}\ (\mathit{MW}\ \mathit{MAbsToExp}))\\
\quad \mathit{App}\ \_\ \_ \rightarrow \mathit{visitApp}\ \mathit{it}\ (\mathit{recurse}\ (\mathit{MW}\ \mathit{MAppToExp0}))\\
\qquad (\mathit{recurse}\ (\mathit{MW}\ \mathit{MAppToExp1}))\\
\quad \mathit{Var}\ \_ \rightarrow \mathit{visitVar}\ \mathit{it}\\
\mathit{visitStep'}\ \mathit{LamT}\ \mathit{it}\ \mathit{recurse} = \textbf{case}\ \mathit{it}\ \textbf{of}\\
\quad \mathit{Lam}\ \_ \rightarrow \mathit{visitLam}\ \mathit{it}\ (\mathit{recurse}\ (\mathit{MW}\ \mathit{MLamToExp}))\\
\mathit{visitStep'}\ \mathit{TypeT}\ \mathit{it}\ \mathit{recurse} = \textbf{case}\ \mathit{it}\ \textbf{of}\\
\quad \mathit{Unit} \rightarrow \mathit{visitUnit}\ \mathit{it}\\
\quad \mathit{Arr}\ \_\ \_ \rightarrow \mathit{visitArr}\ \mathit{it}\ (\mathit{recurse}\ (\mathit{MW}\ \mathit{MArrToType0}))\\
\qquad (\mathit{recurse}\ (\mathit{MW}\ \mathit{MArrToType1}))
\end{array}
$$

`visitPartial'` instead case dispatches on the passed `ContextI`, and again calls the correct adapter function, but uses the `hole` value instead of `recurse` for the value from the context's "hole".

$$
\begin{array}{l}
\mathit{visitPartial'} :: (\mathit{LamTraversalAdapterLam}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterExp}\ t,\\
\qquad\qquad \mathit{LamTraversalAdapterType}\ t) \Rightarrow\\
\quad \mathit{ContextI}\ a\ b \rightarrow b \rightarrow t \rightarrow\\
\quad (\forall\ c \circ \mathit{Reify}\ \mathit{Lam}\ c \Rightarrow \mathit{Movement}\ \mathit{Lam}\ \mathit{Down}\ b\ c \rightarrow t) \rightarrow\\
\quad t\\
\mathit{visitPartial'}\ \mathit{ctx}\ \mathit{it}\ \mathit{hole}\ \mathit{recurse} = \textbf{case}\ \mathit{ctx}\ \textbf{of}\\
\quad \mathit{TypeToAbs}\ \_\ \_ \rightarrow\\
\qquad \mathit{visitAbs}\ \mathit{it}\ \mathit{hole}\ (\mathit{recurse}\ (\mathit{MW}\ \mathit{MAbsToExp}))\\
\quad \mathit{ExpToAbs}\ \_\ \_ \rightarrow
\end{array}
$$

```
    visitAbs it (recurse (MW MAbsToType)) hole
  ExpToApp0 _ →
    visitApp it hole (recurse (MW MAppToExp1))
  ExpToApp1 _ →
    visitApp it (recurse (MW MAppToExp0)) hole
  ExpToLam → visitLam it hole
  TypeToArr0 _ →
    visitArr it hole (recurse (MW MArrToType1))
  TypeToArr1 _ →
    visitArr it (recurse (MW MArrToType0)) hole
```

We now have a lot of machinery, and can now consider how it is helpful. Our aim is to use it to implement a complete traversal function over the entire tree given by a cursor.

```
completeTraversal :: ∀ l t x a.(Traversal l t) ⇒
  Cursor l x a → t
```

The implementation of this function will require two local helper functions: `hook`, which is the template for the `recurse` function passed to `Traversal` implementers; and `foldUp`, which serves a similar role to the previous `foldUp`.

`hook` forms the backbone of the downward traversal into values that arn't on the path of contexts between the cursors' focal item and the root. Using `unbuildOne` it unpeels a value one level, using the `Context` from `unbuildOne` as a parameter to `bindingHook`, and then uses `visitStep` passing it the "hole" value from `unbuildOne` as the item to visit, with `hook` curried with the "hole" value as the function for `recurse`.

```
hook :: ∀ l t a b.(Traversal l t, Reify l b) ⇒
                a → Movement l Down a b → t
hook here movement
  = case unbuildOne movement here of
  Just (ctx, b) →
    let hook' :: ∀ c.Reify l c ⇒
                Movement l Down b c → t
        hook' = hook b
    in  bindingHook ctx (visitStep b hook')
  Nothing → error "Bad movement in traversal!"
```

Note that in `hook`, if the `unbuildOne` call fails there is nothing we can do and therefore call `error` to bail out. However, if the user is using our generated adapter classes, that case is impossible to reach.

`foldUp` walks up the path of contexts from the focal point to the root, using `bindingHook` to propogate binding information, and `visitPartial` (with a curried `hook`) to build up the result value from the traversal. Since we also need the full constructor values from the language, we use `buildOne` to rebuild the tree as we go up too.

```
foldUp :: (Traversal l t) ⇒
  t → a → Path l (Context l) a b → t
foldUp t _ Stop = t
foldUp t here (Step ctx nxt)
  = foldUp (bindingHook ctx
                      (visitPartial ctx next t
                                  (hook next)))
          next nxt
  where
    next = buildOne ctx here
```

A complete traversal then uses `visitStep` to calculate the result from the current focus point, wraps that value in `cursor` to mark the cursor's location, and uses `foldUp` to build the remaining results for the rest of the tree.

```
completeTraversal (Cursor it ctx _)
  = foldUp (cursor (⊥ :: l) (visitStep it hook')) it ctx
  where
    hook' :: ∀ b.Reify l b ⇒ Movement l Down a b → t
    hook' = hook it
```

We will now finish our example, showing how to render LAM code.

First we will need an API for composing text. Because rendering LAM variables will require binding information, we base the type we do rendering under on our `LamBindingImpl` and `LamBinder` API.

Text rendering computations will take place under a `TextRenderer` type. We pair the result of the computation with a `String` that represents the text thus-far rendered.

```
newtype TextRenderer x
  = TR{fromTR :: (LamBinderImpl (String, x))}
```

We can make `TextRenderer` an instance of `Functor`, `Applicative`, `Monad`, and `LamBinder`. We can then use the combinators and methods these type classes make available in our rendering code.

```
instance Functor TextRenderer where
  fmap f (TR x) = TR (fmap (fmap f) $ x)
instance Applicative TextRenderer where
  (TR f) <*> (TR l)
     = TR (fmap (uncurry (<*>)) (liftA2 (,) f l))
  pure v = TR (pure (pure v))
instance Monad TextRenderer where
  return = pure
  (TR f) >>= fn = TR (f >>= fromTR ∘ fn ∘ snd)
instance LamBinder TextRenderer where
  addBinding s (TR x) = TR (addBinding s x)
  lookupBinding i = TR (fmap pure $ lookupBinding i)
  getBindingMap = TR (fmap pure $ getBindingMap)
```

We take advantage of the `Monoid a => Applicative ((,) a)` instance provided in the `Control.Applicative` library to deal with creating initial empty `String`s of rendered text, and concatenating `String`s together. This means appending a string in our library is just a case of making it the first value in the returned tuple.

```
string :: String → TextRenderer ()
string s = TR (pure (s, ()))
space :: TextRenderer ()
space = string " "
```

We can now give the rendering code for all of the constructors. We use an `Applicative` style of programming, where `*>` appends results together.

```
instance LamTraversalAdapterLam (TextRenderer ())
where
  visitLam _ hole = hole
instance LamTraversalAdapterExp (TextRenderer ())
where
  visitAbs (Abs name _ _) ty exp
     = string ("λ " ++ name ++ "::") *>
       ty *> space *> exp
  visitApp _ l r = l *> string "∘" *> r
  visitVar (Var i)
     = (string ∘
         fromMaybe "Free Variable!" =<< lookupBinding i) *>
       (string ∘ subscript $ i)
```

```
instance LamTraversalAdapterType (TextRenderer ())
where
  visitUnit _ = string "τ"
  visitArr _ lhs rhs = lhs ⊛> string " → " ⊛> rhs
instance LamTraversalAdapterCursor (TextRenderer ())
where
  visitCursor _ child = string ">" ⊛> child ⊛> string "<"
```

`subscript` is a small helper that turns a number into a unicode string of subscripted numbers.

```
subscript :: Integer → String
subscript = map (chr ∘ (+) (8320 − (ord '0')) ∘ ord) ∘
            show
```

So to actually get the rendered text of the cursor, we perform a complete traversal, then unwrap the `TextRenderer` constructor from the result, then unwrap the `LamBinderImpl`, pass in an empty map to the now exposed function, and then extract the `String` result from the resulting tuple:

```
render :: Cursor Lam x a → String
render = (λ(s, ()) → s) ∘ ($Map.empty) ∘
         fromLBI ∘ fromTR ∘ completeTraversal
```

We can then hook all this code togehter into a UI, and, recalling our example LAM program from the start of the paper, present a small navigating cursor example using it: (The full UI example program is included in our library).

$> \lambda\, x :: \tau\ \rightarrow\ \tau\, x_0 \circ \lambda\, y :: \tau\, y_0 \circ x_1 <$
Move down twice (from `Lam` to `Abs` and then `Abs` to `Arr`).
$\lambda\, x :: > \tau\ \rightarrow\ \tau <\ x_0 \circ \lambda\, y :: \tau\, y_0 \circ x_1$
Move right
$\lambda\, x :: \tau\ \rightarrow\ \tau\ > x_0 \circ \lambda\, y :: \tau\, y_0 \circ x_1 <$
Move down
$\lambda\, x :: \tau\ \rightarrow\ \tau\ > x_0 < \circ \lambda\, y :: \tau\, y_0 \circ x_1$
Move right
$\lambda\, x :: \tau\ \rightarrow\ \tau\, x_0 \circ > \lambda\, y :: \tau\, y_0 \circ x_1 <$
etc.

## 3. Related Work

There has been a lot of noise about zipper data structures in the Haskell community recently. Practical, popular applications (5) and general libraries (4) are emerging based on the underlying ideas of the original paper (1). Like our library, these examples take the general principles of contexts and a focal point, and tailor them to specific domains (managing stacks of windows for a window manager, or providing a usable interface for editing a large number of related items, with the option of changing your mind). However, we have taken our specific application (a structured editor for $F_C$), and generalised out a reusable library, and provided automated generation of all the boilerplate.

There are existing reusable, zipper-based libraries in the literature. In (3) the authors consider a data structure that is parametric over the type being traversed, and requires much less boilerplate to implement. However their library does not consider traversals over a heterogeneous data type and there does not appear to be a succinct extension to the work that would allow such a traversal.

In (2) the author presents an elegant GADT based zipper library that is able to traverse across heterogenous data types, and requires no boilerplate to use. However we believe that it is not a practically useful library without some additional boilerplate being written; the implementation requires that at all use sites a lot of type information is available to allow up/left/right movements, and down movements require the precise type of what is being moved into to be available. In an application that is interactively allowing a user to update the cursors position, it would require a complicated existential context with type classes or type witnessess being present to allow these movements to happen. With our library, we provide both the type specific movments, but have also provided the additional boilerplate needed to recover the generic movements that can move a cursor without any additional type constraints being present.

An alternate approach to the cursor library was explored in (6). Here, the zipper library is parameterised by a traversal function and uses delimited continuations to move around the tree. The authors also show how to support a statically known number of subcursors, allowing something like our route/bookmark functions. They however, are working in the context of filesystems and do not need to consider lexically bound information in the interface they present.

## 4. Conclusion

### 4.1 Future work

Unsurprisingly, there is always more functionality we could add to our library. Some particular extensions we wish to explore include persisting our cursors so the user's context of work can be saved and restored. We have also only looked so far at simple languages, we have not considered cursors for languages that are themselves parameterised by types, or languages with GADTs in them, both of these could present interesting challenges.

Furthermore, the zipper datastructure was originally designed around the idea of needing to perform local updates and edits, and not necessarily global traversals; while we justify this by arguing that in an editor context many local edits and changes may take place between the global renders; we should perform some performance and complexity analysis of our global traversals against some alternative schemes.

There are some other issues; we are using some experimental features of GHC (e.g. type families), which are not completely implemented yet - when a complete implementation is released we can neaten our library by, for example, not requiring the thin wrappers on the `Context` type implementation. Currently (to our knowledge) Template Haskell does not support the generation of GADTs or type family instances and so our generation scripts output the source code for compilation to new files; this is an ugly indirection step that we would like to avoid in future.

### 4.2 Summary

We have outlined a cursor library based on ideas from Huet's original paper, but using GADTs to allow navigation around a heterogeneous data type. We have abstracted away from a ground example language and made a parameterisable library; this means that tools can be developed which depend only on the library and will be re-usable for any language.

The code presented has been split into three parts, that which the user provides, that which forms a generic library, and that which we automatically generate using Template Haskell. At no point has the user been required to implement any boiler-plate code themselves.

We have shown how it is possible to encode routes in the tree from the cursor that could be used for bookmarks in an application, and shown how to use the context representation that we automatically generate to allow the user to neatly express lexical binding rules. Finally, we gave an example usage of the library, where the user could render the tree with the location of the cursor and binding information. To do this we implemented a generic traversal API and provided automatically generated adapters that make the users interface to the traversal API idiomatic.

## 5. References


## References

[1] Huet, G. The zipper. Journal of Functional Programming, 7(5):549-554, 1997

[2] Adams, M. Functional Pearl: Scrap Your Zippers. Unpublished, 2007

[3] Hinze, R. and Jeuring, J. Functional Pearl: Weaving a Web in J. Functional Programming, 11(6):681-689, November 2001.

[4] Yorgey, B. zipedit library (Online), 2008, http://byorgey.wordpress.com/2008/06/21/zipedit/

[5] Stewart, D. Roll Your Own Window Manager: Tracking Focus with a Zipper (Online), 2007, http://cgi.cse.unsw.edu.au/ dons/blog/2007/05/17

[6] Kiselyov, O. Tool demonstration: A zipper based file/operating system. In Haskell Workshop. ACM Press, September 2005

[7] Sulzmann, M. and Chakravarty, M. M. T. and Jones, S. P. and Donnelly, K. System F with Type Equality Coercions, in The Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07), January 2007.

[8] Allwood, T. Clase library download and screenshots, (Online), 2008, http://www.zonetora.co.uk/NonBlog/toral/lib/.

[9] de Bruijn, N. G. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem, in Indagationes Mathematicae (34) 381–392, 1972