Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 253 (2010) 3–18
www.elsevier.com/locate/entcs

Strengthening the Zipper

Tristan O.R. Allwood[1] Susan Eisenbach[2]

Department of Computing
Imperial College
London
United Kingdom

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 its potential applications. In this paper we take inspiration from Huet, and describe a library suitable as an underpinning for structured editors. 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.

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

1 Introduction

We have recently found ourselves implementing an interactive tool for visualising and manipulating an extended λ-calculus, F_C [7], which is the current, explicitly typed, intermediate language used in GHC. Our editor allows the user to view an entire F_C term (e.g., a Haskell function translated to F_C). The user has a cursor, which indicates the current subterm of the entire term that has the focus. The user can then apply operations to the current subterm, such as applying a β-reduction simplification or unfolding a global definition at the first available place within the subterm.

The tool provides a view of the variables that are currently in scope at the cursor location and this information is also provided to the functions that operate on the currently focused subterm. The tool provides several views on the internal data structure, and the user has ways of manipulating it with the output from what is rendered is just a view.

[1] Email: tristan.allwood@imperial.ac.uk
[2] Email: susan.eisenbach@imperial.ac.uk

doi:10.1016/j.entcs.2010.08.028

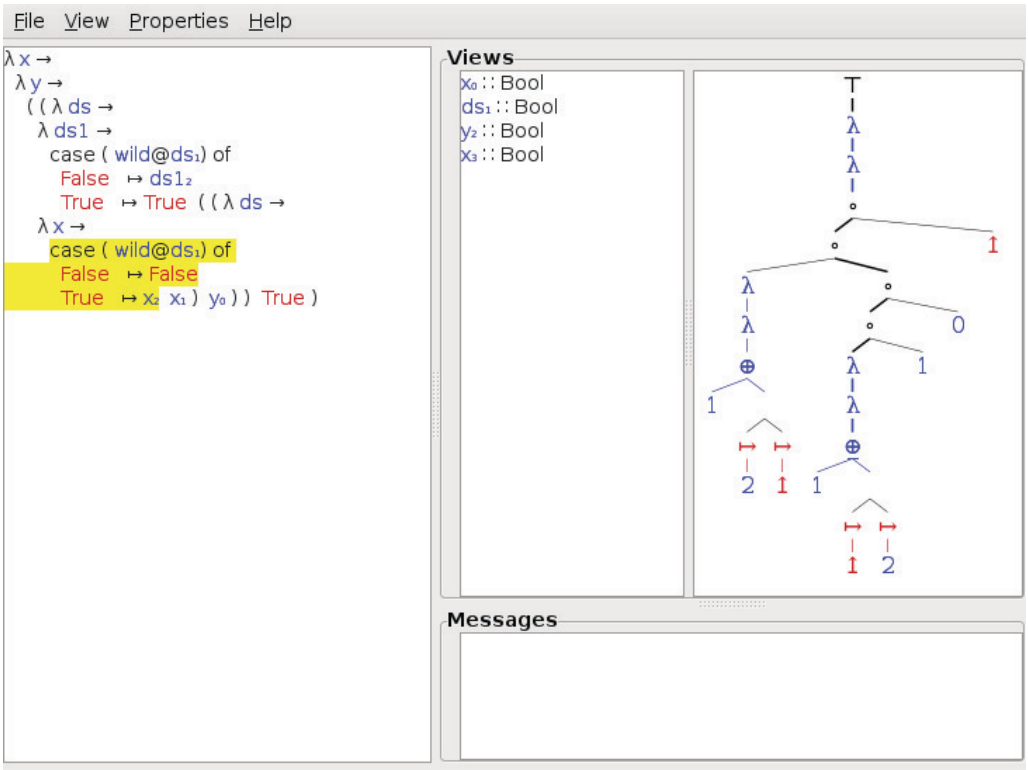

Fig. 1. Our Interactice Core editor.

We wanted to support *bookmarks* i.e., the ability to mark a subterm as *bookmarked* and then later to return the cursor to that location. However, if the user makes a change to the term by manipulating a subterm, we want to be able to detect which bookmarks may have become invalidated by that change, and fix them in some way.

During implementation we developed an internal data structure based around a design pattern similar in many ways to the classic *zipper* [1]. However we had to address several issues that the original paper doesn't cover, and that we couldn't find in the literature. These issues relate to traversal in order to render the different UI views and where complicated by the fact that our underlying term language was made up of several (possibly) mutually recursive data types.

Borne out of our experiences in implementing this tool, we have developed a library, CLASE, that can provide a cursor data structure over a set of data types, and an interface for moving the focus, as-well as traversing the cursor to perform actions such as rendering.

The rest of this paper is organised as follows: We begin in Section 2 by sketching the underlying design of our *Cursor* with a simple example, motivating our use of Generalized Algebraic Data Types (or GADTs, [11]) to allow us to create cursors that can traverse heterogeneous data types. We then introduce CLASE, the cursors it uses and how a user can move a cursor about in Section 3. In Section 4 we motivate some duplication and complexity that can occur when attempting to render a cursor

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

Fig. 2. The Lam Language

(to e.g., a *String*) and outline how CLASE makes this easier and more idiomatic for a user to do. Another feature of CLASE is its support for bookmarks into the cursor, which are explained int Section 5. Finally in Section 6 we outline related work and conclude.

2 Contexts and simple Cursors

To demonstrate our techniques we have created small language Lam presented in Figure 2. The *Lam* type marks the root of our program, and its sole constructor, *Root*, is a simple wrapper over a Lam expression. [3]

Expressions are either lambda abstractions, *Abs*, which carry a *String* name for their variable, a *Type* for their variable and an expression 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*.

The following Lam program represents the term $\lambda x :: \tau \to \tau.(x \circ \lambda y :: \tau.(y \circ x))$:

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

A simple *Cursor* is a pair of the value (i.e., subterm) currently in focus (sometimes referred to hereafter as *it*), and some *context*, which will allow reconstruction of the entire term. Our cursors are analogous to the zipper design pattern, allowing O(1) movement up or down the tree. In the down direction there are ways to choose which child should be visited.

In Figure 3 we visualise a cursor moving over the Lam value:

```
Root ((Abs "x" (Unit `Arr` Unit) (Var 0)) `App` (Var 1))
```

[3] Lam refers to the "language", which is defined by the *Lam*, *Exp* and *Type* types.

The cursor in (a) starts focused at the *Root* constructor. Since this constructor is "at the top" of the term's structure, the context is empty. We then in (b) move the cursor down, so the focus is on the *App* constructor, in doing so we add a *Context constructor* (here *ExpToRoot*) to the front of the context. The context constructors explicitly witness the change needed to be applied to the current focus in-order to rebuild one layer of the tree. Moving the focus onto the *Abs* constructor on the left hand side of the application in (c) pushes a new constructor onto the context. *ExpToApp0* both indicates the focus is in the first *Exp* child of an *App* node, and carries the values that where the right-hand-children of the *App* node (*Var* 1). Moving down once more in (d) puts the focus on the *Arr* constructor inside the *Type* child of the *Abs* node, and again pushes an appropriate context constructor to be able to rebuild the *Abs* node should the user wish to move up.

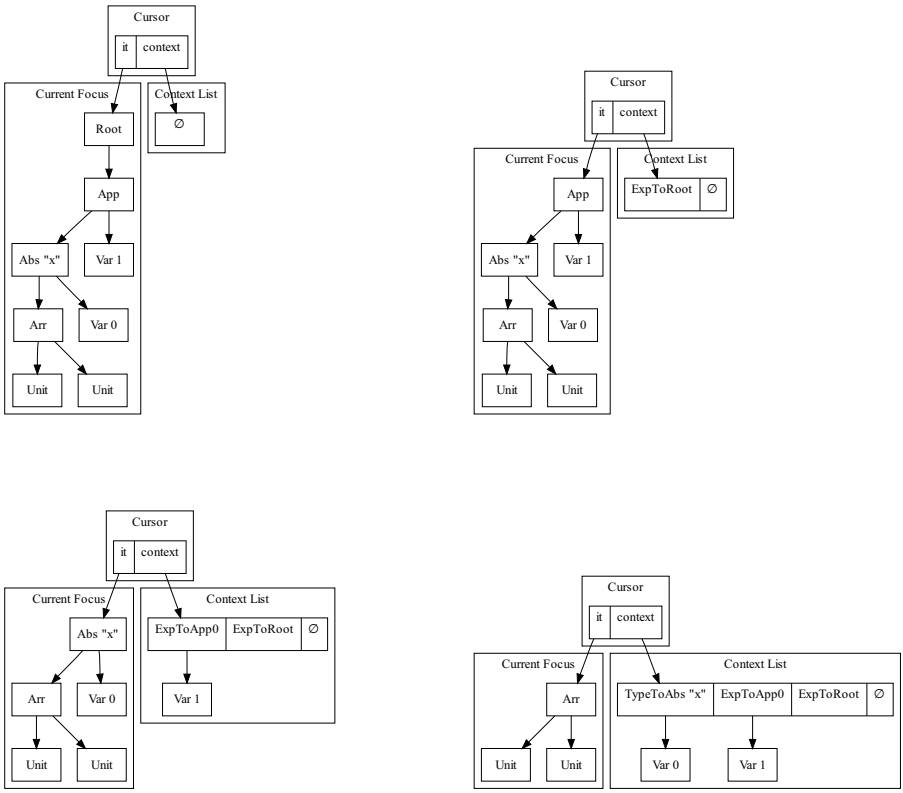

Fig. 3. Cursor structure changes due to moving the focus

In order to be able to move around easily, given our original data types (*Lam*, *Exp* and *Type*), we need context constructors to represent all of the possible contexts for our cursor. We use GADTs to push into the type system the types that our context constructors expect for their "missing value", and the type of the constructor they ultimately represent. We will later use this extra information to help maintain some general invariants and as a sanity check that our implementation of a cursor is

correct.

```
data Context from to where
  ExpToRoot :: Context Exp Lam
  ExpToApp0 :: Exp → Context Exp Exp
  TypeToAbs :: String → Exp → Context Type Exp
  ...
```

As we saw in the diagram, when we move around our term, we build up a stack of *Context*s. If our contexts were ordinary data types we could use a list, however we need to ensure that the *to* parameter of our first *Context* matches up with the *from* parameter of the next *Context*. To do this we use a new data type called *Path*.

```
data Path ctr from to where
  Stop :: Path ctr anywhere anywhere
  Step :: (..) ⇒ ctr from middle → Path ctr middle to → Path 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 its type at a later time, and hence the (for the moment unspecified) class constraint.

Our basic cursor structure is then simply:

```
data Cursor here = Cursor{
  it :: here,
  context :: Path Context here Lam
}
```

The current point of focus is denoted by *it*, and the *context* we are in is a path from *here* up to the root of our language, *Lam*. The cursor data structure in our library CLASE extends this data type in two useful ways.

3 CLASE

We now present our library, CLASE, which is a *Cursor Library for A Structured Editor.*[4] As shown in Figure 4, a typical use of CLASE consists of three parts. There is the code the developer has to write, typically the data types that express the language to be structurally edited (e.g., the LAM datatypes), a couple of specific type classes useful for rendering as discussed in Section 4, and of course, the application that uses CLASE itself. CLASE then provides some Template Haskell scripts that automatically generate some boilerplate code, and an API of general functions for building, moving and using CLASE cursors.

[4] It is available for download with documentation from [8].

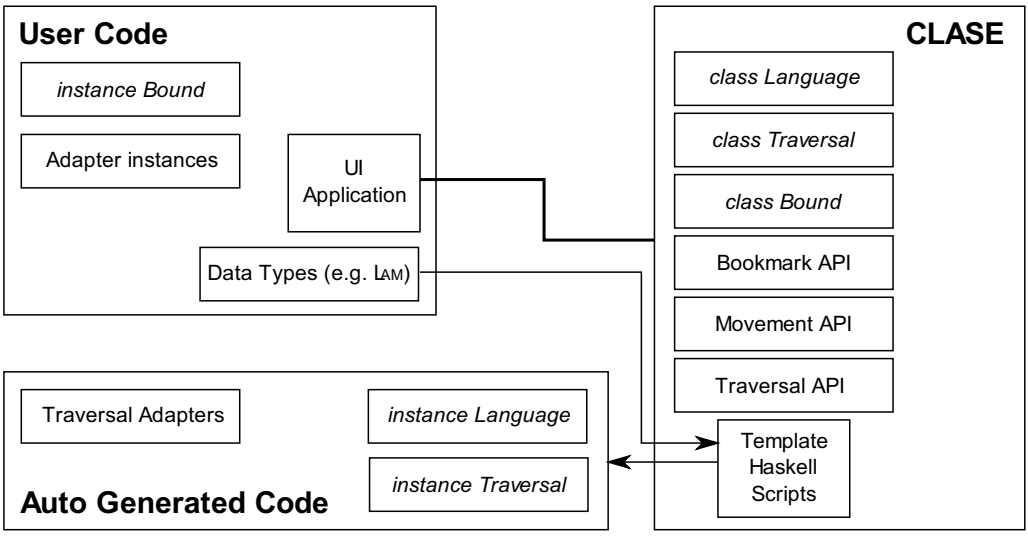

Fig. 4. An overview of using CLASE

CLASE requires the user to implement a typeclass called *Language*, shown in Figure 5. The instance should be the type that is at the root. So for our Lam example, we will make an instance *Language Lam*.

A *Language* needs to have data types (expressed using associated data type families [12]) corresponding to its *Context*s, primitive *Movement*s and a way of reflecting on the different types in the language (*TypeRep*). The *Context Lam* is as shown earlier. Primitive *Movement*s are GADT constructors that witness all of the one-step movements *Up* or *Down* that the cursor could do. For example, for Lam they would be:

```
data Movement d a b where
  MUp :: Movement Down b a → Movement Up a b
  MRootToExp :: Movement Down Lam Exp
  MAppToExp0 :: Movement Down Exp Exp
  MAbsToType :: Movement Down Exp Type
  ...
```

The way of reflecting on the different types in the language is provided by a simple GADT:

```
data TypeRep a where
  ExpT  :: TypeRep Exp
  LamT :: TypeRep Lam
  TypeT :: TypeRep Type
```

This is then linked by the user to the type class *Reify*, which provides a sometimes more convenient way of passing around the witnesses. *ExistsR* is a data type that provides an existential wrapper for something that is parameterised by some type in the 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
  movementEq :: Movement l d a b → Movement l d a c → Maybe (TyEq b c)
  reifyDirection :: Movement l d a b → DirectionT d
  contextToMovement :: Context l a b → Movement l Up a b
  downMoves :: TypeRep l a → [ExistsR l (Movement l Down a)]
  moveLeft :: Movement l Down a x → Maybe (ExistsR l (Movement l Down a))
  moveRight :: Movement l Down a x → Maybe (ExistsR l (Movement l Down a))
class Reify l a where
  reify :: a → TypeRep l a
data ExistsR l (r :: * → *) where
  ExistsR :: (Reify l a) ⇒ r a → ExistsR l r
data Up
data Down
type family Invert d :: *
type instance Invert Up = Down
type instance Invert Down = Up
data DirectionT a where
  UpT :: DirectionT Up
  DownT :: DirectionT Down
data TyEq a b where
  Eq :: TyEq a a
```

Fig. 5. The *Language* typeclass and supporting data structures and types

With these data types for the language declared, the user then has to provide some primitive functions that rebuild values given a *Context* and a value for it's "hole" (*buildOne*), or take apart a value given an indication of which child we want to become the "hole" (*unbuildOne*).

We also require that all movements are invertible (*invertMovement*), and can be tested for equality (*movementEq*). Movement equality needs to provide a type equality witness in the case that the two movements are the same. All *Movement*s can only move up or down, which is what *reifyDirection* requires, and all *Context*s must correspond to an upward movement (*contextToMovement*).

Finally, in order to provide more generic move up/down/left/right operations that don't need an explicit witness, the user needs to enumerate all the possible down movements starting from a particular type (*downMoves*) and finally how to

logically shift left or right a downward movement (*moveLeft* and *moveRight*). For example *moveLeft MAbsToExp* = *Just* (*ExistsR MAbsToType*)

For simple languages, such as our LAM language, the instance of *Language* is straightforward and easily mechanically derivable. For such languages we provide a Template Haskell function that can automatically generate a module containing the *Language Lam* instance. As the current state of Template Haskell precludes the inline generation of GADTs and Associated Data Types, our generation function outputs the module as a source file that can be imported later.

The Template Haskell script provided with CLASE requires the user to specify the module to create the instance of *Language* in, the root data type (*Lam*), and the types that the cursor should be able to navigate between (*Lam*, *Exp* and *Type*). The library user invokes the script using a splice, $(...), and refers to the root and navigable types using TH quasiquotes (''), as shown:

```
$ (languageGen ["Lam", "Language"]
               '' Lam
               ['' Lam, '' Exp, '' Type])
```

The *languageGen* function works by using TH *reify* to access the shape of *Lam*, *Exp* and *Type* data types, and then processes that to work out the simple *Context*s, and from there simple *Movement*s and the implementations of the primitive functions such as *buildOne* and *unbuildOne*.

With a suitable instantiation of *Language*, CLASE then provides the library functions and data types for representing and manipulating cursors. The core *Cursor* data structure is an extended version of the *Cursor* seen previously. It is now parameterised by three types; l is the same type used to instantiate the *Language* typeclass this *Cursor* is used for, x is used for bookmark like behaviour, and will be discussed in Section 5, and a is the type of the current focus. *it* and *ctx* are the current focus and context as before, and for bookmark behaviour we also add a *log* field, that again will be discussed in Section 5.

```
data Cursor l x a = (Reify l a) ⇒ Cursor{
  it  :: a,
  ctx :: Path l (Context l) a l,
  log :: Route l a x
}
```

CLASE provides two ways of moving the focus of a cursor around. The first is a way of applying the specific movements described in the *Language* instance. This is through a function *applyMovement*:

```
applyMovement :: (Language l, Reify l a, Reify l b) ⇒
  Movement l d a b → Cursor l x a → Maybe (Cursor l x b)
```

so given a *Movement* going from a to b (in either direction), and a *Cursor* focused on an a, you will get back a *Cursor* focused on a b. This is a very useful function if

a GUI wants to apply a set of movements around a known tree structure. However it does require knowing up-front the type of the *Cursor*'s current focus and that you have a *Movement* which matches it.

CLASE also provides a set of generalized movement operators. These do not need the calling context to know anything about the *Cursor* they are being applied to. There are four ways of generically moving around a tree, up, depth-first down, or moving left/right to the adjacent sibling of where you currently are. Since it is unknown what type the focus of the *Cursor* will be after applying one of these operations, they return *Cursor*s with the type of *it* existentially quantified.

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

In the case of *genericMoveUp / Down* we also return the *Movement* constructor that could have been used via an *applyMovement* call to achieve the same effect, should an application find that useful.

4 Rendering and binding

One of the things you want to do with a structured editor is to display the contents, indicating where the current focus is. Assuming you wish to provide a textual rendering, given one of our *Cursor*s, this would require you to:

(i) Convert the value of the focus (i.e., a *Lam*, *Exp* or *Type*) to a *String*.

(ii) Modify the *String* value from rendering the focus to indicate the current focus (e.g., by wrapping it in marking brackets ala `"> ... <"`).

(iii) Render all of the context constructors, passing in the *String* from rendering what is below that constructor as the *String* to use as the value for the "hole" in the constructor.

In Figure 6 we outline part of our first attempt at just rendering expressions and contexts, ignoring the control flow needed to fold context results into each other. Since the Lam language has bound variables, and we wish to render their names in variable position, we perform the computation in some fictional monad M and use its API function *addBinding* to make a binding for a new variable available in a sub-computation.

```
renderExp :: Exp → M String
renderExp (Abs str ty exp) = do
  tys ← renderType typ
  rhs ← addBinding str (renderExp exp)
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
renderCtx :: Context Lam from to → M String → M String
renderCtx (TypeToAbs str exp) rec = do
  tys ← rec
  rhs ← addBinding str (renderExp exp)
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
renderCtx (ExpToAbs str ty) rec = do
  tys ← renderType ty
  rhs ← addBinding str rec
  return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
addBinding :: String → M a → M a
...
```

Fig. 6. Parts of a first attempt at rendering a Lam cursor

As is clear in the example, the code is highly repetitious, and the logic for when to call *addBinding* is intermingled with the code for rendering and traversing. In practice we have found this binding code hard to manage (particularly if the language grows to more complexity), and the duplication highly undesirable.

In CLASE we solve this problem by completely factoring away the binding code, and then provide adapters to make writing the rendering code more factored. CLASE has a typeclass *Bound* which allows a user to use the context-constructors generated for their language to express binding constraints. The intuition is that when a traversal moves down through the tree into the "hole" described by the constructor, the user can alter the result of the traversal value.

```
class (Language l) ⇒ Bound l t where
  bindingHook :: Context l from to → t → t
```

For the Lam language, an appropriate instance for our rendering code above would be to wrap the *hole* value in an *addBinding* application whenever we move through an *Abs* from it's *Exp* child, and in all other cases, to leave it unchanged.

```
instance Bound Lam (M a) where
  bindingHook (ExpToAbs str _) hole = addBinding str hole
  bindingHook _ hole = hole
```

CLASE also provides a combinator for doing render-like traversals over a *Cursor*. It is a library function with the signature:

$$completeTraversal :: \forall\ l\ t\ x\ a.(Traversal\ l\ t) \Rightarrow Cursor\ l\ x\ a \rightarrow t$$

The *Traversal* type-class contains functions for performing the three actions itemized above, and the library contains the glue-code to make the traversal work. However instead of requiring the user to implement this type-class directly, CLASE features another Template Haskell script to automatically create an instance of *Traversal* given instances of some adapters. Our TH script (*adapterGen*) generates the follow code for LAM.

```
class LamTraversalAdapterExp t where
  visitAbs :: Exp → t → t → t
  visitApp :: Exp → t → t → t
  visitVar :: Exp → t
class LamTraversalAdapterLam t where
  visitLam :: Lam → t → t
class LamTraversalAdapterType t where
  visitUnit :: Type → t
  visitArr :: Type → t → t → t
class LamTraversalAdapterCursor t where
  visitCursor :: Lam → t → t
instance (LamTraversalAdapterLam t,
  LamTraversalAdapterExp t,
  LamTraversalAdapterType t,
  LamTraversalAdapterCursor t,
  Bound Lam t) ⇒ Traversal Lam t where
  ...
```

As an example, the user logic for rendering an abstraction constructor is now restricted to just the instance of the *visitAbs* function in *LamTraversalAdapter*:

```
instance LamTraversalAdapterExp (M String) where
  visitAbs (Abs str _ _) ty exp = do
    tys ← ty
    exps ← exp
    return ("λ " ++ str ++ " : " ++ tys ++ " . " ++ exps)
  ...
```

Using the API is straightforward, given user instances of the adapters above, then converting a *Cursor* to an *M String* is simply a case of calling *completeTraversal* in the correct-type context:

```
render :: Cursor Lam x a → M String
render = completeTraversal
```

5 Routes and Bookmarks

An editor developed using CLASE 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. Using the *Path* data type from earlier, we can encode these routes in a new CLASE data type *Route*:

```
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
```

These can be made into a unique routes by disallowing the last *Up* movement to be the inverse of the first *Down* movement, i.e., the following invariant is maintained:

```
route_invariant :: (Language l) ⇒ Route l from to → Bool
route_invariant (Route (Step mup Stop) (Step mdown _))
  = (¬ ∘ isJust) (invertMovement mup `movementEq` mdown)
route_invariant (Route (Step _ ups) downs)
  = route_invariant (Route ups downs)
route_invariant (Route Stop _) = True
```

The CLASE cursor keeps track of a single *Route* 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*.

```
updateRoute :: (Language l, Reify l a, Reify l b) ⇒
  Movement l d a b → Route l a c → Route l b c
resetLog :: Cursor l x a → Cursor l a a
appendRoute :: (Language l, Reify l a,
                Reify l b, Reify l c) ⇒
  Route l a b → Route l b c → Route l a c
followRoute :: (Language l) ⇒
  Cursor l x a → Route l a c → Maybe (Cursor l x c)
```

Should a user application want to bookmark multiple different subterms, this API makes this straightforward to do, and helps ensure the application doesn't forget to update the bookmarks. The user application would have as its state a Cursor with

an empty *log*, and a map of integers to routes that lead from the cursor's current location to somewhere else. For example, a LAM GUI may use the following:

```
data CursorHolder where
  CH :: Cursor Lam a a → Map Int (ExistsR Lam (Route Lam a)) →
                         CursorHolder Lam
```

Creating a new bookmark at the current location is just a case of inserting a value of *ExistsR* (*Route Stop Stop*) into the *Map* at the bookmark's key.

When the GUI tries to move the cursor, the main loop would respond to a keypress. We proceed by unwrapping the cursor holder and applying *genericMoveDown* to the cursor.

```
keypressDown :: CursorHolder → CursorHolder
keypressDown ch@(CH cursor@Cursor{}bookmarks) = fromMaybe ch $ do
  (CWM cursor' _) ← genericMoveDown cursor
  ...
```

At this point of the code, the type of *cursor* is $\exists\ a.Cursor\ Lam\ a\ a$ and *cursor*′ is $\exists\ b.Cursor\ Lam\ a\ b$, i.e., the *log* field of *cursor*′ gives a route back to the a. If we attempted to return a new *CursorHolder* containing *cursor*′ (or *resetLog cursor*′) and the original *bookmarks* it would be a type error. The type system enforces that we update all the bookmarks to make the type parameters match up with the new cursor parameters...

```
...
let bookmarks' = Map.map (λ(ExistsR bm) →
                            ExistsR ((log cursor') `appendRoute` bm))
                         bookmarks
let cursor'' = resetLog cursor'
return $ CWM cursor'' bookmarks'
```

Jumping to a bookmark is then a case of using CLASE's *followRoute* to update the cursor's location, and then using the same logic as above to update all the bookmarks (including the one that was just followed) to the new location.

Detecting whether a change to the current focused subterm may invalidate a bookmark is also easy. A route will only point inside the current subterm if it has no up components, i.e., it has the shape *Route Stop something*.

6 Conclusions

There has been a lot of discussion 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).

One of the fundamental underpinnings of our work (and much of the related work) is that of a one-holed context. These have been discussed in [10], and provide an interesting relationship between differential mathematics and data structures. Indeed it is due to this link that we know we can automatically generate the *Context*s for simple data structures using our Template Haskell scripts.

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 heterogeneous 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 cursor's position, it would require a complicated existential context with type classes or type witnesses being present, to allow these movements to happen. With our library, we provide both the type specific movements, 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 sub-cursors, 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.

Unsurprisingly, there is always more functionality we could add to our library. We have also only looked so far at *Language* instance generation for 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 for auto-generating their *Language* instance.

Furthermore, the zipper data structure 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 will be able to neaten up some of the automatically generated instances. Also, 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.

CLASE was borne out of experience of implementing a structured editor. We now want to retrofit CLASE back into our structured editor, and in doing so hopefully find other useful traversals and features we can generalise out that may be useful in a more general setting.

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 believe that cross-datatype zippers are a useful extension to the original zipper. The provision of the generic moveUp/moveDown/moveLeft/moveRight API across these heterogeneous data type was the major difficulty.

Using CLASE makes it straightforward to render a cursor, and encode bookmarks.

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 boilerplate code themselves.

Acknowledgement

We would like to thank the many people who have commented on tool demos and talks on this work at Anglo Haskell and the Haskell Symposium. Many thanks also to the SLURP research group at Imperial, for their good ideas and useful ways to present our work.

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, 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/clase/.

[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

[10] McBride, C. The Derivative of a Regular Type Is Its Type of One-Hole Contexts, Unpublished, 2001, http://strictlypositive.org/diff.pdf

[11] Jones, S. P and Vytiniotis. D and Weirich S, and Washburn G. Simple unification-based type inference for GADTs, in ICFP, 2006

[12] Chakravarty, M. M. T. and Keller, G. and Jones, S. P. and Marlow, S. Associated types with class, In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, http://www.cse.unsw.edu.au/~chak/papers/assoc.ps.gz, 2005