CLASE: Cursor Library for A Structured Editor

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 (1) only scratches the surface of some of the potential applications of the zipper. In this work we have taken inspiration from Huet, and built 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 Design, Languages.

Introduction

Our library, CLASE (3) has been borne out of our own experiences trying to write a structured editor for F_C (2) – GHC's intermediate language. As manipulations are to be done in user-selected focused areas of the tree, we decided to use a cursor presentation indicating the current focus. Although we were drawn to a zipper-style of implementation for our underlying representation, during implementation we were finding that binding, traversal and rendering code were horribly intermingled. We developed this library to help seperate these concerns.

Usage of CLASE splits into three main parts: Firstly there is the code the user has to explicitly write, which includes the data type representing the language they wish to have a cursor for, and any language specific APIs for rendering or binding, as-well as implementing a hook function explaining when to use the binding API. The second part is the instance of the CLASE typeclass *Language*, which is generated automatically using Template Haskell (6) scripts that come with CLASE. This typeclass uses Associated Data Types (5) to represent *Context*s (constructors in the original language with a "hole" in them (7)) and primitive *Movement*s. Because the user's language involve heterogeneous data types, we generate GADT instances of *Context* and *Movement* which witness the types that they move between. Finally, CLASE provides a library of combinators and composite data structures (including the *Cursor* data structure) which use the primitives defined in the generated code to provide generalized movement, bookmarking and some traversals, which a structured editor would find useful.

```
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

Example

We proceed by giving a small demonstration of using CLASE, showing how to collect the names of in-scope variables in a cursor location. In Figure 1 we present a small language, LAM, that we wish to create a structured editor for. LAM is based on the simple λ-calculus, using deBruijn indices (4) for variables to refer to the depth of the enclosing abstraction that binds them.

When editing a LAM program, our editor needs to keep track of the current focus, and a way of moving that focus between its *Lam*, *Exp* and *Type* datatypes.

Our CLASE library provides a generic *Cursor* data structure, and operations for moving the cursor (*genericMoveUp*, *genericMoveDown*, *genericMoveLeft* and *genericMoveRight*) for any data types that implement a *Language* interface. In order to create an instance of *Language* for LAM, we use a Template Haskell script provided with CLASE, specifying the module to create our instance of *Langauge* in, our root data type (*Lam*), and the types that our 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 data structure for a cursor given in CLASE is:

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

where the type parameter l is the language over which the cursor navigates (here *Lam*), x is an arbitary point in the tree (used to

Haskell'08, September 25, 2008, Victoria, BC, Canada.
ACM 978-1-60558-064-7/08/09.

generate bookmarks), and a is the type of the current focus. it therefore is the current focus, ctx is a path of $Context$s, which are language specific and express nodes in the tree with a hole in them. This path starts at the current location (of type a) and terminates at the root of the tree (l). The log represents the shortest route from the current focus to some arbitary place in the tree.

Our $languageGen$ script will have created an instance of $Language\ Lam$, which includes the associated type $Context\ Lam$, and also the appropriate instances of $Reify\ Lam\ a$ for a user to be able to create a $Cursor\ Lam\ x\ a$.

For our editor we want to be able to show the user the entire current program highlighting the current focus. We also wish to show some derived information, for example the currently in-scope varibles. Both require the capability to traverse the program tree.

For these traversals the LAM language will need to keep track of variables as they become bound. Assuming the LAM language already has a $Monad$ that supports the following interface, suitable for tracking the names of variables as they become bound:

$$\begin{aligned}&\textbf{class}\ LamBinder\ c\ \textbf{where}\\&\quad addBinding :: String \to c\ a \to c\ a\\&\quad getBindingMap :: c\ (Map\ Integer\ String)\end{aligned}$$

We can tell CLASE how to update the binding information as it traverses the tree by implementing a library typeclass called $Bound$.

$$\begin{aligned}&\textbf{class}\ (Language\ l) \Rightarrow Bound\ l\ t\ \textbf{where}\\&\quad bindingHook :: Context\ l\ from\ to \to t \to t\end{aligned}$$

The single function $bindingHook$ takes a $Context$ (which was generated earlier) and a value that is the result of the traversal for the "hole" in the $Context$, and expects a modified value for performing any binding necessary.

In the case of LAM, whenever we move into an Abs node from its containing Exp, we need to make the binding of the current abstraction become visible. In all other cases we can just propogate up the value of the traversal:

$$\begin{aligned}&\textbf{instance}\ (LamBinder\ c) \Rightarrow Bound\ Lam\ (c\ a)\ \textbf{where}\\&\quad bindingHook\ (CW\ ctx)\ hole = bindingHook'\ ctx\\&\quad\ \textbf{where}\\&\quad\quad bindingHook' :: ContextI\ from\ to \to c\ a\\&\quad\quad bindingHook'\ (ExpToAbs\ s\ \_) = addBinding\ s\ hole\\&\quad\quad bindingHook'\ \_ = hole\end{aligned}$$

The current implementation of the code generator introduces a level of indirection to enable this code to work with GHC 6.8, making the associated type $Context\ Lam$ a constructor wrapper CW over a GADT called $ContextI$. With GHC 6.10 this indirection should be able to be removed.

CLASE currently provides two simple traversal operations that use this binding hook. The first interrogates the current focus of a $Cursor$ and then uses the ctx path to nest the appropriate $bindingHook$ calls over its value:

$$\begin{aligned}inBindingScope :: (Bound\ l\ t) \Rightarrow (a \to t) \to\\ Cursor\ l\ x\ a \to t\end{aligned}$$

Using this, we can simply get a map of the currently in-scope variables for LAM under our $Monad$:

$$varsInScope = inBindingScope\ (const\ getBindingMap)$$

The second type of traversal supported by CLASE is a complete traversal, suitable for rendering the program, while also indicating where the cursor currently is located. This traversal needs to know (in the case of LAM) how to compute values for normal (Lam, Exp and $Type$) values, as-well as how to modify this result to indicate the cursor location, and finally how to combine a partial location in to a $Context$ as the modified cursor location result needs propagating up the tree inside its enclosing $Context$s.

The code to render a $Context$ with a missing value and to render a normal value would be almost identical (except that to render a normal value an extra recursive call to render would have to be made whereas the context has it made already). In order to prevent this duplication, and to ensure that the previously defined $Bound$ information is used, CLASE provides a script to generate a set of adapters that explain how to combine values from the constructors together. Instantiating these adapters allows a very natural implementation of rendering that abstracts away from binding and traversal code.

An editor for LAM may want to keep track of multiple locations in the tree (e.g. to provide a bookmark functionality). Ideally we would like these bookmarks to be persistent across updates to the tree, and coping with invalidated bookmarks caused by changing the LAM program. Underlying CLASE is a notion of language specific primitive movements (declared in an Associated Type $Movement$ in type class $Language$). $Route$s (e.g. the log field in $Cursor$) represent a unique path from the current cursor location to some other part of the tree by tracking a path of upward movements to some location, and a path of downward movements from that intermediate place down a different branch to the destination. CLASE provides an API for creating $Route$s, appending them and incrementing them by primitive movements.

The internals of the CLASE library are designed to handle heterogeneous data types. This is achieved by using GADTs to represent the $Context$s which are parameterised by the type of the "hole" in the context, and the type of the value that can be built when a suitable value for the hole is available. Movement is handled using the primitive $Movement$ associated data types as witnesses which also express where to move to and from. The library also recovers generalized up/down/left/right operators that return an existentially wrapped cursor.

We also support persisting and restoring the $Cursor$ data structure to / from $String$s. Another Template Haskell script is used to generate some boilerplate code for this, but it simplifies what would be otherwise quite a hard and tedious task for the user.

Our library is necessarily a work in progress and incomplete. It was borne out of our own practical experience of trying to build a structured-editor like application, and as we re-integrate it back into our motivating program we hope to find further interesting extensions we can add to our library to make it more useful.

References

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

[2] 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.

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

[4] 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

[5] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pages 1–13, 2005. ACM Press.

[6] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In ACM SIGPLAN Haskell Workshop 02. Pages 1–16, 2002. ACM Press.

[7] C. McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.