Interactive editing of tree-structured data

Bernard Fotsing Talla *

* Département d'Informatique
Université de Yaoundé I
B.P 812 Yaoundé
Cameroun
bfotsing@yahoo.fr

This work was supported by a grant delivered by the *Agence Universitaire de la Francophonie*

RÉSUMÉ. Un éditeur est un programme interactif qui conserve les modifications subies par les objets édités au cours de son exécution. Dans cette contribution, nous utilisons la monade d'états combinée avec celle des entrées/sorties pour définir un tel éditeur de manière fonctionnelle . Nous introduisons un ensemble d'opérateurs fonctionnels en vue d'aboutir à un langage dédié, plongé dans le langage fonctionnel Haskell. Ces combinateurs permettent de combiner des éditeurs pour les données structurées. L'édition est réalisée à travers une vue abstraite obtenue par projection de la structure concrète. La modification de la vue abstraite implique la propagation des changements sur la structure concrète. Nous utilisons le langage ainsi obtenu pour offrir une implémentation de la *mise à jour de vues*, un problème familier de la communauté des bases de données.

ABSTRACT. An editor is an interactive program which records the various modifications on the edited documents during its execution. In this paper, we use the state monad in combination with the monad of input/output to define an editor in a functional manner. We also define a domain-specific language (DSL) embedded in Haskell (in the form of a set of functional operators) which allows to combine tree-structured data editors. The data can be edited through an abstract view obtained by projection of the concrete structure. The modification of the abstract view implies the propagation of the updates on the concrete representation of the document. We use our DSL to implement the problem of *view update*, a well-known problem of the database community.

MOTS-CLÉS : Grammaires algébriques, Programmation fonctionnelle, DSL, données structurées, mise à jour de vues

KEYWORDS : Context-free Grammars, Functional Programming, DSL, Structured Data, View Update

1. Introduction

The problem of designing and implementing syntax-directed editors is well documented, see for example [13] for a survey. Our own understanding of the subject matter is based on the *Modeless structure editor* of Sufrin and Oege de Moor [12].

The tree representation of a document in an interactive editor should be at the same time localized (giving a focus on the part of the document on which the current editing action takes place) and partially defined (because the document being edited is still incomplete). A data structure is intentionally represented in the form of a syntax tree associated with an abstract context-free grammar (we do not consider terminal symbols since we are not interested in any concrete syntax). In order to take into account the current editing focus, we represent the structured document by a zipper, a data structure introduced by Gérard Huet [8] for representing a subtree together with its context, i.e. a tree with a focus that points to some node inside it. The main purpose of the zipper is to facilitate the navigation through the document ; but it also enables one to directly localize the editing actions at the proper place in the document (the current position of the cursor). We intend to handle generic editing operations, namely the classical insert, cut, copy, and paste operations ; genericity means that the basic insertion operations are canonically associated with the productions of the abstract context-free grammar. We should also be able to undo and redo each of the preceding operations at any moment via a clipboad. The concrete intentional representation can be manipulated in a synchronous manner by a user through an abstract view. This abstract view may be given as in [1] by a projection on a subset of syntactic categories of the grammar, called visible symbols. Any modification of the abstract view should be propagated to the concrete representation.

Greenwald and al. [5] has proposed a linguistic approach of this view update problem, a familiar problem of the database community. They consider a set of combinators, the so-called *lenses*, to obtain a domain-specific language (DSL) for bi-directional tree transformations. A lens is a bi-directional map relating a concrete structure to its abstract view. Our purpose is to put forward a domain-specific language, embedded in Haskell, for generically specifying editors for structured documents. Incidently we address the update view problem by describing the concept of lifting transformations, which associates with each legal transformation of an abstract view, an equivalent transformation of the concrete structure. A bijective correspondance can be established between the “uniform” liftings of transformations and the so-called *well-behaved lenses* of Greenwald and al. [5] ; and we show that the view update problem can easily be implemented using our DSL.

The rest of the paper is organized as follows. In the next section, we present a brief introduction to monads, an algebraic structure used to implement side effects in a functional context. Section 3 presents the main contribution of this paper, namely the definition of a domain specific embedded language for generic structured editors. We end in section 4 by illustrating our approach for editing a context-free grammar and by providing an alternative implementation of the well-known view update problem.

2. State monad associated with I/O

In the paradigm of functional programming, the evaluation of an expression is “pure”, i.e. there are no side effects. Haskell is a purely functional language which uses monads to handle computations with side effects.

The concept of state in programming languages is identified with any abstraction of the history of the execution of a program. It may for instance be a counter incremented with each evaluation of a variable, a chain of characters which contains the trace of the execution of a program, buffers of input/output, etc. For that purpose the state transformer type has been introduced in Haskell :

```
ST s a = ST (s -> (s,a))
```

where s indicates the type of the objects representing the state and a the type of the expression to be evaluated. This evaluation depends on the context, held by state s. The result of the execution of the function $(s \rightarrow (s, a))$ is a pair made of the subsequent state and the value of the associated expression. The corresponding monadic state transformer is very useful to pass the state information along computations using the method $bind$ of the class $Monad$.

In addition to this transformation of states, interactive programs require interactions with the external world (the end-user). For that purpose we combine the state transformer with the monad of input/output (IO) resulting in the following definition :

```
newtype IOState s a = IOState (s -> IO (s,a))
```

The result is thus likely to react to the interactions of the user. The code implementing this type as a monadic type system is given by the following instantiation of the monad class

```
instance Monad (IOState s) where
 -- return :: a -> IOState s a
 return a = IOState (\s -> return (s,a))
 -- >>= :: IOState s a -> (a -> IOState s b) -> IOState s b
 c >>= k = IOState (\s -> (applyIO c s) >>= (\(s,a) -> applyIO (k a) s))
```

where the function $applyIO$, used to extract the state and the value of an input/output state monad from an initial state, is defined by

```
applyIO :: IOState s a -> s -> IO (s,a)
applyIO (IOState f) s = f s
```

3. A DSEL for interactive structured editing

3.1. Definition of an editor

Intuitively, an editor is an interactive program which reacts to the actions of a user to modify its internal representation, and which produces a result corresponding to a view of this internal representation. Thus, an editor is a state monad associated with the monad of I/O which takes as input, edit actions, and returns as output a (possibly empty) list of commands. The Figure 1.a gives a graphical representation of an editor.

The user actions are transformed into edit actions (defined in an owner format) and placed in the input buffer inp. The state s of the monad models the internal representation of the editor (represented in general in the form of the zipper [8]). The value of the expression associated with the monad is carried by the output buffer out. These output values are commands (edit actions) which must be transmitted to the same editor or to another one to carry out specific tasks.

Finally, an editor is a function which associates to an edit action an interactive program ($IOState$ monad) :

```
type Editor inp out s = inp -> IOState s [out]
```

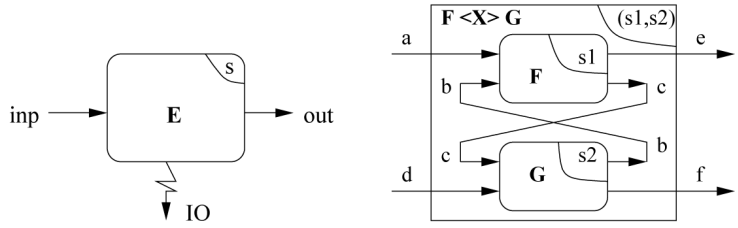

Figure 1. ***a)*** *The architecture of an editor* ***b)*** *Combining two editors F and G*

3.2. Combining two editors

We thus view editors as open reactive processes : they react to input edit commands by possibly emitting other edit commands to similar editors. When combining two editors F and G (as it is represented in the Figure 1.b) we want to internalize the edit commands emitted by one of them and received by the other one. We therefore split the input and output set of ports of each editor accordingly : we let F be an editor with internal state of type s_1 whose input edit actions are either of type a, or b (*Either a b*) and whose output commands are either of type e, or c (*Either e c*). Similarly the state of the editor G is of type s_2, its input edit actions are of type c, or d (*Either c d*), and its output commands are of type b or f (*Either b f*). Now the corresponding state of the combined editors is a pair (s_1, s_2) made of the corresponding states for each constituent subeditor. This is due to the fact that it is a synchronous composition and we do not have to represent any internal buffers of communication between them. Its input are of type a, or d and its output of type e, or f. Thus the internal edit commands from F to G (of type c) or from G to F (of type b) are externally not accessible.

One can for instance use this combinator to connect a structured editor (reacting to the edit action of an external user) to its clipboad (also seen as an editor). The clipboard reacts to the external *undo* and *redo* commands and it interacts with the editor. The clipboard may then be designed independently of the editor. We illustrate another application of that combinator to the view update problem by allowing the edition of the concrete view of a document through commands arising from the edition of the corresponding abstract view. This combinator (see Fig. 1.b) is based on two more basic combinators corresponding respectively to the disjoint union of two editors and a loop construct.

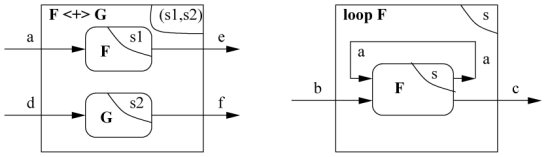

Figure 2. ***a)*** *Combinator* ***<+>*** ***b)*** *Combinator* ***loop***

The combinator <+> for the disjoint union of two editors is represented in Fig. 2.a. The types of the corresponding editors should be as follows :

```
F :: Editor a e s1 = a -> IOState s1 [e]
```

```
G :: Editor d f s2 = d -> IOState s2 [f]
F<+>G :: Editor (Either a d) (Either e f) (s1,s2)} =
        (Either a d) -> IOState (s1,s2) [Either e f]
```

If we plug the editors F and G side by side, we obtain an object of the following type

```
(Either a d) -> IOState (s1,s2) (Either [e] [f])
```

We therefore need a conversion function *fusion*

```
fusion :: IOState s (Either [e] [f]) -> IOState s [Either e f]
fusion p = mapIOState fct p  where fct (Left xs)  = map Left xs
                                   fct (Right xs) = map Right xs
```

to transform the type *Either [e] [f]* to the required type *[Either e f]* :

```
(<+>)::Editor a e s1 -> Editor d f s2 -> (Either a d)(Either e f)(s1,s2)
f <+> g  =   h where h (Left a)  = fusion (lift_left (f a))
                    h (Right d) = fusion (lift_right (g d))
```

The functions *lift_left* and *lift_right* allow for the transformation of an editor with one state into an editor with two states whose only left (respectively right) state and value are concerned by the editing process.

```
lift_left :: IOState s a -> IOState (s,s') (Either a a')
lift_left (IOState f) = IOState g
 where g (s,s') = mapIO (\(s1,x) -> ((s1,s'),Left x)) (f s)
lift_right :: IOState s a -> IOState (s',s) (Either a' a)
lift_right (IOState g) = IOState h
 where h (s',s) = mapIO (\(s1,x) -> ((s',s1),Right x)) (g s)
```

The functions *mapIO* and *mapIOState* are used to manipulate the I/O state monad.

```
mapIO :: (a->b)-> IO a -> IO b
mapIO f p = do{x<-p; return (f x)}
mapIOState :: (a -> b) -> IOState s a -> IOState s b
mapIOState f p = do { x <- p; return (f x)}
```

The loop combinator (see Fig. 2.b) is then given as follows :

```
loop :: Editor (Either a b) (Either a c) s -> Editor b c s
loop f = g where g b = h (Right b)
                 h x = f x >>= (mapIOState concat) . (mapM h)
```

The composition of two editors is then obtained, using the preceding two combinators, by adding the necessary wiring to redirect the various output ports to the appropriate input ports (see Fig. 3) :

```
(<X>) :: Editor (Either a b) (Either e c) s1 ->
         Editor (Either c d) (Either b f) s2 ->
         Editor (Either a d) (Either e f) (s1,s2)
f <X> g = loop (bimap phi (f <+> g) psi)

phi :: Either(Either b c)(Either a d) -> Either(Either a b)(Either c d)
phi (Right (Left a))  = Left (Left a)
phi (Left (Left b))   = Left (Right b)
phi (Left (Right c))  = Right (Left c)
phi (Right (Right d)) = Right (Right d)
```

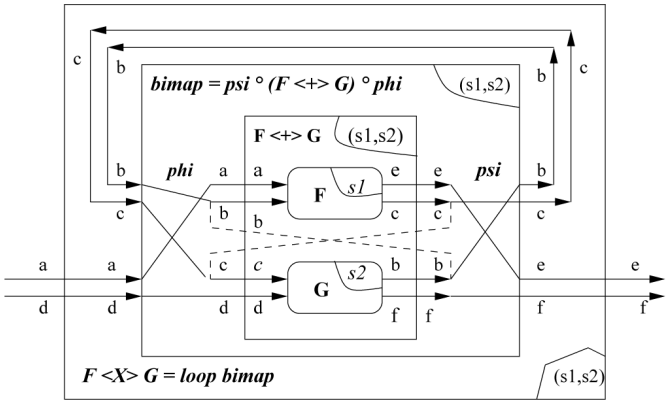

Figure 3. *Graphical representation of the solution (<X>)*

```
psi :: Either(Either e c)(Either b f) -> Either(Either b c)(Either e f)
psi (Left (Left e))   = Right (Left e)
psi (Left (Right c))  = Left (Right c)
psi (Right (Left b))  = Left (Left b)
psi (Right (Right f)) = Right (Right f)

bimap::(in'->inp)-> Editor inp out s ->(out -> out')->(Editor in' out' s)
bimap phi e  psi = (mapIOState (map psi)) . e. phi
```

4. Illustration : editing a context-free grammar

An editor associated with a context-free grammar reacts to the usual edit actions (insertion and deletion of a production, navigation) by updating its internal state (a zipper) and it forwards that information to the clipboard editor (via its input buffer) so that it can update its own internal state. On the other hand, the clipboard editor receives from the end-user the edit actions *Undo* and *Redo* to cancel or redo the preceding actions. Its state is a set of two lists (stacks) of actions done (which can possibly be cancelled by the action *Undo*) and actions cancelled (which can possibly be re-done by the action *Redo*). This situation is represented in Fig. 5.a. The editor itself may be further decomposed due to the distinction usually made between a concrete and an abstract view of the manipulated document. As an illustration, let us consider the mathematical expression $\frac{1}{x} = (x + 2)$ formed according to the following context-free grammar.

```
Exp --> Id | Nb | Fr | ( Exp ) | Exp Op Exp | Fr Op Exp
Fr  --> Exp Exp
Nb  --> 0 | .. | 9
Id  --> a | .. |z | A | ..| Z
Op  --> + | - | = | * | /
```

The derivation tree of this expression, represented in Fig. 4.a, is the concrete structure manipulated during the edition. If we suppose that the visible symbols are those marked by small black points, the algorithm of projection described in [1] will give the abstract view represented in Fig. 4.b. All user modifications on the abstract view must be propagated on the concrete view ; and this again may be obtained by composing a pair of editors associated respectively with the concrete and the abstract representations (see Fig. 5. b).

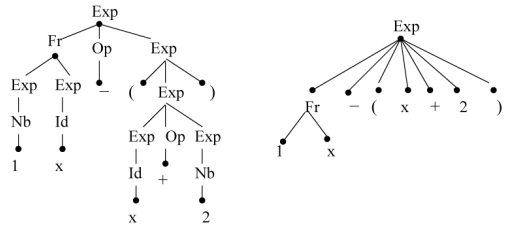

Figure 4. ***a)*** *Concrete view* ***b)*** *Abstract view*

Let C be the set of concrete views, A the set of abstract views, a transformation function of abstract views is associated with an edit action such as *insert, delete, copy, paste, navigation operations*, etc. We define by $T \subseteq A \to A$ the set of acceptable transformations, and $\pi : C \to A$ the projection function on the visible symbols. As noticed in [1], the same abstraction may have map to an infinite number of concrete views.

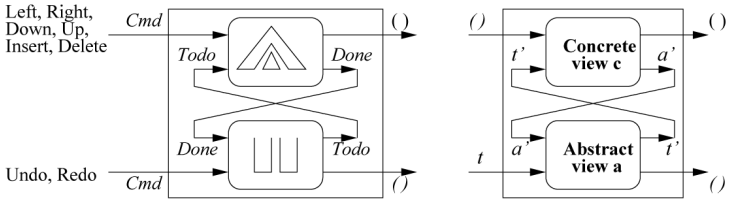

Figure 5. ***a)*** *Context-free grammar editor* ***b)*** *Lifting transformations*

Definition 1 (Lifting of transformations) *$\theta :: T \to (C \to C)$ is a lifting of the set of admissible transformations $T \subseteq A \to A$ to C, if $\pi\ ((\theta\ t)\ c) = t\ (\pi\ c)$ for all $t \in T$ and $c \in C$. This lifting is said to be uniform if moreover $t_1\ (\pi\ c) = t_2\ (\pi\ c) \Rightarrow \theta\ t_1\ c = \theta\ t_2\ c$.*

This corresponds to the view update problem where we explicitly take into account the transformation of the abstract view, an aspect that is encapsulated in the language defined in [5] for bi-directional tree transformations. We guarantee that the concrete view $c' \in C$ can be obtained unambiguously by applying the lifting θ on the transformation t and on the concrete view $c\ (\theta\ t\ c)$, such that $\pi\ c' = a'$ where $a' = t\ a$ and $\pi\ c = a$. When an action t is applied on the current abstract view, the corresponding transformation $t' :: C \to C$ obtained by $\theta\ t$ is instantly sent to the concrete view editor which modifies the structure of the object being edited. The projection π is also applied to the new concrete view to refresh the state of the abstract view editor with a'.

5. Conclusion and further work

We have introduced a small Domain Specific Language Embedded in Haskell for interactive structure editing using the concept of monads. We have illustrated this DSL by providing a batch editing environment for tree-structured data described by context-free grammars, and a solution for the view update problem. A complex structured document is intentionally represented as a tree decorated with attributes characterised by an attribute grammar as described in [4]. Therefore, we would like to enrich our DSL by a set of functional combinators (similar to the functional monadic parser combinators [3, 10]

and the editors combinators of O. Braun [2]) to provide a Domain Specific Language embedded in Haskell for the encoding of attribute evaluators. Using these combinators the programmer will specify his attribute grammar (mainly he will write the semantic rules) but by doing so he will actually build an Haskell program for the corresponding evaluator of attributes or even maybe for an associated interactive editor. The structure of arrows of J. Hughes [9], a generalization of monads, and more specifically the new notation for arrows introduced by R. Paterson [11] (similar to the *do* notation of monads) can be useful in that respect.

6. Bibliographie

[1] E. BADOUEL, M. TCHOUPE TCHENDJI, « Projections et cohérence de vues dans les grammaires algébriques », *8e CARI*, novembre 2006, Cotonou, Bénin.

[2] O. BRAUN, « Editors combinators. improving the user interface. », Master's thesis, University of Munich, 2000.

[3] JEROEN FOKKER, « Functional parsers », Advanced Functional Programming ; First international Spring School on Advanced Functional Programming Techniques, pages 1-23 , may 1995. Springer (LNCS 925), Berlin, Heidelberg.

[4] B FOTSING TALLA, G. E. KOUAMOU, « Une approche formelle de description et de manipulation des objets structurés mathématiques ». *Revue ARIMA*, 3(2), :pp 71-86, 2005.

[5] MICHAEL B. GREENWALD, JONATHAN T. MOORE, BENJAMIN C. PIERCE, ALAN SCHMITT, « A language for bi-directional tree transformations », *Technical Report MS-CIS-03-08,*, University of Pennsylvania, August 5, 2003.

[6] « Haskell, a purely functional language », *http ://www.haskell.org*.

[7] ZHENJIANG HU, SHIN-CHENG MU, MASATO TAKEOCHI, « A programmable editor for developing structured documents based on bidirectional transformations », *In Proceedings of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation*, Verona, Italy, August 2004. ACM Press.

[8] GÉRARD HUET, « The zipper », *Journal of Functional Programming*, n° 7(5) : 549-554, 1997. Functional Pearl.

[9] JOHN HUGHES. « Generalising monads to arrows ». *Science of Computer Programming*, 37(1-3), :67-111, 2000.

[10] G. HUTTON, E. MEIJER, « Monadic parsing in Haskell », *Journal of Functional Programming*, n° 8(4) :437-444, 1998.

[11] ROSS PATERSON, « A New Notation for Arrows », Department of Computing, City University, London.

[12] BERNARD SUFRIN, OEGE DE MOOR, « Modeless structure editing », *Programming Research Group*, 1999.

[13] G. SZWILLUS, L. NEAL, « Structure-based editors and environments », *Academic Press*, 1996.