Editor Combinators — A First Account

WOLFRAM KAHL

OLIVER BRAUN

JAN SCHEFFCZYK

Bericht Nr. 2000-01
Juni 2000

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Editor Combinators — A First Account

Wolfram Kahl* Oliver Braun Jan Scheffczyk

Department of Computing Science, Federal Armed Forces University Munich

Technical Report 2000-01, 16 June 2000

Abstract

In the functional programming context, parser generators are more and more replaced by parser combinator libraries. In the same way we strive to replace generators for syntax-directed editors by sets of *editor combinators*, and present a first working prototype.

Since lexical behaviour of editors is more complex than that of parsers, we present the abstraction of *scanlets* that are not only used to build parsers, but also to direct interactive editing.

In the course of this work we also identify uses for different extensions of Haskell towards second-order polymorphism, only some of which are currently implemented in existing Haskell systems.

Contents

*Contact author; e-mail: `kahl@informatik.unibw-muenchen.de`

1 Introduction

Just as parsers may be generated from language descriptions via parser generators, also syntax-directed editors may be generated from (more detailed) language descriptions; for a prominent example see [RT89].

Following [Wad85], there has evolved a strong tendency in the functional programming community to construct parsers using *parser combinators* instead of via generators. One "school" uses monadic parser combinators [HM98, Lei00], while for the fast, error-correcting "pre-compiling" parser combinators of [SD96, SAA99] the monad abstraction turns out to be too restrictive.

The main advantages of the use of combinator libraries for parser construction (or, in fact, for any purpose) are, as has frequently been explained, on one hand compositionality, and thus greater modularity, and on the other hand the fact that only one language needs to be used, such that the full power of the host language is also available for parser construction.

In this paper we set out to leverage these advantages for editor construction.

Before starting to define editor combinators, we need to clarify what an editor is. We use the following definition:

> An *editor for a type* `a` is a piece of machinery that allows to manipulate *values* of type `a` by maintaining an *internal state* defining (perhaps among other data) a *view* on the *current value*.
> Of any one editor, there may be several *editor instances* active at any given moment, each manipulating a different value of type `a` by maintaining its own value of the internal state type.

Editors that most people will be familiar with are programs such as `vi`, Emacs, or `edit`; these programs are mostly used to edit the contents of text files, so we consider these editors as editors which edit string values. The internal state usually contains the edited strings together with at least a cursor position, a display window which determines which part of the edited value is visible (where this display window usually contains the cursor position), and often also selections or "clipboards". The cursor position and the display window together determine the *view* on the edited text; this can be influenced — without changing the edited value — via cursor movements and other navigation actions.

Different editor instances of these editors are then different invocations of the same editor, which usually edit different files, i.e., different string values.[1]

For more examples, in modern graphical user interfaces one encounters editors in many disguises: check boxes serve as editors for Boolean values, and radio button groups for enumeration types — for these two examples, there is often no extra internal state besides the represented value, although the mapping from values to the optical appearance of the views may be non-trivial. Entry areas are often used as editors for single lines of text, and "text widgets", such as that of the Tk toolkit [Ous94], are configurable text editors which may be embedded into larger applications and which may have lots of internal state.

Some editors, most notably programmable editors like Emacs, offer different "*modes*" for different file formats: when editing program source files, certain key strokes will induce editor actions that are different from those induced by the same key strokes when editing a text document. From our point of view, each mode is an editor of its own (which probably shares some functionality with its base editor and with other modes of that editor).

Now there are applications where different modes are activated for different regions of one text file: Block comments may make the editor behave different from program code, equations may induce an extra mode in a text editor, a *literate programming system* [Knu84] like the CWeb mode in emacs uses a text document format mode for documentation, and a program code mode for code chunks.[2]

As programming languages and text document formats vary, it would be extremely attractive to be able to *combine* arbitrary text editors with arbitrary program editors for arbitrary literate programming

[1] The possibility that an editor may offer *multiple views* on a single edited value is ignored here for the sake of simplicity, but should not be too hard to incorporate into our approach.

[2] The current document is actually a literate program containing the Haskell source code of the editor combinator library it describes; for more information see page 22.

systems, all along the principle that a literate document consist of alternate text and program fragments, separated by some markers specific to the literate programming system.

Our thesis is that such systems can be realised by a library of *editor components* and *editor combinators*, together with an *editor instantiation machinery*:

- *Editor components* are basic building blocks, perhaps including editors for program chunks and editors for text chunks, and also editors for the separating markers — for constant separators, the value edited by the separator editors is the trivial value `()` (which is the unique value of the trivial type `()`, the type of "zero-tuples").
- *Editor combinators* are functions which take editors as arguments and construct new, combined editors from them. The most important combinators are
  - sequential composition for a sequence of two editors; the values edited by the combined editor are pairs consisting of values of the component editors, and
  - alternative composition for a choice of different editors for different values; the values edited by the combined editor are elements of a variant type with an alternative for each component.
- Finally there needs to be a way to construct an editor instance from a combined editor, which is provided by the *editor instantiation machinery.* For `vi`, this is just the program loader of the underlying operating system, but then there are no easy-to-use combinators available that allowed `vi` to be embedded seamlessly into more complicated applications. For a Tk text widget, the instantiation machinery may be said to be the Tk runtime environment, and this may be used to create several instances of "identical" editors within the same application.
  For the combined editors built in this paper we use the Haskell GUI system FranTk [Sag99]. In pure FranTk, an "edit area", which essentially provides access to the Tk text widget, is characterised by a list of "edit area configuration options" and instantiated into a "component" via application of the function `mkEdit :: [Conf Edit] -> Component` — this allows the same abstract edit area (i.e., the same list of edit area configuration options) to be instantiated into several distinct editor instances within a single application.

The story so far is very similar to the setting of *parser combinator libraries*, which also provide elementary parsers (on top of some primitive token concept), parser combinators, and a parser application function that calls the combined parser on some concrete input, constructing a parsing result.

A more general approach to editors that change behaviour depending on the context of the editor's focus (which usually is the cursor position) is that of structure-oriented editors [RT89]. There, in extreme cases the "mode" may change for every token of the input. For example, some editors provide "completion" that attempts to guess identifiers or keywords from already-typed-in prefixes — such a feature only rarely makes sense for numbers. Furthermore, recalling the literate programming example, the concept of token may differ in different parts of the edited document, prohibiting the use of a single monolithic scanner.

From this we derive the motivation to generalise on the lexical level, too. We therefore replace the primitive token concept of parser combinators by the concept of *scanlet.* A scanlet is intend to represent scanning functionality for a single token class, but not as a finite-state machine that just accepts or rejects as is sufficient for mere parsing. The concept of scanlet is tuned for the editor context, and is the essential ingredient of primitive, token-level editors. Since features such as completion belong to the lexical level, it is natural that such features can be incorporated into scanlets.

A combined editor will therefore draw from a scanlet library for its token editors, and then use combinators to combine these into editors for grammatical structures. Like the situation in parser combinators, where the result of parsing a string usually is not a string, but some (user-defined) data-structure, also in our editor combinators we offer the possibility to combine editors that, from the point of view of the enclosing application, edit user-defined data-structures via the (enriched) string view they present at the graphical user interface.

In the context of functional programming, the best-known editor data-type (apart from primitive character list pairs and the like) is the *Zipper* of Huet [Hue97]. The Zipper provides a navigation and modification interface for trees labelled with strings, and an efficient internal representation of the current view which does not *contain* the current value, but an enriched representation of this value which is tuned towards efficient navigation and modification and also allows efficient *reconstruction* of the current value from the current view.

The drawback of the Zipper is that it is geared towards a string representation of the edited data, and does not allow direct editing of user-defined data-types. Huet shows how to construct custom zippers for user-defined data-types, but for this construction to be really useful, generic programming tools would be necessary since they would allow to mechanise the tedious task of creating the complex auxiliary data-type definitions required there.

In [SdM99], a single-language structure editor is presented, where the internals are based on the Zipper of Huet [Hue97]. A working implementation of this editor [SdMS00] is based on the graphical user interface system FranTk [Sag99].

In this paper we sacrifice the efficiency of the zipper, concentrating on the task to identify the interface that is needed for editor combinators

- that allow to easily construct editors for user-defined data-types,
- that can easily be used with present-day Haskell systems, and
- that allow to program the behaviour of the resulting editor on a reasonably high level.

We start the body of the paper in Sect. 2 by discussing algebraic properties that basic parts of the editor interface will have to fulfil. Then we present a user-level view of our primitive editor building blocks in Sect. 3, and of our primitive editor combinators in Sect. 4, where we also extend these with a few derived combinators.

This allows us to present a full-fledged example editor in Sect. 5 even before we dive into the details of lexical analysis composed from *scanlets* in Sect. 6.

In Sect. 7 we shortly sketch the implementation of our editor data-type, with details on the treatment of locations following in Sect. 8. In Sects. 9 and 10 we discuss areas where more support for second-order polymorphism in Haskell would be useful.

The full implementation of our editor combinator library may be found in the appendix and on our editor combinator web page at URL: `http://ist.unibw-muenchen.de/EdComb/`.

2 An Abstract View on (Part of) the Editor Interface

From now on, we use the notation of the purely functional programming language Haskell [HPJW+92] both for actual code fragments like declarations and definitions, prefixed with the "Bird tracks" of literate Haskell programs, "> ", and for laws, specifications and ficticious code, all prefixed with "> ".

We assume the type of editors for values of type `a` and with internal state `s` to be an abstract type for the time being:

```
> data Editor s a
```

Such an editor is an abstract entity of which there may be many concrete instances in any application, and every instance will maintain its own state value of type `s`.

The value inside the editor can be extracted via:

```
> eExtract :: Editor s a -> s -> a
```

Conversely, for any value of type `a` it is possible to construct a default editor state with:

```
> eBuild :: Editor s a -> a -> s
```

For all reasonable editors we shall expect `eBuild` to be total and injective, and `eExtract e` to be the left inverse of `eBuild e`, that is `eExtract e . eBuild e = id :: a -> a`, but we cannot expect full isomorphism since `eBuild e` usually will not be surjective, and `eExtract e` not injective, and sometimes not even total.

For initialisation we do not provide an initial *value*, but only an initial *state*:

```
> eEmpty :: Editor s a -> s
```

This allows us to define editors where `eExtract s (eEmpty e) = undefined`.

There are navigation functions which all have the same type, and which just serve to change the view on the current value, which for this purpose is considered to be tree-structured:

```
> eRoot, eLeft, eRight, eUp, eDown :: Editor s a -> s -> s
```

They should not change the value itself, so we usually demand `eExtract e . eNav e = eExtract e` for any `eNav` from the above.

In addition there are state predicates

```
> eLeftmost, eRightmost, eTopmost,  eBottommost :: Editor s a -> s -> Bool
```

for which we have:

```
< eLeftmost   e s = (eLeft  e s == s)
< eRightmost  e s = (eRight e s == s)
< eTopmost    e s = (eUp    e s == s)
< eBottommost e s = (eDown  e s == s)
```

One might be tempted to demand `eRight e (eLeft e s) = s` whenever `eLeftmost e s` is false, but this is actually too restricting.

Further data available in any editor include a function to turn edited values into strings, and a function to initialise the editor from a string:

```
> eShow  :: Editor s a -> s      -> String
> eRead  :: Editor s a -> String -> s
```

Saving and reading back should of course return the initial edited value, so we demand:

```
< eExtract e . eRead e . eShow e = eExtract e
```

(This leads to problems with parsing when taken literally also for states for which `eExtract e` delivers an only partially defined result.) We can compose `eBuild e` at the right and obtain:

```
< eExtract e . eRead e . eShow e . eBuild e = eExtract e . eBuild e = id
```

This shows that `eShow e . eBuild e` has to be injective, and `eExtract e . eRead e` has to be surjective. Only in very special circumstances injectivity of `eRead e` or even `eExtract e . eRead e` will be necessary; we shall not demand these here.

An important combinator is a kind of `map`; however, in contrast to most other situations we here also need the inverse of the embedding:

```
> eMap :: (a -> b) -> (b -> a) -> Editor s a -> Editor s b
```

This combinator is characterised by the following equations:

```
< eExtract (eMap f g e) = f . eExtract e
< eBuild   (eMap f g e) =     eBuild   e . g
```

In order have an editor `eMap f g e` fulfil the above requirements whenever `e` does, we have to demand that `g` is total and injective, `f . g = id :: b -> b` (so `f` has to be surjective), and since `eExtract e` is surjective, we also need that `f` is total, too, if `f . eExtract e` is to be total.

If injectivity of `eExtract e . eRead e` was required, it would also be necessary that the restriction of `g . f` to the range of `eExtract e . eRead e` was contained in the identity on `a` — failure to meet this condition results in more different internal editor states for at least certain values in `b`.

3 Basic Component Editors

After discussing some aspects of editors in general, we now turn to the presentation of the concrete basic component editors and editor combinators that we provide in our first editor combinator library.

We restricted ourselves to rather simple capabilities, although keeping in mind future extensibility. Some of our design choices therefore may have to be revised for future developments.

The most primitive editor has a constant appearance and no observable behaviour, so it is an editor for the trivial type, and also has the trivial type as its internal state:

```
> tagEditor :: String -> Editor () ()
```

For non-constant lexical tokens, the primitive building block is `tokenEditor` which produces editors for `Strings`, and uses for their internal state the type `SPair` which includes information about the current position inside this string. A `tokenEditor` only serves for tokens of one fixed token class; therefore the capability to recognise tokens of this token class must be built into the `tokenEditor`. We choose to do this by supplying a *scanlet* that recognises the token class in question; where the scanlet type currently happens to be the same as the parser type we use here, but the semantics is slightly different. Scanlets are discussed in detail below in Sect. 6. The `String` arguments to the `tokenEditor` are the name of the token class, and an initialisation string, respectively:

```
> tokenEditor :: String -> String -> Scanlet -> Editor SPair String
```

Below in Sect. 6 we shall see among other examples the following function which allows to construct scanlets for any data-type that belongs to the type classes `Read` and `Show`:

```
> readScanlet :: (Read a, Show a) => a -> Scanlet
```

These scanlets are then the central ingredient for the corresponding editors — the dummy argument of type `a` both in `readScanlet` and in `readEditor` is included only for typing reasons:

```
> readEditor :: (Read a, Show a) => String -> String -> a -> Editor SPair a
```

The two string arguments are passed on to `tokenEditor`, so creating editors for numbers is no effort at all:

```
> intEditor    = readEditor "<Integer>" "" (0 :: Integer)
> doubleEditor = readEditor "<Double>"  "" (0 :: Double )
```

4 Basic Editor Combinators

Sequential composition of editors is graphical juxtaposition and results in an editor for a pair which has a *pair editor state* of type `PES x y`; such a state includes a flag indicating which of the two components contains the current position:[3]

```
> type PES x y = ((x, y), Bool)
> (<*>) :: Editor x a -> Editor y b -> Editor (PES x y) (a,b)
```

An important difference between `tag`- and `tokenEditors` on one hand and sequentially composed editors on the other hand is that the internal states of the former are considered to be single units, while the internal states of the latter are always considered to be sequences of units, which are composed associatively via nested applications of `<*>`. This will be relevant when discussing locations in Sect. 8.

We also need a combinator for alternatives, with a *sum editor state* `SES x y` indicating which alternative is actually valid:

```
> (<|>) :: Editor x a -> Editor y b -> Editor (SES x y) (Either a b)
```

Alternatively composed editors may have states which are single units and other states that are sequences; when sequentially prepending such an alternative to some other editor, the location of that editor may change depending on internal state changes of the alternative editor.

Therefore, and for other structuring purposes, we need a combinator that has to be employed for editors that should be considered as representing a logical unit — this is relevant also for navigation and selection behaviour. This combinator has a *group editor state* `GES x` which includes a flag to indicate whether or not the current position focuses on the whole group:

```
> type GES s = (s, Bool)
> eGroup :: Editor s a -> Editor (GES s) a
```

[3] In modern parser combinators, for example in those of [SD96, SAA99], the sequence operator `<*>` is used with the type `Parser (a -> b) -> Parser a -> Parser b`; for transferring such a type to our editor setting we would have needed an additional argument of type `b -> (a, a -> b)`, which is not a very natural decomposition setting. Therefore we considered it more natural to build pair editors and let users employ `eMap` to transform them into editors for friendlier types.

Finally we have a combinator that allows construction of editors for recursive data-types — with its second-order typing it takes a *generic* editor transformer as its argument and returns a fixed-point editor, using *generic datatype recursion* `DRec` to construct a recursive state type:

```
> data DRec c = DRec (c (DRec c))
> eRec :: (forall s . Editor s a -> Editor (c s) a) -> Editor (DRec c) a
```

Its implementation creates a recursive editor via mapping the generic recursive datatype constructor and its inverse across its state:

```
> eRec ee = e where e = eMapState DRec unDRec $ ee e
> unDRec (DRec x) = x
```

For the editor state mapping combinator used here, the laws of Sect. 2 imply that for an editor `eMapState f g e` we need that `f` and `g` are both total, and `g . f = id :: x -> x` (so `f` is injective and `g` surjective).

```
> eMapState :: (x -> y) -> (y -> x) -> Editor x a -> Editor y a
```

On top of these basic combinators, we can define derived editor combinators.

As a first example consider "one-sided decoration editors" that are juxtapositions ignoring the value of one component:

```
> (<*) :: Editor x a -> Editor y b -> Editor (PES x y) a
> e1 <* e2 = eMap fst (\a -> (a, eExtract e2 (eEmpty e2))) $ e1 <*> e2
> (*>) :: Editor x a -> Editor y b -> Editor (PES x y) b
> e1 *> e2 = eMap snd (\b -> (eExtract e1 (eEmpty e1), b)) $ e1 <*> e2
```

For one-sided composition with editors with trivial internal state we might be tempted to abuse `eMapState` with a non-injective first argument to get rid of the `PES ()` altogether, as in the following code:

```
< (<*<) :: Editor x a -> Editor () b -> Editor x a
< e1 <*< e2 = eMapState (fst . fst) (\x -> ((x,()),False)) $ e1 <* e2
< (>*>) :: Editor () a -> Editor y b -> Editor y b
< e1 >*> e2 = eMapState (snd . fst) (\y -> (((),y),True)) $ e1 *> e2
```

However, the elimination of the Boolean overhead in `PES` has unwanted side effects: in `<*<` constructions it is, for example, impossible to reach a rightmost location via `eRight` since the "projection" always assigns positions in the left component — since `eRightmost` is inherited from the full composition it still requires `True` in the overhead.

Therefore, we have to provide special primitive combinators for this purpose — these can never focus on the ignored component:

```
> (<*<) :: Editor x  a -> Editor () b -> Editor x a
> (>*>) :: Editor () a -> Editor y  b -> Editor y b
```

5 An Example Editor

To illustrate the working of our editor combinator library, we work through an example editor for the following datatype for arithmetic expressions:

```
> data Expr = Var String | Num Integer | BinOp Op Expr Expr
> data Op   = Plus | Minus | Mult | Div
```

The global structure of the associated combinator editor is essentially the same as that of the corresponding combinator parser:

We chose a fully parenthesised representation of operator applications, and, since for typing reasons we cannot directly program a recursive editor here, we define the kernel functional for expression editors as follows:

```
> mkExprEditor0 e = identEditor <|> intEditor <|> eParen (e <*> opEditor <*> e)
```

For variable names, we use a scanlet for identifiers that we define in the next section:

```
> identEditor = tokenEditor "<Ident>" "" identScanlet
```

Parenthesisation is taken care of by a simple auxiliary combinator that uses left and right decoration:

```
> eParen e = eGroup (tagEditor "(" >*> e <*< tagEditor ")")
```

The heart of `opEditor` is a simple alternative:

```
> opEditor0 = (tagEditor "+" <|> tagEditor "-") <|> (tagEditor "*" <|> tagEditor "/")
```

What now remains to be done is gluing the parts together mainly by implementing adaptation functions between the user-defined data-types and their representations in the raw combined editors. This is all rather mechanical and might even be automated by generic programming tools.

First of all we identify the value type of `opEditor0`, which is an editor for a nested binary sum of trivial alternatives:

```
> type OpRepr = Either (Either () ()) (Either () ())
> opEditor0 :: Editor OpEditState OpRepr
```

The abstraction function `opA :: OpRepr -> Op` and the representation function `opR :: Op -> OpRepr` are trivial to program and might be generated automatically:

```
> opA (Left  x) = case x of Left _ -> Plus; Right _ -> Minus
> opA (Right x) = case x of Left _ -> Mult; Right _ -> Div
```

```
> opR Plus  = Left  $ Left  ()
> opR Minus = Left  $ Right ()
> opR Mult  = Right $ Left  ()
> opR Div   = Right $ Right ()
```

The operator editor is simply the result of mapping these abstraction and representation functions over the value type of `opEditor0`:

```
> opEditor :: Editor OpEditState Op
> opEditor = eMap opA opR opEditor0
```

Its state is given here only for completeness' sake:

```
> type OpEditState = SES (SES () ()) (SES () ())
```

Similarly, we also have to identify the representation of the expression data-type in terms of the binary sum and product data-types used for value type construction by the basic editor combinators:

```
> type ExprRepr = Either String (Either Integer (Expr, (Op, Expr)))
```

Mapping the expression abstraction and representation turns `mkExprEditor0` into an expression editor transformer, albeit between editors with different state types (the abstraction and representation functions themselves are again a trivial affair):

```
> mkExprEditor1 :: Editor y Expr -> Editor (ExprEditStateF y) Expr
> mkExprEditor1 = eMap exprA exprR . mkExprEditor0
```

```
> exprA :: ExprRepr -> Expr
> exprR :: Expr     -> ExprRepr
```

```
> exprA (Left  s                     ) = Var   s
> exprA (Right (Left  i            )) = Num   i
> exprA (Right (Right (e1, (op, e2)))) = BinOp op e1 e2
```

```
> exprR (Var   s       ) = Left  s
> exprR (Num   i       ) = Right (Left  i)
> exprR (BinOp op e1 e2) = Right (Right (e1, (op, e2)))
```

Once second-order type matching is supported we shall be able to directly use our combinator `eRec` for building editors for recursive datatypes:

```
< -- ficticious!
< exprEditor0 = eRec mkExprEditor1
```

For the time being, however, we have to take a short detour (equivalent to a slightly longer detour involving `eRec` — the details are discussed in Sect. 9): First of all we have to identify the expansion of `ExprEditStateF` that lies behind the state type of the result editor of `mkExprEditor1` — we may use the type inference mechanisms of the respective Haskell system for this purpose:

```
> type ExprEditStateF y = SES SPair (SES SPair (GES (PES y (PES OpEditState y))))
```

Next we have to produce a recursive data-type building on this type constructor, and define the inverse of its constructor:

```
> data ExprEditState = ExprEditState (ExprEditStateF ExprEditState)
> unExprEditState (ExprEditState x) = x
```

The resulting isomorphism pair serves to create an editor with a recursive state type:

```
> exprEditor :: Editor ExprEditState Expr
> exprEditor = e where e = eMapState ExprEditState unExprEditState $ mkExprEditor1 e
```

Since navigation is not handled locally by the supplied combinators, we wrap the editor into a *binding* that associates the arrow keys with the editor's respective navigation functions:

```
> exprEditor1 = eBind (stdArrowReact exprEditor) exprEditor
```

This editor can now be instantiated and turned into a FranTk component via

```
> editorComponent :: Editor s a -> WComponent
```

so that a stand-alone structured expression editor is obtained through the following line:

```
> main = display (editorComponent exprEditor1)
```

6 Scanlets

Since we have to localise token recognition, we cannot have a full scanner, but have to assemble the scanner from small pieces that we call *scanlets*.

We define the scanlet type similar to a one-result parser type, but with a different intended semantics: a scanlet returns `Nothing` if the input clearly does not start with a token of the token class the scanlet represents, and can never be extended to such a token, and it returns `Just (tok, rest)` if the input can be "legally" segmented into `init++rest` and `init` "identifies" `tok`. Here, "legally" means essentially at a token border, and the concept of "identification" gives the scanlet designer considerable power to implement keyboard shortcuts and completion, as for example a string may be taken to "identify" a token if the token is the input's only completion inside the scanlet's token class.

```
> type Scanlet = String -> Maybe (String,String)
```

A completion scanlet for a single string can be defined in the following way:

```
> completionScanlet :: String -> Scanlet
> completionScanlet s1 s2 = let (init,rest) = splitAt (length s2) s1
>    in if not (null s2) && init == s2 then Just (s1,rest) else Nothing
```

Example behaviour of completion scanlets is the following:

```
< completionScanlet "while" "w"      = Just ("while","")
< completionScanlet "while" "where"  = Nothing
< completionScanlet "while" "whiley" = Just ("while","y")
```

The last example shows that, without additional information about how to recognize token borders, these completion scanlets are still too simplistic for general use.

Another example for scanlet construction works for arbitrary types that belong to both the `Read` and `Show` classes; it employs `reads` to check for the presence of a legal input, and converts it back to the recognised token via `show` — this is powerful, but of course rather inefficient.

We use a dummy argument together with the Haskell 98 prelude function `asTypeOf` to indicate the desired type to the class member functions:

```
> readScanlet :: (Read a, Show a) => a -> Scanlet
> readScanlet x s = case reads s of []         -> Nothing
>                                   (y,rest):_ -> Just (show (y `asTypeOf` x), rest)
```

This yields, for example, `readScanlet (0::Double) "12a" = Just ("12.0","a")`.

Using these scanlets, and `eMap` with `show` (which usually should be injective) and `read`, we obtain token editors for `Read` and `Show` types:

```
> readEditor :: (Read a, Show a) => String -> String -> a -> Editor SPair a
> readEditor title s x =
>   eMap (\ s -> ((read s) `asTypeOf` x)) show $ tokenEditor title s $ readScanlet x
```

Another example is the identifier scanlet we used in our example editor. We employ the usual convention that identifiers should start with letters and contain only alphanumeric characters:

```
> identScanlet [] = Nothing
> identScanlet (c:cs) | Char.isAlpha c  =  let (ls, rest) = span Char.isAlphaNum cs
>                                          in Just (c:ls, rest)
>                     | otherwise       =  Nothing
```

Parsing with Scanlets

Apart from being used in the interactive editor, scanlets also have to be the basic building blocks for the parser associated with the editor.

We use a parser type that replaces the usual list of successes [Wad85] with *at most one success*, since this is closer to the situation in an editor where only one possible interpretation of the current buffer state is kept (we have `eExtract :: Editor s a -> s -> a` and not `eExtract :: Editor s a -> s -> [a]`):

```
> type Parser a = String -> Maybe (a,String)

> (<<$>>) :: (a -> b) -> Parser a -> Parser b
> f <<$>> p = \ cs -> case p cs of Nothing -> Nothing
>                                  Just (x,rest) -> Just (f x, rest)
```

Instead of the more customary

```
> (<<*>>) :: Parser (a -> b) -> Parser a -> Parser b
```

we need a variant that commits to the current alternative if the first parser succeeds, and applies the returned function to a default argument if the second parser fails:

```
> pSeqDft :: Parser (a -> b) -> Parser a -> a -> Parser b
> pSeqDft p q dft = \ cs ->
>   case p cs of Nothing -> Nothing
>                Just (f, cs') -> Just (f y,cs'')
>                  where (y,cs'') = case q cs' of
>                                     Nothing -> (dft,cs')       -- [] instead of cs'?
>                                     Just qresult -> qresult
```

The committing at the first component has as another consequence that parsing results become available lazily, improving performance considerably. A minor point deserving consideration is that it might be reasonable to throw away the remaining input in the error case, or otherwise to investigate possibilities that allow to reconstruct partially defined values.

The alternative parser is again standard:

```
> (<<|>>) :: Parser a -> Parser a -> Parser a
> p <<|>> q = \ cs -> case p cs of Nothing  -> q cs
>                                  Just result -> Just result
```

Every editor has a parser for its state type, which is used to implement `eRead`:

```
> eParser :: Editor s a -> Parser s

> eRead  :: Editor s a -> String -> s
> eRead e ss = case eParser e ss of Nothing      -> eEmpty e
>                                   Just (s,ss') -> s
```

In parser combinators — let us take as reference those of [SD96, SAA99] — there are two kind of primitive token parsers: on one hand one for recognising *one* specific token provided as an argument:

```
> UU_Parsing.pSucceed :: UU_Parsing.Symbol s => s -> UU_Parsing.Parser s s
```

On the other hand there is

```
> UU_Parsing.pToken   :: UU_Parsing.Symbol s => UU_Parsing.Parser s s
```

which just returns the next token, and which can, in conjunction with the combinator `<?>`, be turned into a parser that recognises only tokens from a specific token class — this is a frequent pattern, since not for all token classes parsers can be assembled conveniently via `pSucceed` and the alternative parser combinator.

In our approach, these token class parsers are constructed directly from scanlets; they inspect the scanlet results of prefixes of the input as long as the scanlet does not return `Nothing` (`cTrue = const True`), filtering those where the second component of the scanlet result is empty and the first equals the scanlet input, and of these return the last element:

```
> scanletP :: Scanlet -> Parser String
> scanletP scanlet s =
>   let ps = map (keep1 scanlet) (splits s) -- splits s = zip (inits s) (tails s)
>       ps' = takeWhile ((maybe False cTrue) . fst) ps
>       ps'' =  [p | (Just (pref,[]), p@(init,rest)) <- ps' , pref == init ]
>   in case ps'' of [] -> Nothing; l -> Just (last l)

> keep1 f p@(x,y) = (f x, p)
```

Interactive Use of Scanlets

Consider the following examples for behaviour of scanlets and of scanlet parsers:

```
< readScanlet (0::Double ) "12" = Just ("12.0","")
< readScanlet (0::Integer) "12" = Just ("12","")

< scanletP (readScanlet (0::Double )) "12" = Nothing
< scanletP (readScanlet (0::Integer)) "12" = Just ("12","")
```

This implies that in a language that uses these scanlets, integer literals and floating point literals are disjoint, and the parser will be able to differentiate without any problems.

In the editor, however, the behaviour of the scanlets indicates that, after receiving the input `"12"`, it is not yet clear which token the user wants, and both possibilities should be kept open.

Therefore, the alternative editor state not only includes its two obvious alternatives, but also a third *un-committed* alternative, where it behaves essentially like a `tokenEditor`, but checking its input not only with a single scanlet, but with scanlets for the initial tokens of all possible alternatives.

More concretely, we define the following datatype to represent one such alternative by a triple containing a scanlet for its initial token, a string that might be used in interactive selection (perhaps via a pop-up menu), and a function determining the resulting editor state from the current input, should the alternative be chosen:

```
> type First s = (Scanlet, String, String -> s)
```

We can easily map string and state transformers over lists of `First`s:

```
> mapFirst :: (String -> String) -> (s -> t) -> First s -> First t
> mapFirst f g (scl,str,t) = (scl,f str,g . t)
> mapFirsts f g = map (mapFirst f g)
```

We have to look more closely into interaction with the editor now: every editor has a function determining its behaviour for keyboard input

```
> eReact :: Editor -> KeyReact s
```

where the reaction type currently returns a triple containing the result state, an indication whether the input key was consumed, and an indication whether the editor will not accept further input:

```
> type Reaction s = (s, Maybe Key, Bool)
> type React s = s -> Reaction s
> type KeyReact s = Key -> React s
```

Since the additional information is mostly for communication inside composed Editors, we provide a simple reaction function for external use:

```
> eKey :: Editor s a -> Key -> s -> s
> eKey e k = fst3 . eReact e k
```

The internal states of the `tokenEditor` and of the un-committed alternative represent a string together with the current position inside that string in a standard way via a pair of strings, the first of which is reversed:

```
> type LPair a = ([a],[a])
> type SPair = LPair Char
```

We use functions `lpEmpty`, `lpLeft`, `lpRight`, `lpBuild`, `lpExtract` etc. (see A.3) for manipulating this kind of state. A first simple application is the reaction for a single scanlet, as it is used in the `tokenEditor`:

```
> scanletReact :: (String -> Maybe (String, String)) -> KeyReact SPair
> scanletReact scanlet k p@(init,rest) =
>   let current = lpExtract p
>       currentOk = case scanlet current of Nothing -> False
>                                           Just (_,u) -> null u
>   in case k of KeyChar c -> let new = lpExtract (init,c:rest)
>                             in case scanlet new of
>                                  Nothing -> (p,Just k, currentOk)
>                                  Just (init', []) -> ((c:init, rest),Nothing,False)
>                                  Just (init', rest') -> (p, Just k, currentOk)
>                Delete -> (lpEmpty,Nothing,False)
>                BackSpace -> lpBackSpace p
>                _ -> (p,Just k, currentOk)
```

```
> lpBackSpace :: React (LPair a)
> lpBackSpace p@(xs, ys) = case xs of [] -> (p,Just BackSpace,True)
>                                     (_:xs') -> ((xs', ys), Nothing, False)
```

In the reaction function of editors constructed via `<|>`, we are confronted with a list of `First`s which decide what the `SPair` representing the current state is transformed into.

Our current implementation does the following when confronted with character input:

- `cs` is bound to the string that the current state is going to represent if the character `c` is accepted,
- `firsts'` is that selection of `firsts` (obtained via `firstsFilter`) for which the scanlet does not signal rejection, adorned with the additional information
  - `Nothing` when the scanlet reports an exact fit, i.e., when `cs` contains a complete token with its boundary, resp.
  - `Just s` when the scanlet considers another token `s` as "identified".
- If there is precisely one exact fit, this is used.
- In the absence of perfect fits,
  - if there is exactly one loose fit, this is used,
  - if there is no loose fit either, the key is rejected,
  - otherwise the key is accepted into the un-commited state.
- Also if there are several exact fits, the key is accepted into the un-commited state.

This seems to work satisfactorily on the small examples we have tried, but might need adjustment upon further investigation.

```
> firstsFilter :: String -> [First s] -> [(First s,Maybe String)]
> firstsFilter [] = map (\fi -> (fi,Just []))
> firstsFilter cs = foldr h []
>  where h fi@(scl,_,_) fis =
>           case scl cs of
>             Nothing -> fis
>             Just (s,_) -> (fi, if cs == s then Nothing else Just s) : fis
```

```
> firstsReact :: [First (Either SPair b)] -> Key -> SPair -> Reaction (Either SPair b)
> firstsReact firsts k p =
>  case k of
>   KeyChar c ->
>     let p' = lpInsert c p
>         cs = lpExtract p'
>         firsts' = firstsFilter cs firsts
>         test1 (o,m) = case m of Nothing -> True; _ -> False
>         (full,start) = partition test1 firsts'
>      in case full of
>           [((scl,str,f),_)] -> (f cs,Nothing,False)
>           [] -> case start of
>                   [((scl,str,f),Just s)] -> (f s,Nothing,False)
>                   [] -> (Left p,Just k,False)
>                   _ -> (Left p', Nothing, False)
>           _ -> (Left p', Nothing, False)
>   BackSpace -> tupd3_1 Left $ lpBackSpace p
>   _ -> (Left p,Just k,False)
```

Once the desired semantics is settled, there is of course also considerable room for improvements with respect to efficiency here.

7 The Editor and Editor State Data-Types

The editor data-type is defined as a record providing all the primitive editor functionality; some of the other functions presented above are derived from these. The record fields may be devided into groups as follows:

The **navigation group** comprises proper navigation functions that change the current position, and predicates checking properties of the current position:

```
> data Editor s a = Editor
>   {eRoot, eLeft, eRight, eUp, eDown :: s -> s
>   ,eLeftmost, eRightmost, eTopmost, eBottommost :: s -> Bool
```

The **value group** has been discussed extensively in Sect. 2:

```
>   ,eBuild :: a -> s
>   ,eExtract :: s -> a
>   ,eEmpty :: s
```

The **location group** will be discussed in the next section:

```
>   ,eIsGroup :: s -> Bool
>   ,eSeqWidth :: s -> Int
>   ,eFocus :: Loc -> s -> s
>   ,eLoc, eCharLoc :: s -> Loc
```

The **parsing group** contains as essential components the parser and the scanning aid `eFirsts`. Furthermore there is the function `eAlternatives`, which is derived from `eFirsts` and returns a list of alternatives that are open from the current state, where every alternative is represented by its token name (the second component of `First`) together with the state the editor will be in if that alternative is chosen. Finally, `eTokNames` delivers a list which is non-empty only when the current focus is inside token editors or un-committed alternatives, and then indicates the currently acceptable token classes, also via the second components of the respective `Firsts`:

```
>   ,eParser :: Parser s
>   ,eFirsts :: [First s]
>   ,eAlternatives :: s -> [(String,s)]
>   ,eTokNames :: s -> [String]
```

Closely related to the parsing group is the **reaction** function; since its scope may however be much larger, we list it separately:

```
>    ,eReact :: Key -> React s
```

Finally, the **display group** serves the connection with the GUI system; it contains a function `eLook` to produce a view of the sub-component which is currently in focus, and `eView` to produce a view of the whole editor state:

```
>    ,eLook :: InLoc -> s -> ViewList
>    ,eView :: InLoc -> s -> ViewList
>    }
```

The types of `eLook` and `eView` essentially stem from the structure editor of Sufrin, de Moor and Sage [SdMS00], and the GUI side functionality of `editorComponent` (mentioned at the end of Sect. 5) is an adapted version of their GUI machinery. In a future version, an attractive generalisation will be to replace the editor combinator interface to the type `ViewList` by a suitable class interface, such that these two fields would have the following types:

```
<    ,eLook :: EView v => InLoc -> s -> v
<    ,eView :: EView v => InLoc -> s -> v
```

The class `EView` then had to support an interface that would correspond very closely to that of pretty printing combinator libraries [Hug95, Wad98].

The full definition of the editor combinators mainly consists of "plumbing" of all these components, see appendix A.

In Sect. 4 we have hidden the *sum editor state* used in editors constructed by the alternative `<|>`; after the discussion in the Sect. 6 it should now be obvious that the natural choice is the following:

```
> type SES x y = Either SPair (Either x y)
```

The left component serves for the un-committed state, and the right keeps a committed alternative (which can be uncommitted again via the delete key).

Recall that the internal state of editors constructed with `eGroup` is given by:

```
> type GES s = (s, Bool)
```

If we were to mimic the `Brackets` of [SdM99], we could use

```
< type GES z = (z, Maybe Bool)
```

with `Nothing` taking the rôle of `False`; the `Just` alternatives then would indicate whether the whole group is the current sub-structure, or only one step on the path. Thus we would easily get access to much more powerful editing operations.

8 Locations

For graphical interaction with the editor state we need a concept of locations. Actually, we need two views on our concept of locations, as presented in the first subsection. After that we discuss the location handling in the two combinators most directly involved with locations, namely grouping and pairing.

Outside and Inside Locations

As usual, we use lists of integers as the standard location type:

```
> type Loc = [Int]
```

Our locations are top-down, and the levels are generated by grouping. Furthermore, we distinguish between groups and other components in the following way:

- The function

  ```
  > eIsGroup :: Editor s a -> s -> Bool
  ```

  may be used to determine whether the top-level structure of the current state of an editor is a grouping.

- In a grouping, the location `(i:is)` refers to a part (as indicated by `is`) of the `i`-th logical sub-unit of the *contents of the grouping*.

- If an editor state is not a grouping, it is considered to represent a *sequence of logical units*, and `(i:is)` refers to a part (as indicated by `is`) of the `i`-th logical unit in that sequence.

For determining the length of the sequence represented by non-group editor states, which may essentially result from arbitrary nestings of `<*>`, we use

```
> eSeqWidth :: Editor s a -> s -> Int
```

which returns `1` for group editor states.

The function

```
> eLoc :: Editor s a -> s -> Loc
```

determines the location of the current focus inside an editor state; it presents a location identifying a component of the edited value *from outside*. Therefore we call elements of type `Loc` *outside locations*.

Note that our locations currently identify components of the edited *value*, not of the visible string *representation* of that value. This implies that those parts of the string representation that correspond to trivial components of the value and are *hidden* via one of the combinators `<*<` and `>*>` do *not have a location* of their own, but are considered to be decoration of the grouping that immediately contains them.

A variant of the outside location also respects position inside a token:

```
> eCharLoc :: Editor s a -> s -> Loc
```

This will differ from the result of `eLoc` at most in an additional last element.

Both variants are therefore appropriate as input for the function `eFocus` which, regardless of the present current position, modifies the editor state to focus on the selected position:

```
> eFocus :: Editor s a -> Loc -> s -> s
```

After focusing on an existing location, the location is the one focused on:

```
< eLoc     e (eFocus e loc s) = loc  -- if loc is legal for s
< eCharLoc e (eFocus e loc s) = loc  -- if loc is legal for s
```

Focusing does not change the represented value:

```
< eExtract e (eFocus e loc s) = eExtract e s
```

Focusing on the current character location does not even change the editor state:

```
< eFocus e (eCharLoc e s) s = s
```

In contrast to these outside locations, *inside locations* comprise location information that is passed from the outside to component editors. This kind of information is needed for local actions that need "their position" inside the whole, for example to enable event bindings referring to the current location.

In order to enable inner editors to pass on inside location in such a way that every component editor "knows" which outside location refers to it, we have to include information about the grouping status and whether we are in a hidden branch.

An element `(rloc,mi,hidden)` of the type `InLoc` of inside locations

```
> type RLoc = [Int]
> type InLoc = (RLoc, Maybe Int, Bool)
```

is passed to an editor from the outside with the following meaning:

- `rloc` is the reverse of the outside location of the immediately enclosing group.
- `hidden` is `True` iff the current component is inside a hidden branch.

- `mi` is `Nothing` if the current component is hidden, or if it is the immediate contents of a grouping where it was not `Nothing`, and `mi` is `Just i` if the current component is the `i`-th component in the sequence of local sub-units making up the immediately enclosing grouping.

The topmost inside location therefore is the following:

```
> topInLoc = ([],Nothing,False)
```

The following auxiliary function is used on entry into hidden components, which are assigned to the enclosing group, ignoring the eventual index position in the sub-unit sequence:

```
> hideInLoc (rloc,mi,_) = (rloc, Nothing, True)
```

Finally, whenever a component receives an inside location, it may translate it into the outside location of itself inside the global context using the function `loc1ToLoc` — if a component is inside a hidden structure, it belongs to the grouping indicated by the main location; otherwise it has to take the sequence index into account:

```
> inLocToLoc (rloc, mi, hidden) = let rl' = case mi of Nothing -> rloc
>                                                      Just i -> i : rloc
>                                 in reverse rl'
```

If the support for second-order polymorphism was extended appropriately (see Sect. 10), we would introduce a single inside-location-propagation function

```
> eUseInLoc :: Editor s a ->
>              (forall s' b . Editor s' b -> InLoc -> s' -> r) ->
>              InLoc -> s -> r
```

for which the following law would hold:

```
< eLoc e s = eUseInLoc e ( e loc s -> inLocToLoc loc) topInLoc s
```

Location Handling in Groupings

The trivial facts about a grouping are that it is a grouping, and therefore represents exactly one logical unit:

```
>   ,eIsGroup  = const True
>   ,eSeqWidth = const 1
```

The location-relevant components of `eGroup` therefore have to check whether the contents of the group is again a group, since then one level of the location has to be consumed resp. produced.

For focusing, we omit the check whether the consumed index is in fact `1`, which is the only correct possibility here:

```
>   ,eFocus    = (\ loc (x,_) -> case loc of
>                        [] -> (eRoot e x, True)
>                        (i : is) -> if eIsGroup e x
>                                    then (eFocus e is x,False)  -- assuming i == 1
>                                    else (eFocus e loc x,False))
```

For producing the external location we use a common auxiliary function that inserts a `1` for groupings:

```
>  mkloc0 :: s -> Loc -> Loc
>  mkloc0 x l = case l of [] -> [1]
>                         (i:is) -> if eIsGroup e x then 1 : l else l
```

`eLoc` and `eCharLoc` are then built in the same way — further abstraction would require the kind of second-order polymorphism discussed in Sect. 10.

```
>   ,eLoc     = (\ (x,top) -> if top then [] else mkloc0 x (eLoc     e x))
>   ,eCharLoc = (\ (x,top) -> if top then [] else mkloc0 x (eCharLoc e x))
```

For inside locations in groupings, we use the following auxiliary function:

```
>  contloc (rloc, mi     , True ) = (  rloc, Nothing, True )
>  contloc (rloc, Nothing, False) = (  rloc, Just 1 , False)
>  contloc (rloc, Just i , False) = (i:rloc, Nothing, False)
```

This definition is justified by the fact that, for polymorphic functions

```
> h ::  forall s' b .  Editor s' b -> InLoc -> s' -> r
```

the ficticious location propagation function `eUseInLoc` would have the following definition in the case of `eGroup`:

```
< eUseInLoc (eGroup e) h iloc s = eUseInLoc e h (contloc iloc) s
```

Location Handling in Pairings

Pairings are definitely not groupings, and have as their width the sum of the widths of their components:

```
>   ,eIsGroup = const False
>   ,eSeqWidth = (\ ((x,y),r) -> eSeqWidth e1 x + eSeqWidth e2 y)
```

For pairings, we assemble the auxiliary function for external locations from two sub-functions, and for the second we already expect the width `wl` of the first editor state component `x` as an argument:

```
>  mkloc wl x y r l1 l2 = if r then mkloc2 wl y l2 else mkloc1 x l1
```

In both cases we have to do the group checking again — probably a more elegant setup can be found that moves this level insertion into `eGroup`:

```
>  mkloc1 :: x -> Loc -> Loc
>  mkloc1 x l = if eIsGroup e1 x then 1 : l else l
```

For the second component, the width `wl` of the first component is needed to serve as sequence index offset:

```
>  mkloc2 :: Int -> y -> Loc -> Loc
>  mkloc2 wl y l = let l' = if eIsGroup e2 y then 1:l else l
>                  in case l' of [] -> [wl + 1]
>                                (i:is) -> wl+i : is
```

`eLoc` and `eCharLoc` again use this in the same way:

```
>   ,eLoc     = (\ ((x,y),r) -> let wl = eSeqWidth e1 x
>                                   l2 = eLoc e2 y
>                                   l1 = eLoc e1 x
>                               in mkloc wl x y r l1 l2)
>   ,eCharLoc = (\ ((x,y),r) -> let wl = eSeqWidth e1 x
>                                   l2 = eCharLoc e2 y
>                                   l1 = eCharLoc e1 x
>                               in mkloc wl x y r l1 l2)
```

Focusing has to reassemble the argument location for the right component if that is not a grouping; in the grouping cases, one level has to be dropped:

```
>   ,eFocus = (\ l ((x,y),r) -> case l of
>                     [] -> ((eFocus e1 [] x, y), False)
>                     (i:is) -> let wl = eSeqWidth e1 x
>                               in if i <= wl
>                                  then let l' = if eIsGroup e1 x then is
>                                                else l
>                                       in ((eFocus e1 l' x, y), False)
>                                  else let l' = if eIsGroup e2 y then is
>                                                else i-wl : is
>                                       in ((x,eFocus e2 l' y), True)
>             )
```

Conversely for the auxiliary functions for inside locations, which strictly speaking need not expect `Nothing` for the `mi` component:

```
>  contloc1 iloc@(rloc, mi, hidden) =
>     if hidden then (rloc, Nothing            , True )
>               else (rloc, Just (maybe 1 id mi), False)

>  contloc2 (rloc, mi, hidden) x =
>     if hidden then (rloc, Nothing                                , True )
>               else (rloc, Just $ eSeqWidth e1 x + maybe 1 id mi, False)
```

9 Combining Editors for Recursive Types

The keys to constructing Editors for recursive types are `eRec` and `eMap`. First of all we have to identify the functor behind the recursive data type — in the case of lists this is the following:

```
> type ListF a l = Either (a,l) ()
```

Now it is possible to build an editor combinator for this functor (the state type will be explained later):

```
> listFEditor :: Editor x a -> Editor y l -> Editor (ListEditF x y) (ListF a l)
> listFEditor eA eL = (eA <*> eL) <|> trivEditor
```

Then we need abstraction (`listA`) and representation (`listR`) isomorphisms for the recursive type in question:

```
> listA :: ListF a [a] -> [a]
> listA (Left (x,xs)) = x:xs
> listA (Right _) = []

> listR :: [a] -> ListF a [a]
> listR [] = Right ()
> listR (x:xs) = Left (x,xs)
```

Using these as arguments to `eMap`, we construct an editor combinator that, when partially applied to an element editor, returns an editor transformer for the recursive type:

```
> listFEditor1 :: Editor x a -> (Editor y [a] -> Editor (ListEditF x y) [a])
> listFEditor1 eA eL = eMap listA listR $ listFEditor eA eL
```

It would now be nice if we could directly use `eRec`, with `ListEditF x` instantiated for `c`, and define:

```
< -- ficticious!!
< listEditor1 eA = eRec (listFEditor1 eA)
```

However, there is currently no Haskell extension that treats `c` in the type of `eRec` as a second-order type variable — this is closely related to the restriction that type synonyms must not be partially applied.

For being able to use `eRec`, we therefore first have to identify the state type of the editors involved:

```
> type ListEditF x l = SES (PES x l) ()
```

Then we have to wrap `ListEditF` in a `newtype`:

```
> newtype ListEditFF x y = ListEditFF (ListEditF x y)
> unListEditFF (ListEditFF x) = x
```

This allows us to use `eMapState` to create a variant of `listFEditor1` where the type variable `y` is a proper argument to a real type constructor application, which implies that the partial application of this combinator may be used as argument for `eRec`:

```
> listFEditor2 :: Editor x a -> Editor y [a] -> Editor (ListEditFF x y) [a]
> listFEditor2 eA eL = eMapState ListEditFF unListEditFF $ listFEditor1 eA eL

> listEditor2 :: Editor x a -> Editor (DRec (ListEditFF x)) [a]
> listEditor2 eA = eRec (listFEditor2 eA)
```

In view of all the effort necessary to arrive at the application of `eRec` one may as well consider to directly create a recursive type for the editor state, and then feed the isomorphisms into `eMapState`, thus arriving at the editor combinator for the recursive type without use of `eRec`:

```
> data ListEdit x = ListEdit (ListEditF x (ListEdit x))
> unListEdit (ListEdit x) = x

> listEditor0 :: Editor x a -> Editor (ListEdit x) [a]
> listEditor0 eA = eL
>   where eL = eMapState ListEdit unListEdit $ listFEditor1 eA eL
```

We hope that future language extensions will enable direct use of `eRec`, allowing production of editor combinators for recursive types without having to identify the type constructors for the editor states.

Since usually we want a list to appear as one unit with many children, we group them on the top level:

```
> listEditor :: Editor x a -> Editor (GES (ListEdit x)) [a]
> listEditor = eGroup . listEditor0
```

The most natural alternative, namely inserting `eGroup` into `listEditor1`, would yield an interface reflecting the cons-structure of lists, where the n-th element of a list is n levels below the root.

10 More Second-Order Polymorphism Needed!

One drawback of our approach is that the values represented by the sub-editors at different positions may be of many different types, so that a function to retrieve such a value would have an existentially quantified result type: `selected :: Editor s a -> s -> exists b . b`. Such a result would be very difficult to process further.

The same holds for the internal states of sub-editors.

One approach to solve this kind of problem in a general way is to use universally quantified types. The following function takes as arguments an editor `e` and a generic function `h` that may be applied to editors with arbitrary state and value types, but always returns a result of some fixed type `r`; it then returns a function that acts on internal states of `e` and returns as result (of type `r`) the result of applying `h` to the currently selected sub-editor and its corresponding sub-state.

```
> eCurrent :: Editor s a ->
>             forall r. ((forall s' b . Editor s' b -> s' -> r) -> s -> r)
```

It is possible to declare this function in GHC, but it is not possible to define it as necessary with the current type system since its polymorphic argument function needs to be passed to sub-editors of `<*>` etc., and this would imply `forall`s in result types.

The actual definitions would be rather straightforward — the primitive editors are always their own selection:

```
< eCurrent e@(tagEditor str) h x = h e x
< eCurrent e@(tokenEditor title str scl) h x = h e x
```

For sequential composition we have to descend into the selected component, even in the decoration variants where the decoration component is inaccessible by navigation:

```
< eCurrent   (e1 <*> e2) h ((x,y),r) = if r then eCurrent e2 h y else eCurrent e1 h x
< eCurrent e@(e1 <*< e2) h   x       =                                eCurrent e1 h x
< eCurrent e@(e1 >*> e2) h     y     =           eCurrent e2 h y
```

For `eGroup`, the whole editor is the argument of `h` if the current position focuses on the whole group:

```
< eCurrent e@(eGroup e') h p@(x,top) = if top then h e p else eCurrent e' h x
```

Mapping is again trivial:

```
< eCurrent   (eMap      f g e)     = eCurrent e
< eCurrent   (eMapState f g e) h x = eCurrent e h $ f x
```

The only combinator that might be perceived as non-straightforward is the alternative, since in its uncommitted state there is no component editor related to its state — but since in that state the location is always top, the argument function has to be applied to the whole editor again:

```
< eCurrent   (e1 <|> e2) h (Right (Left  x)) = eCurrent e1 h x
< eCurrent   (e1 <|> e2) h (Right (Right y)) = eCurrent e2 h y
< eCurrent e@(e1 <|> e2) h s@(Left lp) = h e s
```

Obviously, this use of second-order polymorphism is not very involved, since we just pass polymorphic functions along until they are finally applied. If second-order polymorphism is included in some future standardisation of Haskell, we would therefore plead to make also this kind of application possible. In view of the current justification for prohibiting such uses, namely feasibility of type inference, we would gladly put type annotations wherever needed in order to be able to freely handle polymorphic functions.

11 Changing the Behaviour

It is easy to define a *preemptive pairing combinator* `<*!>` which allows to type the second component before the first:

```
> (<*!>) :: Editor x a -> Editor y b -> Editor (PES x y) (a,b)
```

Let us first look into its application. For this purpose we slightly modify the expression editor of Sect. 5, by using a preemptive pairing before the binary operator:

```
> mkBinEdit0 e = (tagEditor "(" >*> e) <*!> opEditor <*> (e <*< tagEditor ")")
> mkExprEdit0 e = identEditor <|> intEditor <|> eGroup (mkBinEdit0 e)
```

Since this does not change any of the types involved we can finish the definition of the expression editor exactly as before:

```
> mkExprEdit1 = eMap exprA exprR . mkExprEdit0
> exprEdit = e where e = eMapState ExprEditState unExprEditState $ mkExprEdit1 e
```

If the user just types "`+-*/12a+3b*4c/5d`" into this editor, the resulting display will be "`((((12/a)*(3+b))-(4*c))+(5/d))`". This is implemented simply by extracting the `eFirsts` of the second component and prepending them (after adaptation) to the `eFirsts` of the pair editor:

```
> e1 <*!> e2 = let e' = e1 <*> e2
>                  h y = ((eEmpty e1,y),False)
>                  f2 = mapFirsts ('_':) h $ eFirsts e2
>              in e' {eFirsts = f2 ++ eFirsts e'}
```

Note that the letters jump into the next component by themselves, since the integer scanlet rejects them and the reaction in pairings passes rejected keys on to the right.

Together with the list editor combinator from Sect. 9 we then construct an editor that allows to edit one expression per line:

```
> exprListEditor :: Editor (GES (ListEdit ExprEditState)) [Expr]
> exprListEditor = listEditor (exprEdit <*< tagEditor "\n")
```

Here, keys that are "superfluous" on the expression level flow over into the next line to build or modify an expression there.

Note that currently there is no means to insert expression in-between existing elements of the list, since this requires locations to be refined into the "brackets" of [SdM99], which we have not yet implemented.

This expression list editor is just a small example to demonstrate how already now we can achieve interesting effects with little effort.

12 Conclusion and Future Work

In parallel to the move from parser generators to parser combinators we have shown that also the move from editor generators to editor combinators is feasible. Confronted with the more complex requirements on the lexical level we have introduced the abstraction of scanlets that brings comfort and flexibility to the user interface, and at the same time continues to function as the basis for the parser, that, of course, has to be combined alongside the combined editor.

Although our first prototype is now working, there is still a lot of work to be done. A straight-forward extension will be to add a clipboard. Allowing more powerful editing actions as with the `Bracket` type of [SdM99] should not be difficult either, as pointed out in Sect. 7. It should also be possible to transfer (and abstract) further features of their editor, such as precedence and associativity integration.

Other features of structure editors, as can be found e.g. in [RT89], will surely make their way into future generations of editor combinators, which for this purpose might want to integrate the first-class attribute grammars of [dM99].

Browser Combinators

A browser can be seen as an editor where either there can be no changes to the represented value, or where the value may be changed by navigation, such that the equations

```
> eExtract e . eRight e = eExtract e
```

etc. do not necessarily hold anymore.

Therefore our editor combinators can also be used as *browser combinators*, and interesting new applications open up, such as intercepting certain kinds of links or monitoring certain kinds of activities.

Our editor combinator library is of course work in progress; a snapshot of a reasonably current state will be kept at `http://ist.unibw-muenchen.de/EdComb/`.

Acknowledgements. We would like to thank Oege de Moor and Meurig Sage for giving out the code of their structure editor and of FranTk. We also would like to thank Lothat Schmitz for his comments on a draft version of this paper.

References

[dM99] Oege de Moor. First-class attribute grammars. Available via URL: `http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`, 1999.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *J. Functional Programming*, 8(4):437–444, July 1998.

[HPJW+92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992. See also `http://haskell.org/`.

[Hue97] Gérard Huet. The Zipper. *J. Functional Programming*, 7(5):549–554, 1997. Functional Pearl.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer, 1995.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Lei00] Daan Leijen. Parsec, a fast combinator parser, 2000. URL: `http://www.cs.uu.nl/~daan/parsec.html`.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[RT89] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Texts and Monographs in Computer Science. Springer, 1989.

[SAA99] S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. In Jan Pavelka, Gerard Tel, and Miroslav Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, 1999, November*, volume 1725 of *LNCS*, pages 111–129, 1999.

[Sag99] Meurig Sage. FranTk – a declarative GUI system for Haskell, 1999. URL: `http://haskell.cs.yale.edu/FranTk/`.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Second International Summer School on Advanced Functional Programming*, volume 1126 of *LNCS*, pages 184–207. Springer, 1996.

[SdM99] Bernard Sufrin and Oege de Moor. Modeless structure editing. In A. W. Roscoe and J.C.P. Woodcock, editors, *Proceedings of the Oxford-Microsoft symposium in Celebration of the work of Tony Hoare, September 13–15, 1999*. 1999.

[SdMS00] Bernard Sufrin, Oege de Moor, and Meurig Sage. An extensible structure editor. 2000. Available via URL: `http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/`.

[Wad85] Philip Wadler. How to replace failure by a list of successes — A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture*, LNCS, pages 113–128. Springer, 1985.

[Wad98] Philip Wadler. A prettier printer, 1998. Available via `http://cm.bell-labs.com/cm/cs/who/wadler/topics/recent.html`.

[Wil92] Ross N. Williams. *FunnelWeb User's Manual*, May 1992. Part of the FunnelWeb distribution, available at `http://www.ross.net/funnelweb/`.

Appendix

The current document has been written in FunnelWeb [Wil92], a language-independent literate programming system with support for the generation of several files from a single document. The Haskell code that can be retrieved from the web site indicated above has been produced from this document via the *code tangling* capabilities of FunnelWeb.

The standard document output of FunnelWeb includes detailed information where every code chunk is headed — we have disabled this information for the sake of readability.

In this appendix, however, we enable just the display of FunnelWeb *macro calls* inside code chunks, so that items like "*MacroName*[123]" appear inside code chunks to indicate that at this position the code chunk named *MacroName* and with first chunk number 123 is included in the Haskell output.

Although these names and numbers do not appear in their defining positions, it should usually be clear from the context which code chunks are referred to, and the items like "*MacroName*[123]" only serve as a hint.

A Implementation of the Editor Combinators

Here we fill in the remainder of the full definitions of our editor combinators, which would only have detracted from the most interesting points presented in the body of this report. A few simple auxiliary functions that we use in several places can be found in our prelude extensions in appendix B.3.

A.1 Display Tools

Since we still re-use much of the display machinery of the structure editor of [SdMS00], we also use the following central type for display chunks:

```
> type TreeView =  Listener TreeTag -> Listener TreeTag -> Int -> [Structured]
```

Since we currently do not use the `Int` designed for indentation, we have to insert ad-hoc zeros in certain places.

For administrative purposes we enrich `TreeViews` with further information, which currently is the tag of the first token if any:

```
> type View = (TreeView,Maybe String)
```

We also need a correspondingly enriched list variant:

```
> type ViewList = ([TreeView],Maybe String)
```

Concatenation of `ViewLists` passes on the first token of the whole:

```
> concViewList :: ViewList -> ViewList -> ViewList
> concViewList (v1,ms1) (v2,ms2) = (v1++v2, maybe ms2 Just ms1)
```

More importantly, `ViewLists` also may need to be flattened to `Views`:

```
> flattenTreeViews' :: ViewList -> View
> flattenTreeViews' (v,ss) = (tv,ss)
>  where
>   tv lst lst' n = concat (map (($n) . ($lst') . ($lst)) v)
```

This happens mostly for the purpose of tagging them as a single unit, which only makes sense if the list is non-empty:

```
> flattenTreeViews :: TreeTag -> ViewList -> View
> flattenTreeViews tag (v,ss) = (tv,ss)
>  where
>   (tv0,_) = flattenTreeViews' (v,ss)
>   tv lst lst' n = case tv0 lst lst' n of
>       [] -> []
>       structs -> [STextTagged structs (mkEditTag [useIdent (identify tag)]) ]
```

Since the string contents of `Structured` text can be retrieved via `structuredToString` (see page 36) without reference to any further `GUI` components (the listener arguments to the `TreeViews` are used only for non-text components of `Structured`), we can implement `eShow` via `eView`:

```
> eShow :: Editor s a -> s -> String
> eShow e x = let tvs = eView e topInLoc x
>                 structs = fst (flattenTreeViews' tvs) undefined undefined 0
>             in concat (map structuredToString structs)
```

`eLook'` is used mainly for debugging purposes and returns the string representation of the selected component:

```
> eLook' :: Editor s a -> s -> String
> eLook' e x = let dloc = eLoc e x
>                  tag = tagOfLoc dloc
>                  tvs = eLook e topInLoc x
>                  (tv,_) = flattenTreeViews' tvs
>                  structs = tv undefined undefined 0
>              in concat (map structuredToString structs)
```

The main view function that is called from the GUI is the following:

```
> viewEditor :: Editor s a -> s ->
>               Listener TreeTag -> Listener TreeTag -> [Structured]
> viewEditor e x lst lst' =
>   let (tv,_) = flattenTreeViews' (eLook e topInLoc x)
>   in tv lst lst' 0
```

A.2 Auxiliary Functions

For first experiments we provide a simple reaction wrapper that translates cursor keys into navigation functions, special character keys like `Space` and `Return` into standard `KeyChar` keys, and ignores modifier key presses:

```
> stdArrowReact :: Editor s a -> KeyReact s -> KeyReact s
> stdArrowReact e r k x = case k of
>    CursorUp    -> if eTopmost e x
>                   then (x,  Just k, undefined)
>                   else (eUp e x, Nothing, undefined)
>    CursorDown  -> if eBottommost e x
>                   then (x,  Just k, undefined)
>                   else (eDown e x, Nothing, undefined)
>    CursorRight -> if eRightmost e x
>                   then (x,  Just k, undefined)
>                   else (eRight e x, Nothing, undefined)
>    CursorLeft  -> if eLeftmost e x
>                   then (x,  Just k, undefined)
>                   else (eLeft e x, Nothing, undefined)
>    Return      -> r (KeyChar '\n') x
>    Space       -> r (KeyChar ' ') x
>    Alt         -> noreact x
>    Control     -> noreact x
>    Shift       -> noreact x
>    Caps        -> noreact x
>    _           -> r k x
>  where noreact s = (s,Nothing,False)
```

A very simple auxiliary editor combinator allows to turn such a reaction wrapper into an editor wrapper:

```
> eBind :: (KeyReact s -> KeyReact s) -> Editor s a -> Editor s a
> eBind rr e = e {eReact = rr (eReact e)}
```

For extracting `eAlternatives` from `Firsts` we use the following, somewhat ad-hoc definition:

```
> fToA :: First s -> (String,s)
> fToA (sc,s,f) = (s,f "")
```

This can be used directly to generate default alternatives for any editor:

```
> eDefaultAlternatives :: Editor s a -> [(String,s)]
> eDefaultAlternatives e = map fToA $ eFirsts e
```

Finally we need a few derived navigation functions which will help in the definitions of the primitive navigation functions of combined editors:

```
> eLeftEnd     z x = if eLeftmost  z x then x else eLeftEnd  z (eLeft  z x)
> eRightEnd    z x = if eRightmost z x then x else eRightEnd z (eRight z x)
> eUpToTop     z x = if eTopmost   z x then x else eUpToTop  z (eUp    z x)
> eRootDefault z x = eLeftEnd z (eUpToTop z x)
```

A.3 Token Editors

The `LPair` manipulation functions mentioned in Sect. 6 are defined as follows:

```
> lpEmpty = ([],[])
> lpIsEmpty (xs,ys) = null xs && null ys
> lpLeft p@([],_) = p
> lpLeft (x:xs, ys) = (xs, x:ys)
> lpRight p@(_,[]) = p
> lpRight (xs, y:ys) = (y:xs, ys)
> lpExtract (xs,ys) = reverse xs ++ ys
> lpMake i = pupd1 reverse . splitAt i
> lpBuild l = (reverse l,[])
> lpLength (xs,ys) = length xs + length ys
> lpStart p = ("",lpExtract p)
> lpLeftmost = null . fst
> lpRightmost = null . snd
> lpInsert x (xs,ys) = (x:xs, ys)
```

Further `LPair` functions more specifically oriented towards their application in our editors are the following:

```
> lpLoc = (:[]) . length . fst
> lpFocus [] = lpStart
> lpFocus (i:_) = lpMake i . lpExtract
```

Since the token editors are responsible for displaying actual text, they are also responsible to implement event bindings on that test and therefore need to know their internal position; the external position extracted from this is used to bind mouse events in `crown` resp. `crown'` (see page 35), where `crown` is used in hidden components and `crown'` adds an identifying tag to tokens that represent a non-trivial part of the edited data-structure.

```
> inLocStringView loc@(_,_,hidden) s =
>    let tag = tagOfLoc (inLocToLoc loc)
>        mycrown = if hidden then crown else crown'
>    in if null s then ([],Nothing) else ([mycrown s tag],Just tag)
> lpLook loc x = inLocStringView loc (lpExtract x)
> lpView loc x = inLocStringView loc (lpExtract x)
```

A token editor is equipped with a `title` string for the token name, used e.g. in choice menus, a `start` string for initialisation, and a `scanlet` for recognising its tokens, and uses mostly the `LPair` and scanlet functions to build an editor from this information:

```
> tokenEditor :: String -> String -> Scanlet -> Editor SPair String
> tokenEditor title start scanlet = e where
>  e = Editor
>   {eRoot = lpStart, eLeft = lpLeft, eRight = lpRight, eUp = id, eDown = id
>   ,eLeftmost = lpLeftmost, eRightmost = lpRightmost
>   ,eTopmost = cTrue, eBottommost = cTrue
>   ,eBuild = lpBuild, eExtract = lpExtract
>   ,eEmpty = eBuild e start
>   ,eIsGroup = cTrue
>   ,eSeqWidth = const 1
>   ,eFocus = lpFocus, eLoc = const [], eCharLoc = lpLoc
>   ,eParser = lpMake 0 <<$>> scanletP scanlet
>   ,eFirsts = [(scanlet, title, lpBuild)]
>   ,eAlternatives = const $ eDefaultAlternatives e
>   ,eTokNames = const [title]
>   ,eReact = scanletReact scanlet
>   ,eLook = lpLook, eView = lpView
>   }
```

The editor for constant strings is a more trivial variant of the `tokenEditor`; we use it as an editor for the type `()`, so that it has no internal state at all. Nevertheless it needs a scanlet for parsing and for interactive selection; for the time being we use the completion scanlet introduced on page 9:

```
> tagEditor :: String -> Editor () ()
> tagEditor str = e where
>  scanlet = completionScanlet str
>  e = Editor
>   {eRoot = id, eLeft = id, eRight = id, eUp = id, eDown = id
>   ,eLeftmost = cTrue, eRightmost = cTrue, eTopmost = cTrue, eBottommost = cTrue
>   ,eBuild = const (), eExtract = const (), eEmpty = ()
>   ,eIsGroup = const True
>   ,eSeqWidth = const $ if null str then 0 else 1
>   ,eFocus = cId, eLoc = const [], eCharLoc = const []
>   ,eParser = const () <<$>> scanletP scanlet
>   ,eFirsts = [(scanlet,str,const ())]
>   ,eAlternatives = const $ eDefaultAlternatives e
>   ,eTokNames = const []
>   ,eReact = (\ k -> (\ (_,mk,b) -> ((),mk,b)) .
>                     scanletReact scanlet k . (const (reverse str,"")))
>   ,eLook = (\ loc x -> inLocStringView loc str)
>   ,eView = (\ loc x -> inLocStringView loc str)
>   }

> trivEditor :: Editor () ()
> trivEditor = tagEditor ""

> constEditor :: a -> String -> Editor () a
> constEditor a s = eMap (const a) (const ()) $ tagEditor s
```

A.4 Grouping

A group represents one level for the definition of locations.

Recall that the state of the group editor is defined to contain a flag which is `True` if the whole group is the current sub-structure, and `False` if the current sub-structure is an internal part of the group:

```
> type GES s = (s, Bool)
> topGroup x = (x,True)
```

```
> inGroup x = (x,False)

> eGroup :: Editor s a -> Editor (GES s) a
> eGroup (e :: Editor s a) = e' where
```
GroupAux[115]
```
>  e' = Editor
>   {eRoot       = pupd (eRoot e) cTrue
>   ,eUp         = (\ p@(x,top) -> if top then p
>                                  else if eTopmost e x then (eRoot e x,True)
>                                       else (eUp e x,False))
>   ,eDown       = (\   (x,top) -> (if top then eRoot e x else eDown e x, False))
>   ,eLeft       = (\ p@(x,top) -> if top then p     else (eLeft e x, top))
>   ,eRight      = (\ p@(x,top) -> if top then p     else (eRight e x, top))
>   ,eLeftmost   = (\   (x,top) -> if top then True  else eLeftmost   e x)
>   ,eRightmost  = (\   (x,top) -> if top then True  else eRightmost  e x)
>   ,eTopmost    = (\   (x,top) -> top)
>   ,eBottommost = (\   (x,top) -> if top then False else eBottommost e x)
>   ,eExtract    = (eExtract e) . fst
>   ,eBuild      = (\ a -> (eBuild e a,True))
>   ,eEmpty      = (eEmpty e, True)
```
GroupComponents[113]
```
>   ,eParser = topGroup <<$>> eParser e
>   ,eFirsts = mapFirsts id inGroup (eFirsts e)
>   ,eAlternatives = (\ (x,top) -> if top
>              then eDefaultAlternatives e'
>              else map (pupd2 inGroup) $ eAlternatives e x)
>   ,eTokNames = (\ (x,top) -> if top then [] else eTokNames e x)
>   ,eReact   = (\ k (x,top) ->
>       case k of
>        Delete | top -> (eEmpty e',Nothing,False)
>        BackSpace | top -> ((eDownR e x, False),Nothing, False)
>        _ -> let (x',mk,b) = eReact e k x
>             in case mk of Nothing -> ((x',False), Nothing, b)
>                           Just k -> ((x',top),mk,b))
>   ,eLook    = (\ loc p@(x,top) ->
>                  if top then eView e' loc p
>                         else eLook e (contloc loc) x)
>   ,eView    = (\ loc (x,top) ->
>                let tag = tagOfLoc $ inLocToLoc loc
>                    loc' = contloc loc
>                    v = eView e loc' x
>                    (tv,ss) = flattenTreeViews tag v
>                in ([tv],ss))
>   }
```

A.5 Product Editors

The *product editor state* `PES x y` contains the two component editor states together with a Boolean indicating whether the current position is in the second component:

```
> type PES x y = ((x, y), Bool)
```

Most of the components of the product editor just have to refer to the active sub-editor; some kind of communication between the two sub-editors happens when `eLeft` and `eRight` cross the border between them, and when a key press inside the active first sub-editor is rejected and passed on to the second sub-editor:

```
> (<*>) :: Editor x a -> Editor y b -> Editor (PES x y) (a,b)
> (e1 :: Editor x a) <*> (e2 :: Editor y b) = let
```
ProductAux[121]
```
>  e = Editor
>   {eRoot = eRootDefault e
>   ,eUp   = (\ ((x,y),r) -> (if r then (x,eUp   e2 y) else (eUp   e1 x,y), r))
>   ,eDown = (\ ((x,y),r) -> (if r then (x,eDown e2 y) else (eDown e1 x,y), r))
>   ,eLeft = (\ ((x,y),r) -> if r
>               then if eLeftmost e2 y
>                    then ((eRightEnd e1 (eUpToTop e1 x), eUpToTop e2 y),False)
>                    else ((x,eLeft e2 y), r)
>               else ((eLeft e1 x, y), r)
>            )
>   ,eRight      = (\ ((x,y),r) -> if r then ((x, eRight e2 y), r)
>                     else if eRightmost e1 x
>                          then ((eUpToTop e1 x, eRoot e2 y), True)
>                          else ((eRight e1 x, y), r)
>                  )
>   ,eLeftmost   = (\ ((x,y),r) -> not r && eLeftmost e1 x)
>   ,eRightmost  = (\ ((x,y),r) -> r && eRightmost e2 y)
>   ,eTopmost    = (\ ((x,y),r) -> if r then eTopmost e2 y else eTopmost e1 x)
>   ,eBottommost = (\ ((x,y),r) -> if r then eBottommost e2 y
>                                       else eBottommost e1 x)
>   ,eBuild = (\(a,b) -> ((eBuild e1 a, eBuild e2 b), False))
>   ,eExtract = (\((x,y),r) -> (eExtract e1 x, eExtract e2 y))
>   ,eEmpty = ((eEmpty e1, eEmpty e2), False)
```
ProductComponents[120]
```
>   ,eParser = (\ x y -> ((x,y),False)) <<$>>
>              eParser e1 `pSeqDft` eParser e2 $ eEmpty e2
>   ,eFirsts = mapFirsts id (\x -> ((x,eEmpty e2), False)) (eFirsts e1)
>   ,eAlternatives = (\ ((x,y),r) ->
>             if r then map (pupd2 (\y -> ((x,y),r))) $ eAlternatives e2 y
>                  else map (pupd2 (\x -> ((x,y),r))) $ eAlternatives e1 x)
>   ,eTokNames = (\ ((x,y),r) -> if r then eTokNames e2 y else eTokNames e1 x)
>   ,eReact = (\ key s@((x,y),r) -> if r
>                 then let (y', mk,b) = eReact e2 key y
>                      in (((x,y'),True),mk,b)
>                 else case eReact e1 key x of
>                       (x',Nothing,b) -> (((x',y),b),Nothing,False)
>                       (x', Just k, True) -- not consumed, but satisfied
>                          -> let (y', mk, b) = eReact e2 k y
>                             in (((x',y'),True),mk,b)
>                       (x', Just k, False) -- not consumed and angry
>                          -> (s, Just k, False)
>             )
>   ,eLook = (\ loc ((x,y),r) -> if r then eLook e2 (contloc2 loc x) y
>                                     else eLook e1 (contloc1 loc  ) x)
>   ,eView = (\ loc ((x,y),r) ->          eView e1 (contloc1 loc  ) x
>                          `concViewList` eView e2 (contloc2 loc x) y)
>   }
>  in e
```
Decoration at the right is completely trivial:
```
> (<*<) :: Editor x a -> Editor () b -> Editor x a
> e1 <*< e2 = Editor
>   {eRoot  = eRoot e1, eUp    = eUp    e1, eDown = eDown e1
```

```
>   ,eLeft  = eLeft e1, eRight = eRight e1
>   ,eLeftmost = eLeftmost e1, eRightmost  = eRightmost  e1
>   ,eTopmost  = eTopmost  e1, eBottommost = eBottommost e1
>   ,eBuild = eBuild e1, eExtract = eExtract e1, eEmpty = eEmpty e1
>   ,eIsGroup = eIsGroup e1, eSeqWidth = eSeqWidth e1, eFocus = eFocus e1
>   ,eLoc = eLoc e1, eCharLoc = eCharLoc e1
>   ,eParser = const <<$>> eParser e1 `pSeqDft` eParser e2 $ ()
>   ,eFirsts = eFirsts e1, eAlternatives = eAlternatives e1, eTokNames = eTokNames e1
>   ,eReact = eReact e1
>   ,eLook = eLook e1
>   ,eView = (\ loc x ->  eView e1 loc x `concViewList` eView e2 (hideInLoc loc) ())
>   }
```

With decoration at the left the only critical point is parsing, i.e., `eParser` together with `eFirsts`. Although the hidden component `e1` does not have a location of its own, because of our deterministic approach to parsing it still is crucial in determining whether the decorated component is to be instantiated in the case of its choice.

```
> (>*>) :: Editor () a -> Editor y b -> Editor y b
> e1 >*> e2 = e where
>  e = Editor
>   {eRoot  = eRoot e2, eUp    = eUp    e2, eDown = eDown e2
>   ,eLeft  = eLeft e2, eRight = eRight e2
>   ,eLeftmost = eLeftmost e2, eRightmost  = eRightmost  e2
>   ,eTopmost  = eTopmost  e2, eBottommost = eBottommost e2
>   ,eBuild = eBuild e2, eExtract = eExtract e2, eEmpty = eEmpty e2
>   ,eIsGroup = eIsGroup e2, eSeqWidth = eSeqWidth e2, eFocus = eFocus e2
>   ,eLoc = eLoc e2, eCharLoc = eCharLoc e2
>   ,eParser = (\ x y -> y) <<$>> eParser e1 `pSeqDft` eParser e2 $ eEmpty e2
>   ,eFirsts = mapFirsts id (const (eEmpty e)) (eFirsts e1)
>   ,eAlternatives = eAlternatives e2, eTokNames = eTokNames e2
>   ,eReact = eReact e2
>   ,eLook = eLook e2
>   ,eView = (\ loc y -> eView e1 (hideInLoc loc) () `concViewList` eView e2 loc y)
>   }
```

A.6 Alternative Composition

Recall that the sum editor state contains either an uncommitted `SPair`, or a state of one of two alternative state types:

```
> type SES x y = Either SPair (Either x y)
```

Most of the components are then straightforward plumbing; the one crucial field, `eReact`, has already been discussed in Sect. 6. The auxiliary function `firstsFilter` defined there is also used to let `eAlternatives` present exactly the currently still possible alternatives.

```
> (<|>) :: Editor x a -> Editor y b -> Editor (SES x y) (Either a b)
> (e1 :: Editor x a) <|> (e2 :: Editor y b) = e where
>  e = Editor
>   {eRoot       = eitheR lpStart     (eitheR (eRoot       e1) (eRoot       e2))
>   ,eUp         = eitheR id          (eitheR (eUp         e1) (eUp         e2))
>   ,eDown       = eitheR id          (eitheR (eDown       e1) (eDown       e2))
>   ,eLeft       = eitheR lpLeft      (eitheR (eLeft       e1) (eLeft       e2))
>   ,eRight      = eitheR lpRight     (eitheR (eRight      e1) (eRight      e2))
>   ,eLeftmost   = either lpLeftmost  (either (eLeftmost   e1) (eLeftmost   e2))
>   ,eRightmost  = either lpRightmost (either (eRightmost  e1) (eRightmost  e2))
>   ,eTopmost    = either cTrue       (either (eTopmost    e1) (eTopmost    e2))
>   ,eBottommost = either cTrue       (either (eBottommost e1) (eBottommost e2))
```

```
>   ,eExtract = either (altExtract . fst . unJust . altParser . lpExtract) altExtract
>   ,eBuild = Right . eitheR (eBuild e1) (eBuild e2)
>   ,eEmpty = Left lpEmpty
>   ,eIsGroup    = either cTrue       (either (eIsGroup    e1) (eIsGroup    e2))
>   ,eSeqWidth   = either (const 1)   (either (eSeqWidth   e1) (eSeqWidth   e2))
>   ,eFocus = (\ loc -> eitheR (lpFocus loc) (eitheR (eFocus e1 loc) (eFocus e2 loc)))
>   ,eLoc        = either (const [])  (either (eLoc        e1) (eLoc        e2))
>   ,eCharLoc    = either lpLoc       (either (eCharLoc    e1) (eCharLoc    e2))
>   ,eParser = Right <<$>> altParser
>   ,eFirsts = mapFirsts id (Right . Left ) (eFirsts e1) ++
>              mapFirsts id (Right . Right) (eFirsts e2)
>   ,eAlternatives = (\ s ->
>        let f1 = map (pupd2 (Right . Left )) . eAlternatives e1
>            f2 = map (pupd2 (Right . Right)) . eAlternatives e2
>        in case s of
>             Left lp -> map (fToA . fst) $ firstsFilter (lpExtract lp) $ eFirsts e
>             Right e -> either f1 f2 e)
>   ,eTokNames = (\ s -> case s of
>              Left lp -> map snd3 $ eFirsts e
>              Right e -> either (eTokNames e1) (eTokNames e2) e)
>   ,eReact = (\ k x -> case k of
>     Delete | eTopmost e x && eSeqWidth e x == 1
>       -> (Left lpEmpty,Nothing,False)
>     _ -> case x of
>           Left p -> firstsReact (eFirsts e) k p
>           Right y -> tupd3_1 Right $ ldistr3 $ eitheR (eReact e1 k) (eReact e2 k) y)
>   ,eLook = (\ loc -> either (lpLook loc) (either (eLook e1 loc) (eLook e2 loc)))
>   ,eView = (\ loc -> either (lpView loc) (either (eView e1 loc) (eView e2 loc)))
>   }
>  altParser = (Left <<$>> eParser e1)  <<|>>  (Right <<$>> eParser e2)
>  altExtract = eitheR (eExtract e1) (eExtract e2)
```

A.7 Mapping

`eMap` is actually the simplest primitive combinator with respect to implementation — we only have to adapt the value interface:

```
> eMap :: (a -> b) -> (b -> a) -> Editor s a -> Editor s b
> eMap f g e =
>   e {eBuild = eBuild e . g
>     ,eExtract = f . eExtract e
>     }
```

While the value type only occurs in these two fields, the state type occurs in *all* fields, so that for `eMapState` we have to insert the adaptation functions everywhere:

```
> eMapState :: (x -> y) -> (y -> x) -> Editor x a -> Editor y a
> eMapState (f :: x -> y) (g :: y -> x) (e :: Editor x a) = Editor
>   {eRoot = f . eRoot e . g, eLeft = f . eLeft e . g, eRight = f . eRight e . g
>   ,eUp   = f . eUp   e . g, eDown = f . eDown e . g
>   ,eLeftmost = eLeftmost e . g, eRightmost  = eRightmost  e . g
>   ,eTopmost  = eTopmost  e . g, eBottommost = eBottommost e . g
>   ,eBuild = f . eBuild e, eExtract = eExtract e . g, eEmpty = f (eEmpty e)
>   ,eIsGroup  = eIsGroup  e . g
>   ,eSeqWidth = eSeqWidth e . g
>   ,eFocus = (\ loc -> f . eFocus e loc . g)
>   ,eLoc      = eLoc      e . g
```

```
>   ,eCharLoc  = eCharLoc  e . g
>   ,eParser = f <<$>> eParser e
>   ,eFirsts = mapFirsts id f (eFirsts e)
>   ,eAlternatives = map (pupd2 f) . eAlternatives e . g
>   ,eTokNames = eTokNames e . g
>   ,eReact = (\ k -> (\ (x,mk,b) -> (f x,mk,b)) . eReact e k . g)
>   ,eLook = (\ loc -> eLook e loc . g)
>   ,eView = (\ loc -> eView e loc . g)
>   }
```

A.8 More Auxiliary Functions

Many more auxiliary functions such as derived navigation operations will be useful in later stages of this development.

In the GUI we need a function to extract the tag of the first token of the currently selected sub-component:

```
> eLocTag e s = snd $ eLook e topInLoc s
```

Most of the functions presented in the following are derived from their zipper equivalents in the structure editor of [SdMS00].

First we list material that needed to be adapted to our different setting.

When calculating sibling locations we have to take into account that our locations are top-down:

```
> locSibling :: Loc -> Loc -> Bool
> locSibling [] [] = False
> locSibling [] _  = False
> locSibling _  [] = False
> locSibling [x] [y] = y > x
> locSibling (x:xs) (y:ys) = x == y && locSibling xs ys
```

```
> sibling :: Editor s a -> s -> Loc -> Bool
> sibling e x = locSibling (eLoc e x)
```

One use of siblings is for a crude approximation of cursor positions when the focus is on an empty sub-editor (which is impossible in [SdMS00]): we look right, and if necessary also left, for a non-empty component, but restrict the search to siblings in the hope that between siblings there are not too many interfering hidden decorations:

```
> leaning :: Editor s a -> s -> Maybe (Either TreeTag TreeTag)
> leaning e s = let
>   leaningR e loc s = let sR = eRight e s
>                          locR = eLoc e sR
>                      in if locSibling loc locR
>                         then case eLocTag e sR of
>                                Nothing -> leaningR e locR sR
>                                Just t -> Just (tagOfLoc locR)
>                         else Nothing
>   leaningL e loc s = let sL = eLeft e s
>                          locL = eLoc e sL
>                      in if locSibling locL loc
>                         then case eLocTag e sL of
>                                Nothing -> leaningL e locL sL
>                                Just t -> Just (tagOfLoc locL)
>                          else Nothing
>   loc = eLoc e s
>  in case leaningR e loc s of
>      Just t -> Just (Right t)
>      Nothing -> fmap Left $ leaningL e loc s
```

The following is from [SdMS00], but as long as we still don't have brackets, it is essentially useless:

```
> bracket0 :: Editor s a -> Loc -> s -> s
> bracket0 e l x = if sibling e x l
>                  then eFocus e l x
>                  else x
```

For the time being, we only carry `()` in the clipboard.

```
> type Clip = ()
> emptyClip = ()

> type Edit0 s = EditAction s Clip

> idEdAct :: EditAction s c
> idEdAct (st,c) = ((st,c),[])

> type EStatus s = EditSt s Clip -- = (s,())
```

The remaining functions stem directly from [SdMS00]:

```
> type Paint a b = a -> (a,[b])  -- a is state, b is update information

> (-.-) :: Paint a b -> Paint a b -> Paint a b
> (-.-) f g s = (u, p++q)
>               where (t,p) = g s
>                     (u,q) = f t

> type EditSt s c      = (s, c)  -- state, clipboard
> type EditAction s  c = Paint (EditSt s c) s

> nav :: (s -> s) -> EditAction s c
> nav f (s,c) = ((f s, c),[])

> eDownR :: Editor s a -> s -> s
> eDownR e = eRightEnd e . eDown e

> eNext :: Editor s a -> s -> s
> eNext e x = if eBottommost e x then eRightup e x else eDown e x

> ePrev :: Editor s a -> s -> s
> ePrev e x = if eBottommost e x then eLeftup e x else eDownR e x

> eRightup :: Editor s a -> s -> s
> eRightup e = eRight e . until (\x -> eTopmost e x || not (eRightmost e x)) (eUp e)

> eLeftup :: Editor s a -> s -> s
> eLeftup e = eLeft e . until (\x -> eTopmost e x || not (eLeftmost e x)) (eUp e)

> eNextSuchThat :: Editor s a -> (s -> Bool) -> s -> s
> eNextSuchThat e p x = if p x' then x' else x
>   where x' = until (\x -> eTopmost e x || p x) (eNext e) (eNext e x)

> ePrevSuchThat :: Editor s a -> (s -> Bool) -> s -> s
> ePrevSuchThat e p x = if p x' then x' else x
>   where x' = until (\x -> eTopmost e x || p x) (ePrev e) (ePrev e x)

> eChildSuchThat :: Editor s a -> (s -> Bool) -> s -> s
```

```
> eChildSuchThat e p x = if p x’ then x’ else x
>   where x’ = until (\x -> eRightmost e x || p x) (eRight e) (eDown e x)

> eChildSuchThatR :: Editor s a -> (s -> Bool) -> s -> s
> eChildSuchThatR e p x = if p x’ then x’ else x
>   where x’ = until (\x -> eLeftmost e x || p x) (eLeft e) (eDownR e x)
```

B The Modules

B.1 The Editor Combinator Module `Editor.lhs`

We collect all the editor combinator machinery into a module of its own; we have not yet strived to make the `Editor` type abstract and export only the combinators and certain access functions, but this will of course be desirable for a more mature version:

--
-- *file comment*[208]
--
```
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--

> module Editor where

> import FranTkUtils
> import FranTk(Key(..), Listener)
> import ComponentWidgets(Structured(..))
> import Scanlet
> import ExtPrel
> import List(partition)

> infixr 4 <*>, <*!>, <*, *>, <*<, >*>
> infixr 3 <|>
```
Editor[17]

B.2 The Scanlet Module `Scanlet.lhs`

Since scanlets are independent from the rest of the editor combinator machinery and might also find uses in other contexts, we collect the scanlet-specific material from Sect. 6 into a module of its own:

--
-- *file comment*[208]
--
```
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--

> module Scanlet where

> import ExtPrel
```

Scanlet[55]

B.3 Prelude Extensions `ExtPrel.lhs`

There are certain prelude-level functions that we use throughout; we collect these together with some basic parsing stuff into one module:

```
--
-- file comment[208]
--
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--

> module ExtPrel where

> infixl 4 <<$>>, `pSeqDft`
> infixl 3 <<|>>

> init' [] = []
> init' l = init l
> last' x [] = x
> last' x l = last l

> cId = const id
> cTrue = const True
> cFalse = const False
```

ExtPrel[75]

```
> data DRec c = DRec (c (DRec c))
> unDRec (DRec x) = x

> unJust (Just x) = x
> unJust Nothing = error "unJust Nothing"

> pupd2 g (x,y) = (x, g y)
> pupd f g (x,y) = (f x, g y)

> fst3  (x,_,_) = x
> snd3  (_,y,_) = y
> thrd3 (_,_,z) = z
> ldistr3 (Left (x,y,z)) = (Left x, y, z)
> ldistr3 (Right (x,y,z)) = (Right x, y, z)
> tupd3_1 f (x,y,z) = (f x, y, z)

> eitheR f g = either (Left . f) (Right . g)
```

Parser type[64]

B.4 A `Main.lhs` for Testing

A full-fledged editor based on our combinators only needs a few definitions like those in Sect. 5, including the definition for `main` seen there, and a few imports.

Here we also include an `import CLEdit` for the command line interface of the next section.

```
--
-- file comment[208]
--
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--

> module Main(main,mkExprEditor1,mkExprEditor0) where
```

```
> import Editor
> import EditorInstance
> import FranTk(display)
> import CLEdit
> import Char
```

exprEditor[32]

```
> run e = display $ editorComponent $ eBind (stdArrowReact e) e

> main :: IO ()
> main = run $ exprListEditor
> -- main = interactEditor exprListEditor Nothing
> -- main = run exprEdit
> -- main = run exprEditor
> -- main =  interactEditor exprEditor Nothing
```

C Command Line Interface for Testing

Since the editor combinators themselves do not rely on any GUI assumptions at all, for testing purposes it is sometimes handy to have a command line interface that eschews the problems frequently related with GUIs.

For quick experiments, we use a `vi`-like key binding to emulate the cursor keys:

```
> charToKey :: Char -> Key
> charToKey 'k' = CursorUp
> charToKey 'j' = CursorDown
> charToKey 'l' = CursorRight
> charToKey 'h' = CursorLeft
> charToKey 'x' = Delete
> charToKey 'b' = BackSpace
> charToKey c   = KeyChar c
```

This allows us to program a character based editor state transformer that additionally emits a string containing information about the editor state:

```
> useKey :: Editor s a -> Char -> s -> (s, String)
> useKey e c s = let s' = eKey e (charToKey c) s
>                in (s', eShow e s' ++ " ||| " ++
>                        show (eCharLoc e s') ++ " " ++
>                        show (eLoc e s') ++ " ``" ++
>                        eLook' e s' ++ "'' ")

> accum s f [] = []
> accum s f (x:xs) = let (s',u) = f x s
>                    in u : accum s' f xs
```

Embedding an editor into `interact` is then completely straightforward:

```
> interactEditor :: Editor s a -> Maybe a -> IO ()
> interactEditor e ma = let e' = eBind (stdArrowReact e) e
>                           init = case ma of
>                                    Nothing -> eEmpty e'
>                                    Just a -> eBuild e' a
>                           react = accum init (useKey e')
>                       in interact $ unlines . react . filter (/= '\n')
```

We package this up in a module `CLEdit`:

```
--
-- file comment[208]
--
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--

> module CLEdit(interactEditor) where

> import Editor
> import FranTk(Key(..))
```

charToKey[192]

D FranTk Utilities

All the FranTk interface that has to be accessible from the editor combinators is collected into the module `FranTkUtils`.

This comprises mostly auxiliary material taken or adapted from the structure editor of [SdMS00].

```
--
-- file comment[208]
--
-- $Id: EdComb.fw,v 1.8 2000/06/16 11:51:20 kahl Exp kahl $
--
-- Material in this file has mostly been taken resp. adapted
-- from the ‘‘extensible structure editor’’ of
-- Bernard Sufrin, Oege de Moor, and Meurig Sage,
-- available via http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/

> module FranTkUtils where

> import FranTk
> import List (intersperse)

> structsMap :: ([Structured] -> [Structured]) -> (Int -> Int) -> TreeView -> TreeView
> structsMap fs fi v lst lst' m = fs $ v lst lst' (fi m)

------ modified material from structedit/ViewType.hs
```

View[153]

LocUtils[98]

```
> crown' :: String -> TreeTag -> TreeView
> crown' x lbl lst lst' n =
>   if null x then []
>   else let structs = crown x lbl lst lst' n
>        in [STextTagged structs (mkEditTag [useIdent (identify lbl)])]
> crown :: String -> TreeTag -> TreeView
> crown x lbl lst lst' n
>   = if null x then []
>     else [STextTagged [SText x] (mousePress 3 (tellL lst' lbl) $
>                                  mousePress 1 (tellL lst  lbl) $
>                                  mkEditTag [foregroundB blue]  )]
```

```
------ unmodified material from structedit/ViewType.hs

-- tree tags

> type TreeTag = String

> tagOfLoc :: Loc -> TreeTag
> tagOfLoc = ('p':) . concat . intersperse "p" . map show

> locOfTag :: TreeTag -> Loc
> locOfTag = map read . numbers
>             where numbers [] = []
>                   numbers [a]  = []
>                   numbers y = y0 : numbers y1
>                               where (y0,y1) = span (/='p') (tail y)

------ modified material from structedit/Structured.hs

> structuredToString :: Structured -> String
> structuredToString (SText s) = s
> structuredToString (STextTagged ss _) = concat (map structuredToString ss)
> structuredToString (STextMark _) = []
> structuredToString (STextGroup ss _) = concat (map structuredToString ss)
```

E Editor Instances

Finally there is a module `EditorInstance` that defines the function

```
> editorComponent :: Editor s a -> WComponent
```

which creates a FranTk GUI component as an instance of its argument editor. This can then be `display`ed, as seen in `Main` above.

Part of this module is again adapted from the file `Structured.hs` of the structure editor of [SdMS00].

Since it does not contribute in an essential way to the understanding of our editor combinators, we do not include it in this document. The editor combinator web site contains a copy of all source files so that experimenting is possible.