Combinators for layered software architectures

Martijn M. Schrage

Johan Jeuring

Doaitse Swierstra

institute of information and computing sciences, utrecht university

technical report UU-CS-2002-030

www.cs.uu.nl

Combinators for layered software architectures

Martijn M. Schrage, Johan Jeuring, Doaitse Swierstra

November 8, 2002

Abstract

We define a domain specific embedded language in Haskell to describe layered software architectures of editors. By using a typed programming language to describe the architecture, the type correctness of its components is guaranteed by the type checker of the language. Furthermore, because the architecture description of a system is part of the implementation of the system, the implementation will always comply with the architecture.

1 Introduction

In this paper we use the functional programming language Haskell to give a domain specific embedded language (DSEL) for describing layered editor architectures [6]. We use the term editor in a broad sense. Text editors and HTML editors are well-known examples, but also spread-sheets, e-mail agents, or even the preferences screens that are present in most applications with graphical user interfaces can be regarded as editors.

Haskell is a functional language with strong support for abstraction. Describing the architecture in an actual programming language not only makes it possible to verify the types of the components of the architecture, but also provides a framework for implementing the system that is described by the architecture. Because the Haskell architecture is a program in itself, the system can be instantiated by providing implementations for each of the components.

In their survey of architecture description languages (ADLs) [4], Medvidovic and Taylor identify three essential components of an architecture description: a description of the (interface of the) components, a description of the connectors, and a description of the architectural configuration. They claim that the focus on *conceptual* architecture and explicit treatment of connectors as first-class entities differentiate ADL's from, amongst others, programming languages. However, Higher-order typed (HOT) functional programming languages, such as Haskell [5], Clean, and ML offer possibilies for describing the main components of an architecture, and treating all of these components as first-class entities. Furthermore, by using abstraction, the description of the architecture can be focused on the conceptual architecture, while the details are left to the actual components.

Higher-order typed functional languages offer excellent possibilities for embedding domain specific languages [2]. Embedding a domain specific language facilitates reuse of syntax, semantics, implementation code, software tools, as well as look-and-feel. We give a DSEL in Haskell for describing layered editor architectures. We use records that contain functions to describe the components of the architecture. The connectors are combinators, and the configurations are programs (functions) that consist of combinators and components.

The requirements for a DSEL for describing architectures are slightly different from the requirements for more familiar applications of DSELs, such as parsers, pretty printers, etc. The latter are used to write programs that contain many applications of the combinators and that are usually subject to frequent change. An architecture description, on the other hand, will typically be a small and rather static entity, because changes to the architecture involve changes to many components of the system. As a consequence, a concise syntax for combinator expressions is not of the highest importance.

Another difference concerns the efficiency of the combinators. Because in an 'ordinary' DSEL the programs that are constructed typically contain many applications of the combinators, the evaluation of the combinators forms a substantial part of the running time. In contrast, the combinators in an architecture description are the glue that connects the main components of the system. Most of the running time of the system can be expected to be taken up by the components and not by the combinators. Therefore, efficiency of the combinators is not a major concern. Wrapping and unwrapping a few values in the combinators is not a problem if it improves the transparency of the architecture.

In this paper we give four implementation models for layered architectures. First, we introduce a simple example layered editor architecture and explore how its main components can be modeled in Haskell. Then we proceed to connect the components. In section 4, the connection is straightforward, with little abstraction. This is used in section 5 as a basis to develop a more abstract combinator implementation that uses nested cartesian products. In section 6, we present another set of combinators, which employ a form of state hiding to improve on the previous set. Section 7 develops a small generic library for building the architecture-specific combinators of section 6. Section 8 contains a short description of the architecture of the Proxima editor [7], for which the combinators were designed, and section 9 presents the conclusions.

2 A Simple editor

In this section, we introduce a simple layered architecture for an editor that is used in the next sections to illustrate the architecture description methods. The system that is

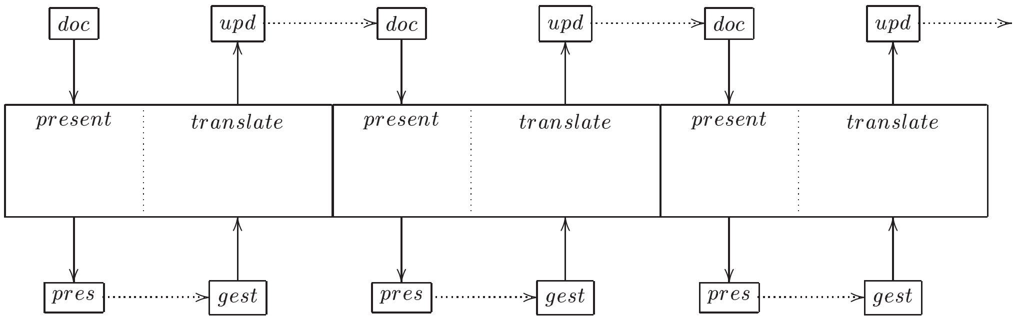

Figure 1: Three cycles in the edit process.

described by the architecture is an editor in which the user can edit a presentation of a document. Although the architecture of the editor is simple, it still contains the typical features of the layered architectures discussed in [6].

The edit process: The main loop of the simple editor consists of the following 5 phases:

1. Compute a rendering, or *presentation*, of the document (presentation phase)
2. Show the presentation to the user
3. Receive an edit gesture from the user
4. Translate the edit gesture to a document update (translation phase)
5. Compute the updated document

Figure 1 contains a schematic representation of the edit process. Each of the boxes represents one complete edit cycle. The emphasis in the figure is put on the presentation and translation phases (phases 1 and 4). Phases 2 and 3 are represented by the dotted arrows at the bottom of the figure, and phase 5 is represented by the dotted arrows at the top.

The layers: Figure 2 shows a more detailed view of a single edit cycle. For now, we focus on the presentation and translation phases (phase 1 and phase 4). Both the presentation and the translation functions (we will call them *present* and *translate*) are compositions of a number of subfunctions, similar to the layered editor in [6]. The presentation of the document is computed via a number of intermediate data structures that are increasingly concrete, and the edit operation on the document is computed via edit operations on the same intermediate data structures. Therefore, the subfunctions of *present* and *translate* are grouped pairwise in horizontal layers. Another reason for grouping the subfunctions is that each subfunction in a layer has some parameters that are the result of the subfunction immediately to the left of it. We will refer to the subfunctions as *layer functions*.

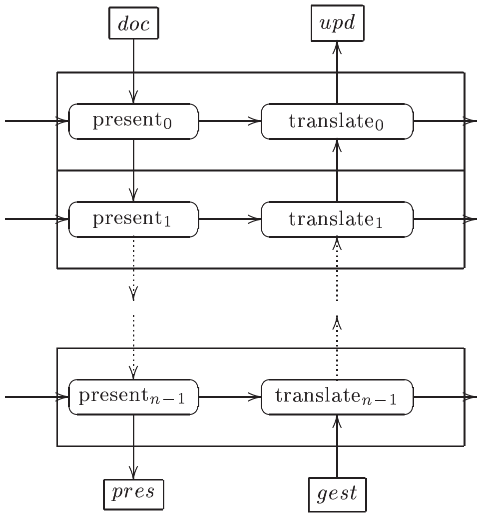

Figure 2: The layered edit cycle.

If we look at the figure, we see that there are values that go from left to right, and values that go either up or down. The values that go in horizontal direction are computed by the functions in the corresponding layer, although not necessarily in the same edit cycle. The vertical direction parameter for a layer function, on the other hand, is the result of that function in a neighboring layer (i.e. a *present* layer function gets its vertical parameter from another *present* function, and a *translate* layer function gets it from another *translate* function). When composing layers, the vertical results from one layer are passed as parameters to the next layer. The outermost vertical parameter is the parameter to the combined layer, and the outermost vertical result is the result of the combined layer.

A single layer: Figure 3 shows the data flow for a single layer. It is somewhat more complex than Figure 2, because not all horizontal parameters to $translate$ are results of $present$ (ie. $state$ is not a result of $present$). The data flow is a simplification of the data flow of the layered editor with local state from [6]. $present$ maps a high level data structure *doc* to a low level data structure $pres$. The $state$ parameter contains state that is local to the layer, as described in [6]. The $translate$ function works in the opposite direction of $present$ and maps an edit operation ($gest$) on the $pres$ data structure to an edit operation (upd) on the *doc* data structure. Besides $pres$, $present$ also returns a value $mapping$. It is used to encode information that is required by $translate$ in order to map $gest$ to upd. An example of such information is a table of pointing information, that relates objects in $pres$ to their origin in *doc*. Because $gest$ may be an edit operation on the local state, $translate$ also has $state$ as a parameter. In that case, $state'$ is the result of performing the update on the local state. In any other case, $state'$ is equal to $state$.

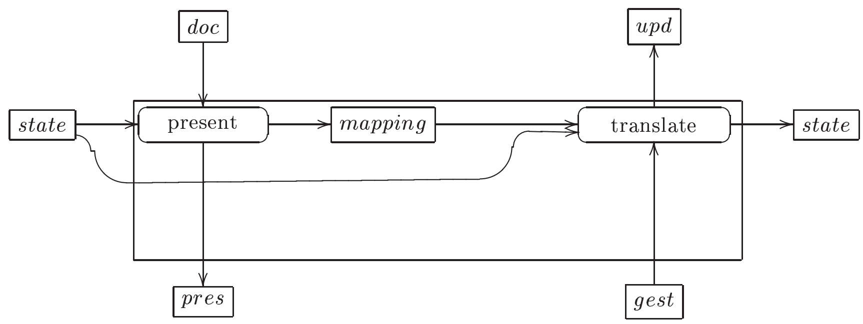

Figure 3: A single layer.

3 A layer in Haskell

In the following sections, we will explore the possibilities of implementing the layered architecture described in the previous section, using the functional language Haskell. The use of a typed programming language for describing an architecture ensures that the types of all components are correct. Furthermore, because the architecture description language is equal to the implementation language, the architecture description is an executable framework for the system being described. We have the certainty that if the types in the architecture description are correct, then the types in the implementation are correct as well. If, on the other hand, a separate architecture description language is used, then the architecture has to be implemented in a programming language, after it has been type checked or proven correct. This is an extra translation step that could possibly lead to errors.

There are two aspects to modeling a layered architecture in Haskell: the building blocks, which are the layer functions, and the connections between the building blocks. The layer functions can simply be modeled with Haskell functions that are put in a record to model a layer. Connecting the layer functions is more complicated. We will present three alternatives in the next sections, but first we give the precise types of the layer functions.

In order to clearly distinguish between horizontal and vertical parameters, we write the type of a layer function in the following form: `horArgs -> vertArg -> (vertRes, horRess)`. Thus, we see that a layer function takes one ore more horizontal arguments, a single vertical argument, and returns a single vertical result, tupled with one ore more horizontal results. We introduce a type synonym for layer functions of this type.

```
type LayerFunction horArgs vertArg horRess vertRes =
       horArgs -> vertArg -> (vertRes, horRess)
```

A layer is a record that contains the two layer functions: *present* and *translate*. The types of the layer functions follow directly from Figure 3. Following is a first attempt at a definition of `Simple'`. The names of the type and selector functions contain an apostrophe (') because we also introduce a slightly different type `Simple`, which will be used in most of the rest of this paper.

```
data Simple' state mapping doc pres gest upd =
       Simple' { present'   :: LayerFunction state doc mapping pres
               , translate' :: LayerFunction (mapping, state) gest state upd
               }
```

Strictly speaking, `Simple` is not a type but rather a type constructor, but since it is inconvenient to have to supply the type parameters every time the type is mentioned, we will denote the type with its constructor in the remainder of this text. More formally, when refering to a type `t`, where `t` is a type constructor with n parameters, we mean the type $\mathtt{t}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n$ for fresh type variables $\mathtt{a}_i$.

The field names in the record (`present'`, `translate'`) are also the names of the field selection functions. Thus, if `layer0` is an `Simple'` value, then `present' layer0` denotes the present function of the layer. The `Simple'` type is parameterized with all the types that appear in the signatures of the layer functions.

To simplify the horizontal connection between layer functions, we prefer a data type in which the horizontal result type (3rd type parameter of `LayerFunction`) of a layer function matches the horizontal argument type (1st type parameter of `LayerFunction`) of the next layer function. This implies that the horizontal result of *present* has the same type as the horizontal argument of *translate*, and that the horizontal result of *translate* has the same type as the horizontal argument of *present*. Figure 4 shows the data flow of the layer functions in the new type `Simple`:

```
data Simple state mapping doc pres gest upd =
       Simple { present ::   LayerFunction state doc (mapping, state) pres
              , translate :: LayerFunction (mapping, state) gest state upd
              }
```

Only in the next section, we use the type `Simple'`. Afterwards, `Simple` is used. Because conversions between the two types are trivial, we will not give them. The reason why we give `Simple'` at all is that it is the more appropriate type from the component programmer's view. It contains exactly the function types we expect the components to have, whereas in `Simple`, present returns the `state` parameter, which is done only to make connecting easier.

Now that the building blocks have been established, we need a way to connect them. We will show three connection methods. First, we explicitly connect all parameters and results by hand, which is a working, but not very transparent, solution. Subsequently, we introduce two more elegant and abstract methods. These methods are specific to the `Simple'` and `Simple` data types, but from the last method we derive a small library that can be used for layers with an arbitrary number of layer functions.

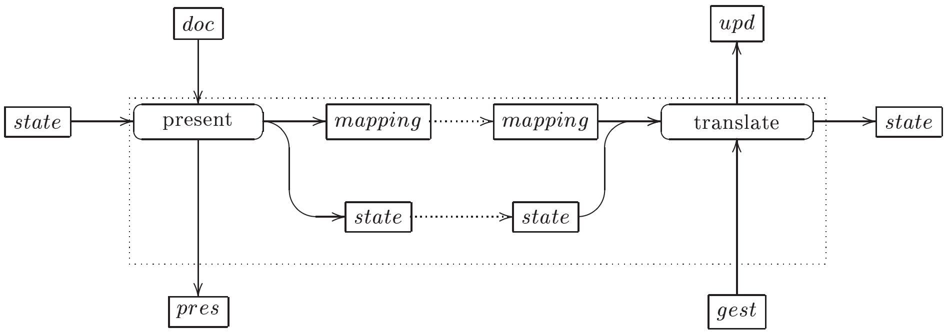

Figure 4: Data flow in layer.

4 Method 1: Explicitly connecting the components

The components of the editor are modeled with layers, but to get a working implementation, we also need to realize the data flow by connecting the components. The document must be fed into the layers, starting at the top layer and yielding the presentation at the bottom, and similarly, the edit gesture must be fed into the bottom layer, yielding the document update at the top. The layers are records that contain functions, so the most straightforward way of tying everything together is to explicitly write the selection and application of each of the functions in each of the layers. We show this by giving an example edit loop for an editor that consists of three layers: `layer0`, `layer1`, and `layer2` of type `Simple'`. The data flow between the layer functions is shown in Figure 5. The corresponding Haskell code for the edit loop is:

```
editLoop (layer0, layer1, layer2) states doc = loop states doc
 where loop (state0, state1, state2) doc =
        do { -- phase 1, Compute presentation:
             let (mapping0, pres1) = present layer0 state0 doc
           ; let (mapping1, pres2) = present layer1 state1 pres1
           ; let (mapping2, pres3) = present layer2 state2 pres2

           ; showPresentation pres3  -- phase 2
           ; gest3 <- getGesture  -- phase 3

             -- phase 4, Compute document update:
           ; let (state2', gest2) = translate layer2 (mapping2, state2) gest3
           ; let (state1', gest1) = translate layer1 (mapping1, state1) gest2
           ; let (state0', update) = translate layer0 (mapping0, state0) gest1

           ; let doc' = updateDocument update doc -- phase 5
           ; loop (state0', state1', state2') doc'
           }
```

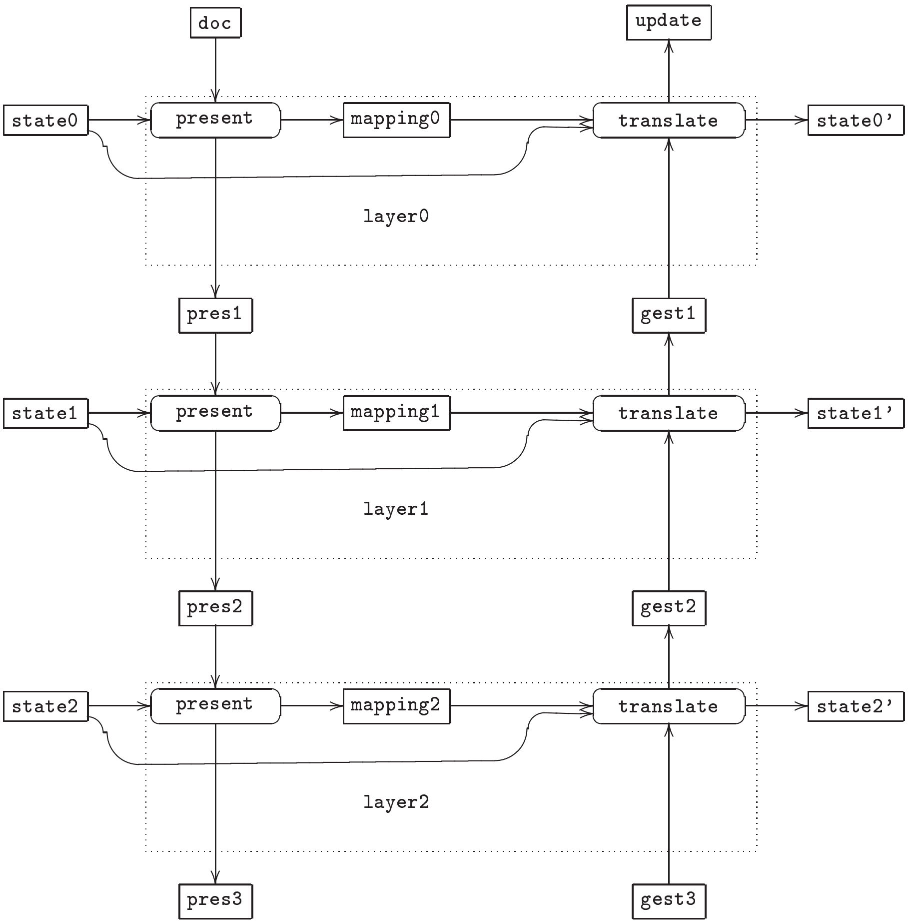

Figure 5: Data flow in and between layers.

The five phases of the simple edit cycle as presented in the previous section are directly visible in the code. Note that `present` and `translate` are selector functions, and that `present layer0` denotes the present function of layer-0. The functions `showPresentation`, `getGesture`, and `updateDocument` are left unspecified, because we want to focus on the connection of the layer functions.

The following function `main` calls `editLoop` with the correct parameters.

```
main layer0 layer1 layer2 =
 do { states <- initStates
    ; doc <- initDoc
    ; editLoop (layer0, layer1, layer2) states doc
    }
```

The functions `initStates` and `initDoc` provide the initial values for `states` and `doc`, and are left unspecified. The layers of the editor are arguments of the `main` function. An editor can now be instantiated by applying the function `main` to three `Simple'` values that implement the different layers. The type system verifies that the implemented layer functions have the correct type signatures.

A disadvantage of the implementation of the edit loop sketched in this section is that the patterns of the data flow are not very transparent. The fact that the mapping parameters are horizontal parameters and that the presentation is a vertical parameter is not immediately clear from the program code. Moreover, the patterns for upward and downward vertical parameters are standard, but have to be given explicitly for each application, increasing the chance of errors and decreasing transparency. Finally, the number of layers is hard coded in the implementation. If the system is extended with an extra layer, variables have to be renamed. If each type appearing in the layers is distinct, the type checker catches mistakes but if some parameters or results have the same type, the type checker will not detect a problem when, for example, two such equally typed variables are swapped. We will therefore try to find a more abstract way of connecting the layers.

5 Method 2: Nested cartesian products

In this section we create an abstraction for the horizontal and vertical data flow patterns in the edit loop of the previous section. We present a number of combinators that allow us to compose layers, so that instead of having to give the applications of the layer functions for each of the layers, we can call the layer function of the composed layer. For the main loop in the previous section, this implies that both the present and the translate phase can be written with one function application instead of three. The combinators also make the data flow more explicit. The direction of the vertical parameter is apparent by the choice of combinator, and not by the way the argument is threaded through the function applications, as is the case in the previous section.

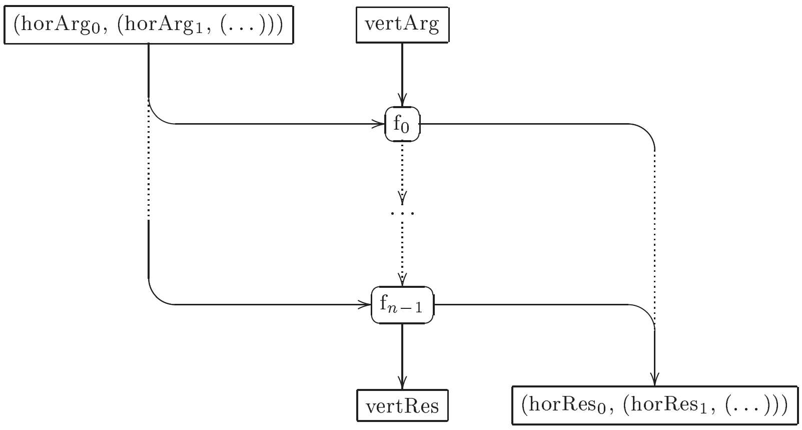

Figure 6: Horizontal parameters in nested cartesian products.

Similar to the way in which the function composition operator $\cdot$ can be used to compose functions f and g into $f \cdot g$, we develop a `combine` combinator that takes two layers and returns a combined layer. The layer functions of the combined layer are the compositions of the layer functions in the layers that are combined.

In the method described in this section, each of the functions in the combined layer not only takes a vertical argument and returns a vertical result, but it also takes a collection of horizontal arguments (one for each layer) and returns a collection of horizontal results (one from each layer). The composition combinator takes care of distributing the horizontal arguments to the corresponding layers, and also collects the horizontal results. The combined layer provides layer functions of type `LayerFunction horArgsC vertArg horRessC vertRes`. The parameters `horArgsC` and `horResC` stand for the types of the collections of horizontal parameters and results. Figure 6 sketches the data flow in the combined layer. Only one layer function with a downward vertical parameter is shown.

Because the types of the horizontal parameters are typically not the same, the collections cannot be represented by lists. The same restriction holds for the horizontal results. Moreover, we want to be able to determine at compile time whether the collection contains the right number of elements. A tuple or cartesian product is more suitable for the task, but has the disadvantage that its components cannot be accessed in a compositional way. Therefore, we will use a nested cartesian product consisting of only 2-tuples to represent the horizontal parameters and results.

An example of such a nested cartesian product containing 4 elements is:$(e_0,(e_1,(e_2,e_3)))$, but also $(((e_0,e_1),e_2), e_3)$ and $((e_0,e_1),(e_2,e_3))$ are valid examples. Two nested cartesian products are of equal type if and only if they have the same structure, and elements at equal positions have equal types. We can access the elements in the product in a compositional way by pattern matching on the top level product. For example: `f`

```
(firstElt, rest) = ...
```

Because nested cartesian products with different structure are of different type, we use the combinators in this section only in a right associative way, so all products are of the form `(e`$_0$`,(e`$_1$`,(e`$_2$`, (..., e`$_n$`)...)))`.

We first define two combinators for composing layer functions. We need two combinators because there are two ways in which layer functions can be composed, depending on the direction of the vertical parameter. An upward vertical parameter passes through the bottom layer first, and then through the top layer, whereas a downward vertical parameter passes through the top layer first. The two combinators are `composeDown` and `composeUp`. Figure 7 shows the data flow for the two combinators.

The combinator `composeDown` composes two layers `u` and `l` by feeding the intermediate vertical result of `u` into `l`, establishing a downward data flow. At the same time, the horizontal parameters for `u` and `l` are taken from the horizontal parameter to the combined layer (which is a tuple), and the horizontal result for the combined layer is formed by tupling the horizontal results of `u` and `l`.

```
composeDown :: LayerFunction horArgsU arg horRessU interm ->
               LayerFunction horArgsL interm horRessL res ->
               LayerFunction (horArgsU, horArgsL) arg (horRessU,horRessL) res
composeDown upper lower =
  \(horArgsU, horArgsL) arg ->
    let (horRessU, interm) = upper horArgsU arg
        (horRessL, res)    = lower horArgsL interm
    in  ((horRessU,horRessL), res)
```

In a similar way, we define a combinator `composeUp` for combining layer functions (e.g. `translate`) in which the vertical parameter goes upward. We only show its type here.

```
composeUp :: LayerFunction horArgsU interm horRessU res ->
             LayerFunction horArgsL arg horRessL interm ->
             LayerFunction (horArgsU, horArgsL) arg (horRessU,horRessL) res
```

Combining layers: Using `composeUp` and `composeDown`, we can define a combinator to combine `Simple` layers. From now on, we use the type `Simple` instead of `Simple'` because the matching horizontal parameter types in `Simple` make the connection easier.

First, we need to define a new data type for combined layers. Although the composition of two layer functions is a layer function itself, we cannot put the compositions of the layer functions in a `Simple` record. According to the definition of `Simple`, the horizontal parameter of the first layer function (`present`) has type `state`, and the result has type (`mapping`, `state`). In contrast, for `present` in the composed layer, the horizontal parameter

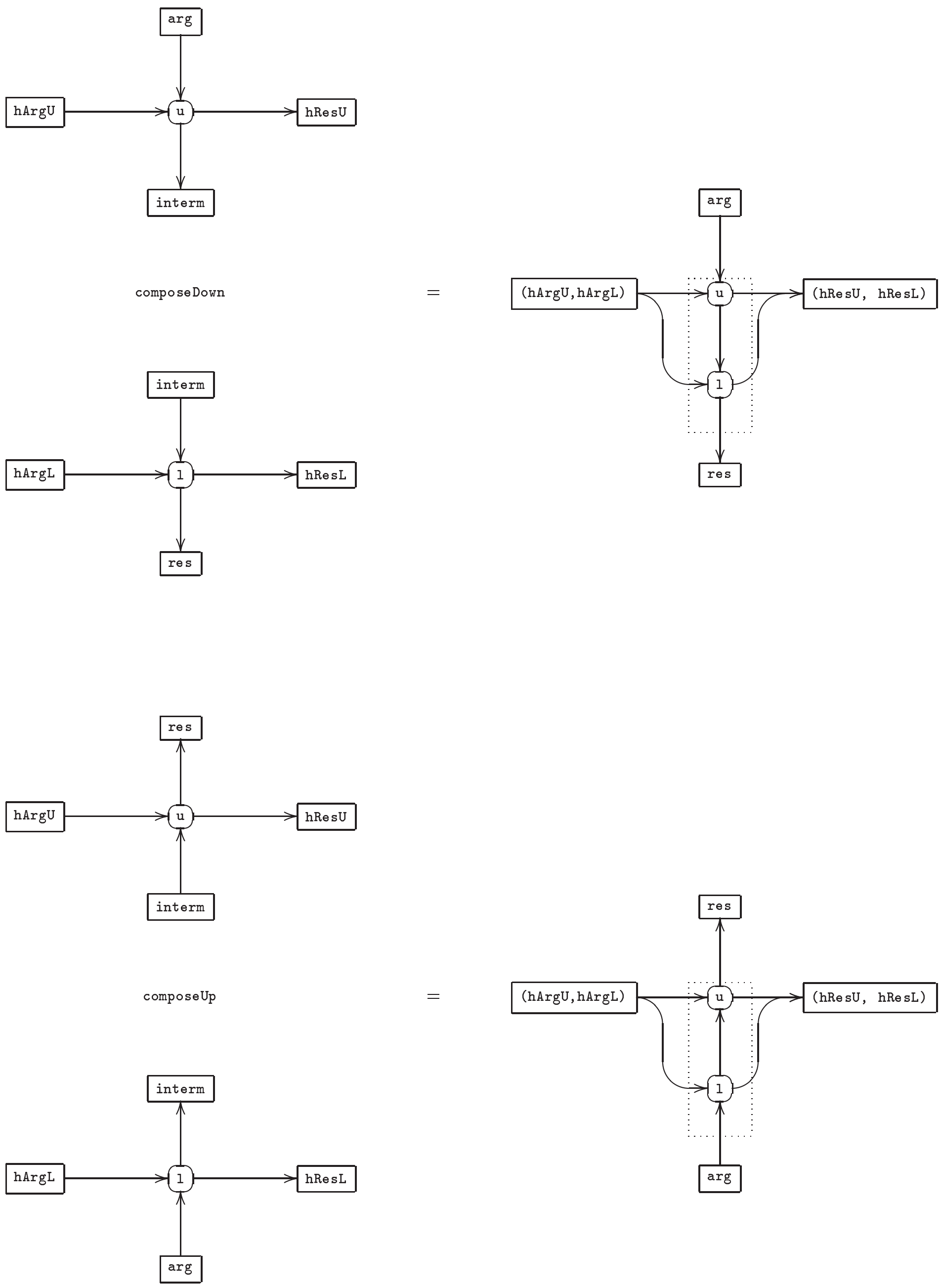

Figure 7: `composeDown` and `composeUp`

is a collection of states and the result a collection of mapping and state tuples. We cannot specify the collection types in the type signature, so we will simply use single type variables for the types.

The type `LayerC` is a more general version of `Simple`; it does not specify the exact structure of the horizontal parameters. The parameter `states` represents the nested cartesian product of `state` values, and the parameter `mappingsStates` represents the nested cartesian product of `mapping` and `state` tuples.

```
data LayerC states mappingsStates doc pres gest upd =
       LayerC { presentC ::   LayerFunction states doc mappingsStates pres
              , translateC :: LayerFunction mappingsStates gest states upd
              }
```

The trivial function `lift` takes a layer of type `Simple` and returns a `LayerC` layer.

```
lift :: Simple a b c d e f -> LayerC a (b,a) c d e f
lift simple =
  LayerC { presentC = present simple
         , translateC = translate simple
         }
```

The `combine` combinator is defined by using the appropriate compose combinator on each of the layer functions.

```
combine :: LayerC a b c d e f -> LayerC g h d i j e ->
           LayerC (a,g) (b,h) c i j f
combine upper lower =
  LayerC { presentC = composeDown (presentC upper) (presentC lower)
         , translateC = composeUp (translateC upper) (translateC lower)
         }
```

Simple editor: The main editor loop from the previous section now reads:

```
editLoop layers states doc = loop states doc
 where loop states doc =
        do { -- Compute presentation:
             let (pres, mappingsStates) = presentC layers states doc

           ; showPresentation pres
           ; gest <- getGesture

             -- Compute document update:
```

```
          ; let (update, states') = translateC layers mappingsStates gest

          ; let doc' = updateDocument update doc
          ; loop states' doc'
          }
```

The main function is almost the same as in the previous section, except that instead of a 3-tuple of layers, the combined layers are passed to editLoop.

```
main layer0 layer1 layer2 =
 do { (state0, state1, state2) <- initStates
    ; doc <- initDoc
    ; let layers = lift layer0 `combine` lift layer1 `combine` lift  layer2
    ; editLoop layers ((state0, state1), state2) doc
    }
```

Conclusions: The nested cartesian products solution is more compositional than the approach of the previous section, and a lot of data flow is hidden from the main loop by automatically passing the results of one layer into another. However, all horizontal parameters are passed all the way through the composite layer, and are visible in the main loop, which is not where they conceptually belong. Consequently, the type of the composite layer is parameterized with all types appearing in the layers, leading to huge type signatures. We will therefore show yet another approach in which the horizontal parameters stay within each layer and are not visible in the main editor loop.

6 Method 3: Direct parameter passing

In the previous section, the horizontal results that are computed during the evaluation of one layer function are returned explicitly and passed as arguments to the next layer function. In the layer functions developed in this section, the horizontal results are not returned explicitly. Instead, the next layer function is returned with the horizontal results already applied to it. As a consequence, the main editor loop becomes more transparent, and the type of the composed layers is simpler, since the type variables for the horizontal parameters are no longer explicitly visible.

We will illustrate the direct parameter passing method with an example. Recall that the main loop of the simple editor, encoded with nested cartesian products method, contains the following code:

```
...
let (mapstates, pres) = present' layers states doc
...
```

```
let (states', update) = translate' layers mapstates gest
...
```

But with the direct parameter passing method, it will contain:

```
...
let (pres, translateStep) = presentStep doc
...
let (update, presentStep) = translateStep gest
...
```

In other words, each combined layer function application returns the result together with the next combined layer function. Horizontal parameters are now completely hidden from the main loop. The code that is shown is not entirely accurate, as it turns out that some pattern matching is required, but it gives the general idea. Note that the order in which the layer functions are evaluated is now enforced by the layer model. It is not possible to evaluate `translateStep` before `presentStep`.

Type definitions: The type of the layer needs to have the following structure: `(Doc -> (Pres, Gest -> (Upd, Doc -> (Pres, Gest -> (Upd, ...)))))`. Unfortunately, we cannot use a type declaration: `type Layer = (Doc -> (Pres, Gest -> (Upd, Layer)))`, because Haskell does not allow recursive type synonyms. A `newtype` declaration must be used with the disadvantage that values of the type have to be wrapped with constructor functions.

We encode the type by dividing the computation in *steps*; one step for each layer function. Because the number of steps is always equal to the number of layer functions, the statement that a layer has n layer functions, amounts to the same as that it is an n step layer. From now on, we will use the term number of layer functions of a layer interchangebly with number of steps of a layer.

For each step, we define a separate type. The types are mutually recursive. Because the types of the vertical parameters may be different for each layer, the step types are parameterized with the types of the vertical parameters. For the `Simple` layer type, we get:

```
newtype PresStep doc pres gest upd =
          PresStep  (doc ->  (pres, TransStep gest upd doc pres))
newtype TransStep gest upd doc pres =
          TransStep (gest -> (upd,  PresStep doc pres gest upd))

type Layer doc pres gest upd = PresStep doc pres gest upd
```

In the next section, we will give a more abstract definition of the step types. For now, we will use the above definitions because they clearly show the mutually recursive structure of the types.

Instead of using a separate type for each of the layer functions, it is also possible to use one `newtype` declaration that encapsulates both layer functions:

```
newtype Layer doc pres gest upd =
  Layer (doc -> (pres, gest -> (upd, Layer doc pres gest upd)))
```

We prefer the mutually recursive types because they are more symmetric than the single `Layer` type. Because of the use of newtypes, values have to be wrapped with constructors, as well as unwrapped. Wrapping and unwrapping occurs either for every step, in case of the mutually recursive types, or only once every edit cycle, in case of the single type encapsulating all layer functions. Doing it for every step leads to more elegant combinators.

We define two combinators for constructing and combining `Layer` values. The first combinator, `lift`, takes a `Simple` layer, which is a record with layer functions, and converts it to a direct parameter passing layer of type `Layer`. To combine layers of type `Layer`, we define a combinator `combine`. Both combinators in this section are specific to the `Simple` type. In the next section, we define a library to construct `lift` and `combine` for arbitrary layers.

Lift: The combinator `lift` takes a layer of type `Simple` and returns a `Layer` value:

```
lift :: Simple state mapping doc pres gest upd -> state ->
        Layer doc pres gest upd
lift layer state = presStep state
 where presStep state = PresStep $
          \doc ->  let ((mapping,state), pres) = present layer state doc
                   in  (pres, transStep (mapping,state))
       transStep (mapping,state) = TransStep $
          \gest -> let (state', upd) = translate layer (mapping, state) gest
                   in  (upd, presStep state')
```

Besides the `layer` parameter, `lift` gets a second parameter, `state`, which is the initial value of the horizontal `state` parameter in the layer. The data flow pattern of the horizontal parameters is encoded entirely in the definition of `lift`. Moreover, the `state` type is not visible in the result of `lift`, so once the horizontal state is passed to the lifted layer, it is no longer visible outside this layer; the `lift` combinator takes care of passing around the horizontal parameters to and from the layer functions, and also to the next edit cycle. This state hiding helps to keep the architecture description transparent, because now the horizontal parameters do not play a rô le anymore when the layers are connected.

Combine: With `lift` we can instantiate layer values of type `Layer doc pres gest upd`. To combine layers, we define a combinator `combine` with the following type:

```
combine :: Layer high med emed ehigh -> Layer med low elow emed ->
           Layer high low elow ehigh
```

The order of the parameters in the type of `combine` might seem a bit odd. The reason for this order is that the first type in each pair of parameters specifies an argument type, and the second type specifies a result type. The composed `pres` function is created by applying the `pres` function of the first layer (`high -> med`) to the `high` argument and then applying the `pres` function of the second layer (`med -> low`) to the result. The `trans` function, however, has a different direction: the `trans` function of the second layer (`elow -> emed`) is applied to the argument, and the `trans` function of the first layer (`emed -> ehigh`) is applied to the result.

The implementation of combine is just plumbing to get the parameters at the right places. The direction of the vertical parameters is encoded in the definition of `combine`. For `pres`, this direction is downward (the function from the upper layer is applied first), whereas for `trans`, it is upward (the function from the lower layer is applied first).

```
combine :: Layer high med emed ehigh -> Layer med low elow emed ->
           Layer high low elow ehigh
combine = presStep
 where presStep (PresStep upr) (PresStep lwr) = PresStep $
           \high -> let (med, uprTrans) = upr high
                        (low, lwrTrans) = lwr med
                    in  (low, transStep uprTrans lwrTrans)
       transStep (TransStep upr) (TransStep lwr) = TransStep $
           \elow -> let (emed, lwrPres') = lwr elow
                        (ehigh, uprPres') = upr emed
                    in  (ehigh, presStep uprPres' lwrPres')
```

Simple editor: The edit loop of the simple editor no longer has a reference to the horizontal parameters. Furthermore, the combined layer is called `presentStep` instead of `layers`, to reflect that it is the present step of the computation.

```
editLoop presentStep doc =
 do { let (pres, TransStep translateStep) = presentStep doc

    ; showPresentation pres
    ; gesture <- getGesture

    ; let (update, PresStep presentStep') = translateStep gesture

    ; let doc' = updateDocument update doc

    ; editLoop presentStep' doc'
    }
```

In the `main` function, the combined layer is created by lifting the layers together with their initial states and using `combine` to put them together.

```
main layer0 layer1 layer2 =
 do { (state0, state1, state2) <- initStates
    ; doc <- initDoc
    ; let PresStep presentStep =           lift layer0 state0
                                 `combine` lift layer1 state1
                                 `combine` lift layer2 state2
    ; editLoop presentStep doc
    }
```

Conclusions: The direct parameter passing model hides the data flow of the horizontal parameters from the main loop of the system. Furthermore, the types of the horizontal parameters, as well as the intermediate vertical parameters are hidden from the type of the composed layer, similar to the way in which the type of the intermediate result for a composition of two functions $f :: b \to c$ and $g :: a \to b$ is hidden from the type of the composition $f \cdot g :: a \to c$. As a result, both horizontal and vertical data flow are made more transparent.

7 Developing a library for architecture descriptions

In this section, we develop a small *meta combinator* library for implementing the direct parameter passing `lift` and `combine` combinators for layered architectures with an arbitrary number of layer functions. The combinators in the library are meta combinators because they are used to build `lift` and `combine`, which are combinators themselves.

The combinators from the previous section are just one case of a layered architecture: a layer with two layer functions and hence two steps. Even though `lift` and `combine` are straightforward to write, some code is duplicated, and small errors are easily made. Therefore, instead of a general description for writing `lift` and `combine` by hand, it is better to have a small library of meta combinators for building them. Another advantage of a meta combinator library is that the direction of the vertical data flow can be expressed by using a meta combinator whose name reflects the direction, instead of explicitly encoding the direction for each of the steps in the `combine` function itself.

In the next sections, we first analyze the data type definitions that are required for a layer with an arbitrary number of layer functions. Then, we derive meta combinators for `lift` and `combine`.

Looking at the definitions of `lift` and `combine` in the previous section, we see that they both consist of two parts: one for each step in the layer. Both functions define a local function for each of the layer functions in the layer. In both `lift` and `combine` for the `Simple` layer type, these local functions are called `presStep` and `transStep`.

The method we use for the derivation of the meta combinators is to start with such a local function and gradually factorize out all step specific aspects. After this process, we are left with a function that, when applied to the step specific aspects, has the same behaviour as the local function we started with. Moreover, because the function is not step specific anymore, it can be used to get the behaviour of the other step functions as well. This function is the meta combinator that we can use to build instances of the combinator for architectures with arbitrary numbers of steps.

For `lift`, the local function is similar for each of the steps, but for `combine`, two variations exist, depending on the direction of the vertical parameter. Consequently, one meta combinator is derived for `lift`, whereas for `combine` we get two meta combinators: one for steps with upward parameters and one for steps with downward parameters.

7.1 Type definitions

In order to extend the direct parameter passing method to layers with an arbitrary number of steps, we first investigate what the data types for each step will be. It turns out that it is not possible to give a general type definition in Haskell that can be used for the step definitions for layers with arbitrary numbers of steps. The reason for this is that the number of type variables depends on the number of steps. However, we can give a simple method for defining the data types, given the number of steps in the layer.

If we take the two step data types for `Simple'` from Section 6, and rename the variables, we get:

$\texttt{newtype Step}_{1,2}\texttt{ a b c d = Step}_{1,2}\texttt{ (a -> (b, Step}_{2,2}\texttt{ c d a b))}$
$\texttt{newtype Step}_{2,2}\texttt{ a b c d = Step}_{2,2}\texttt{ (a -> (b, Step}_{1,2}\texttt{ c d a b))}$

The pattern on the righthand side can be captured by a type synonym $\texttt{Step}_2$, which makes the individual step types somewhat simpler:

$\texttt{type Step}_2\texttt{ next a b c d = (a ->(b, next c d a b))}$

Furthermore, we write the data type definitions as records, in order to get a selector function for each type. The selector functions will be used in the definition of `combine`.

$\texttt{newtype Step}_{1,2}\texttt{ a b c d = Step}_{1,2}\ \{\texttt{step}_{1,2}\texttt{ :: Step}_2\texttt{ Step}_{2,2}\texttt{ a b c d}\}$
$\texttt{newtype Step}_{2,2}\texttt{ a b c d = Step}_{2,2}\ \{\texttt{step}_{2,2}\texttt{ :: Step}_2\texttt{ Step}_{1,2}\texttt{ a b c d}\}$

For a layer with three layer functions, and hence three steps, we need three mutually recursive step data types. Each step has six parameters.

$\texttt{newtype Step}_{1,3}\texttt{ a b c d e f = Step}_{1,3}\ \{\texttt{step}_{1,3}\texttt{ :: a -> (b, Step}_{2,3}\texttt{ c d e f a b)}\}$

```
newtype Step_{2,3} a b c d e f = Step_{2,3} {step_{2,3} :: a -> (b, Step_{3,3} c d e f a b)}
newtype Step_{3,3} a b c d e f = Step_{3,3} {step_{3,3} :: a -> (b, Step_{1,3} c d e f a b)}
```

Which can be written simpler as:

```
type Step_3 next a b c d e f = (a -> (b, next c d e f a b))

newtype Step_{1,3} a b c d e f = Step_{1,3} {step_{1,3} :: Step_3 Step_{2,3} a b c d e f}
newtype Step_{2,3} a b c d e f = Step_{2,3} {step_{2,3} :: Step_3 Step_{3,3} a b c d e f}
newtype Step_{2,3} a b c d e f = Step_{3,3} {step_{3,3} :: Step_3 Step_{1,3} a b c d e f}
```

Clearly, the type definitions for three step layers are very similar to the type definitions for two step layers. However, because the number of parameters for the type depends on the number of steps, it is not possible to give one Haskell type synonym `Step`. We will have to settle for a general description of how to construct the type synonym. For a layer with n layer functions, the type of $\texttt{Step}_n$ has $2n$ parameters. The parameters are named $\texttt{a}_1\ \texttt{r}_1\ \ldots\ \texttt{a}_n\ \texttt{r}_n$ to reflect that for each pair of parameters, the first one ($\texttt{a}_i$) is a vertical argument, and the second one ($\texttt{r}_i$) is a vertical result

```
type Step next a_1 r_1 ... a_n r_n = (a_1 -> (r_1, next a_2 r_2 ... a_n r_n a_1 r_1))
```

The individual step data types are defined as:

```
newtype Step_1 a_1 r_1 ... a_n r_n = Step_1 {step_1 :: Step Step_2 a_1 r_1 ... a_n r_n )}
newtype Step_2 a_1 r_1 ... a_n r_n = Step_2 {step_2 :: Step Step_3 a_1 r_1 ... a_n r_n )}
...
newtype Step_n a_1 r_1 ... a_n r_n = Step_n {step_n :: Step Step_1 a_1 r_1 ... a_n r_n )}
```

Because no general `Step` type synonym is possible, an implementor has to construct the appropriate `Step` type definition, as well as the step data types. In a language with dependent types, such as Cayenne [1], a general type `Step` could have been defined.

7.2 Derivation for `lift`

First we develop a meta combinator for the `lift` function. Below is the code for `lift` for layers with two steps. This is the code from section 6 in which a few variables have been renamed to emphasize the data flow patterns instead of the meaning of the individual steps. The local functions `presStep` and `transStep` are renamed `step1` and `step2`. Furthermore, in `presStep`, `state` is renamed to `horArgs`, and `(mapping, state)` is renamed to `horRess`. And similarly, in `transStep`, `(mapping, state)` is renamed to `horArgs` and `state'` is renamed to `horRess`. Finally, also the vertical parameters and results in both local functions are renamed to `vertArg` and `vertRes`. Because the derivation will not affect the type, we do not rename any type variables. The resulting function does not show its purpose as clear as the original does, but it makes clear that the data flow in both local functions is similar.

```
lift :: Simple state mapping doc pres gest upd ->
        state -> Layer doc pres gest upd
lift simple state = step1 state
 where step1 horArgs = PresStep $
           \vertArg -> let (vertRes, horRess) = present simple horArgs vertArg
                       in  (vertRes, step2 horRess)
       step2 horArgs = TransStep $
           \vertArg -> let (vertRes, horRess) = translate simple horArgs vertArg
                       in  (vertRes, step1 horRess)
```

The definitions of the local functions `step1` and `step2` contain mutually recursive references. We can take the mutual recursion out of the definitions by supplying the next step as a parameter to each function.

```
lift simple state = (step1 (step2 (lift simple))) state
 where step1 next horArgs = PresStep $
           \vertArg -> let (vertRes, horRess) = present simple horArgs vertArg
                       in  (vertRes, next horRess)
       step2 next horArgs = TransStep $
           \vertArg -> let (vertRes, horRess) = translate simple horArgs vertArg
                       in  (vertRes, next horRess)
```

We can turn this into a more elegant definition by taking the following steps. If we drop the `state` parameter on the lefthand and righthand sides of the first line, and rewrite the function application as a composition, we get:

```
lift simple = (step1.step2) (lift simple)
```

Because the result of `lift layer` is the function `(step1.step2)` applied to this result, we can use the combinator `fix`:

```
fix a = let fixa = a fixa
        in  fixa
```

to capture the recursion pattern, yielding:

```
lift simple = fix $ step1 . step2
 where step1 next horArgs = PresStep $
           \vertArg -> let (vertRes, horRess) = present simple horArgs vertArg
                       in  (vertRes, next horRess)
       step2 next horArgs = TransStep $
           \vertArg -> let (vertRes, horRess) = translate simple horArgs vertArg
                       in  (vertRes, next horRess)
```

Except for the applications of the two constructors `PresStep` and `TransStep` and the layer functions `present'` and `translate'`, the local functions are now equal. We eliminate the constructors and the layer functions, by passing them as parameters `pack` and `layerF` to the local functions.

Because `step1` and `step2` are now the same function, we give this function a new name: `liftStep`, which is the meta combinator for building `lift` functions. Below is the new definition of `lift` together with the definition of `liftStep`, which is not a local function anymore. The `liftStep` below is not the final version of the meta combinator. In Section 7.4 we define a type class that contains the pack function, so the final `liftStep` does not get `pack` as a parameter and has a slightly different type with a type constraint in it.

```
lift :: Simple state mapping doc pres gest upd ->
        state -> Layer doc pres gest upd
lift simple =
  fix $ liftStep PresStep (present simple)
      . liftStep TransStep (translate simple)

liftStep :: ((vArg -> (vRes, nStep)) -> step) ->
            (hArgs -> vArg -> (vRes,hRess)) ->
            (hRess -> nStep) ->
            hArgs -> step
liftStep pack layerF next horArgs = pack $
    \vertArg -> let (vertRes, horRess) = layerF horArgs vertArg
                in  (vertRes, next horRess)
```

`liftStep` **works for layers with arbitrary numbers of steps:** The definition of `liftStep` does not depend on the number of steps of a layer, even though the derivation starts with the definition of `lift` for layers with two layer functions. In the case of n steps, there will be n local step functions, each one containing a reference to the next, except for the last one, which contains a reference to the first function.

We can now perform the same steps to the n-step `lift` as we did for the 2-step `lift`. We only sketch the process here. After renaming the variables, the explicit recursion is removed from the local function definitions, by passing the next step as a parameter, and `lift` will look like:

```
lift layer state = (step₁ (step₂ ... (stepₙ (lift layer))...))  state
 where ...
```

which can be written as a composition:

```
lift layer = (step₁ ... stepₙ) lift layer
 where ...
```

And, after using `fix` to capture the recursion, becomes:

```
lift layer = fix $ step₁ ... stepₙ
 where ...
```

Finally, the constructor and layer functions have to be passed as parameters to the local step functions and the calls to the local functions become calls to `liftStep`. If we assume that the n constructors are $\texttt{Step}_i$ and the n layer functions are $\texttt{layerFunction}_i$, then the final definition of `lift` is:

```
lift layer=
  fix $ liftStep Step_1 (layerFunction_1 layer)
      .  ...
      . liftStep Step_n (layerFunction_n layer)
```

The `liftStep` metacombinator can therefore be used to create the `lift` combinator for a layer with an arbitrary number of layer functions.

7.3 Derivation for `combine`

The derivation of the meta combinators for `combine` is largely similar to the derivation for `lift`. We start with the original definition of `combine` for layers with two steps (see Section 6), in which we have renamed some variables. The two local functions are renamed to `step1` and `step2`. Both `uprTrans` and `uprPres'` are renamed to `nextUpr`, and both `lwrTrans` and `lwrPres'` to `nextLwr`. The names of the vertical parameters and results in `presStep` (`low`, `med` and `high`) are already appropriate, so `elow`, `emed` and `ehigh` in `transStep` are also renamed to `low`, `med` and `high`.

```
combine :: Layer high med emed ehigh -> Layer med low elow emed ->
           Layer high low elow ehigh
combine = step1
 where step1 (PresStep upr) (PresStep lwr) = PresStep $
           \high -> let (med, nextUpr) = upr high
                        (low, nextLwr) = lwr med
                    in  (low, step2 nextUpr nextLwr)
       step2 (TransStep upr) (TransStep lwr) = TransStep $
           \low ->  let (med, nextLwr) = lwr low
                        (high, nextUpr) = upr med
                    in  (high, step1 nextUpr nextLwr)
```

The explicit mutual recursion in the local functions is removed by passing the next step as a parameter.

```
combine = step1 (step2 combine)
 where step1 nextStep (PresStep upr) (PresStep lwr) = PresStep $
           \high -> let (med, nextUpr) = upr high
                        (low, nextLwr) = lwr med
                    in  (low, nextStep nextUpr nextLwr)
       step2 nextStep (TransStep upr) (TransStep lwr) = TransStep $
           \low ->  let (med, nextLwr) = lwr low
                        (high, nextUpr) = upr med
                    in  (high, nextStep nextUpr nextLwr)
```

Analogously to `lift`, the `fix` combinator can be used to abstract from the explicit recursion.

```
combine = fix $ step1 . step2
 where step1 nextStep (PresStep upr) (PresStep lwr) = PresStep $
           \high -> let (med, nextUpr) = upr high
                        (low, nextLwr) = lwr med
                    in  (low, nextStep nextUpr nextLwr)
       step2 nextStep (TransStep upr) (TransStep lwr) = TransStep $
           \low ->  let (med, nextLwr) = lwr low
                        (high, nextUpr) = upr med
                    in  (high, nextStep nextUpr nextLwr)
```

Because the constructors in the two local functions not only appear in the righthand side, but also in the patterns of the lefthand side, it is not as straightforward to factorize the constructors out of the definition. For the righthand side occurrences, we can pass the constructor as an argument `pack` to the local function, similar to what was done for `lift`. The constructors also appear on the lefthand side of the local functions. Because we cannot use a parameter for the pattern matching, we pass a selector function `unpack`. Instead of pattern matching, the `unpack` function is used to unpack the arguments of the local functions.

Unfortunately, the selector function is used on two arguments with different types, whereas Haskell only allows monomorphic applications of function arguments. Using a universally quantified type does not work, because we have to quantify over all parameters in the types, and the number of parameters is not determined. For now, the selector function is passed twice, as `unpackL` and `unpackH`. The two unpack functions are used to unpack the `upr` and `lwr` parameters. Together with the constructor (the pack function), the unpack functions are put in a 3-tuple. In the next section, we show how a more elegant approach that uses a type class for the pack and unpack functions.

We assume that the selector functions `presStep` and `transStep` are defined by declaring the `PresStep` and `TransStep` types as records, according to the method shown in Section 7.1.

```
combine = fix $ step1 (PresStep,presStep,presStep)
              . step2 (TransStep,transStep,transStep)
 where step1 (pack, unpackU, unpackL) nextStep upr lwr = pack $
           \high -> let (med, nextUpr) = (unpackU upr) high
                        (low, nextLwr) = (unpackL lwr) med
                    in  (low, nextStep nextUpr nextLwr)
       step2 (pack, unpackU, unpackL) nextStep upr lwr = pack $
           \low ->  let (med, nextLwr) = (unpackL lwr) low
                        (high, nextUpr) = (unpackU upr) med
                    in  (high, nextStep nextUpr nextLwr)
```

Unlike in the case for `lift`, the two step functions are not the same now, because the directions of the vertical parameters for `step1` and `step2` are different. Therefore, instead of one meta combinator, we get two: `combineStepUp` and `combineStepDown`. Because all step specific information has been factorized out, the functions can be used for arbitrary steps.

```
combine :: Layer high med emed ehigh -> Layer med low elow emed ->
           Layer high low elow ehigh
combine = fix $ combineStepDown (PresStep,presStep,presStep)
               . combineStepUp (TransStep,transStep,transStep)

combineStepDown :: ( (h -> (l,nStepC)) -> stepC
                   , stepU -> h -> (m, nStepU)
                   , stepL -> m -> (l, nStepL) ) ->
                   (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepDown (pack, unpackU, unpackL) nextStep upr lwr = pack $
    \high -> let (med, nextUpr) = (unpackU upr) high
                 (low, nextLwr) = (unpackL lwr) med
             in  (low, nextStep nextUpr nextLwr)

combineStepUp :: ( (l -> (h,nStepC)) -> stepC
                 , stepU -> m -> (h, nStepU)
                 , stepL -> l -> (m, nStepL)) ->
                 (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepUp (pack, unpackU, unpackL) nextStep upr lwr = pack $
    \low -> let (med, nextLwr) = (unpackL lwr) low
                (high, nextUpr) = (unpackU upr) med
            in  (high, nextStep nextUpr nextLwr)
```

Like `liftStep`, the meta combinators do not depend on the number of steps, so they can be used to build `combine` for layers with an arbitrary number of layer functions.

7.4 A type class for pack and unpack

The previous subsections define meta combinators for the functions `lift` and `combine`. Because explicit constructor appearances have been removed from the meta combinators, both functions have a function argument that is used for packing the result, and `combine` has two function arguments to unpack its two layer arguments. Although the unpack function for both layer arguments is the same, it has to be passed twice because the function is used on arguments of different type. In this section, we construct a type class `Pack` that contains the pack and unpack functions for the step data types. The type class eliminates the need for explicitly passing the pack and unpack functions to the meta combinators. Consequently, the unpack function does not need to be passed twice anymore.

First, we take a look at the types of the pack and unpack parameters of the meta combinators. Because the types of the pack and unpack functions have the same structure in each of the three combinators, it does not matter which meta combinator we look at, exception that `liftStep` does not have an unpack parameter. Therefore, we take a look at `combineStepDown`:

```
combineStepDown :: ( (h -> (l,nStepC)) -> stepC
                   , stepU -> h -> (m, nStepU)
                   , stepL -> m -> (l, nStepL)) ->
                   (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepDown (pack, unpackU, unpackL) nextStep upr lwr = pack $ ...
```

The type of `pack` is `(h -> (l, nStepC)) -> stepC`. The result of the function (`stepC`) is a step type (`PresStep` or `TransStep` in the `Simple'` case) and its argument is a function from `h` to a tuple of `l` and the next step value (`nStepC`). We rename the parameters in the type of `pack` to the more general: `(arg -> (res, nStep)) -> step`.

For the unpack function, we only look at `unpackU`, since the structure of both unpack functions is the same. The type of `unpackU` is `stepU -> h -> (m,nStepU)`. The function takes a step value (`stepU`) and returns a function from type `h` to a tuple of `m` and the next step type (`nStepU`). Analogously to `pack`, we rename the type variables: `unpack :: step -> arg -> (res,nStep)`.

Thus, our first shot at a type class with `pack` and `unpack` is:

```
class Pack step arg res nStep where
  pack :: (arg -> (res, nStep)) -> step
  unpack :: step -> arg -> (res, nStep)
```

However, this definition is not complete yet. In order to see why, we have to look at the new type of `combineStepDown`:

```
combineStepDown :: ( Pack stepC h l nStepC
```

```
                     , Pack stepU h m nStepU
                     , Pack stepL m l nStepL ) =>
                     (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
```

The type is ambiguous, because type variables `h`, `m` and `l` appear in the constraint, but not in the actual type of the function. The type inference algorithm cannot determine the values (which are types) of `h`, `m` and `l` in an application of `combineStepDown`. The problem only arises in the combine step meta combinators, because in the type of `liftStep`, all variables in the constraint appear in the type as well.

We resolve the ambiguity by adding a functional dependency [3] `step -> arg res` between `step` and `arg` and `res` to the class declaration.

The meta combinators are used to build the `lift` and `combine` combinators, and therefore the constraints in the types of the meta combinators also have consequences for the types of the combinators. More precisely, because in the combinator definition, a meta combinator is applied to each step in the layer, for each step, constraints will appear in the type of the combinators. However, the types of the combinators do not refer to these intermediate steps, so the types are ambiguous. As an example, we can look at the type of `lift` for `Simple`:

```
lift :: (Pack step doc pres nstep, Pack nstep gest upd step) =>
        Simple state mapping doc pres gest upd -> state -> step
```

The `Pack` constraints refer to `nstep`, which does not occur in the type of the function. The ambiguity in the types can be resolved by adding yet another functional dependency to the type class definition, stating that the next step is uniquely determined by the current step: `step -> nStep`. The final functional dependency is: `step -> arg res nStep`.

It is straightforward to see that the functional dependencies hold. Instances of `Pack` are always of the following form: (for a given type constructor `TheStep` and its next step type `NextStep`).

```
instance Pack (TheStep a r ...)  a r (NextStep ... a r) where ...
```

The dependency states that given `(TheStep a b ...)`, the types `a`, `r` and `(NextStep ...  a r)` are uniquely determined. For `a` and `r`, this is true because `a` and `r` appear in `(TheStep a r ...)`. Furthermore, because each step has a unique next step, and this next step has the same type variables as the step, the type `(NextStep ... a r)` is also determined by `(TheStep a r ...)`. Therefore, instances of `Pack` will never conflict with the functional dependency.

The class definition becomes:

```
class Pack step arg res nStep | step -> arg res nStep where
  pack :: (arg -> (res, nStep)) -> step
  unpack :: step -> arg -> (res, nStep)
```

With the use of the type class, the meta combinators change slightly. We show the changes to the definition of `combineStepDown`. The other meta combinators undergo similar changes and are shown in the next section.

The `combineStepDown` function loses the 3-tuple parameter with the pack and unpack functions, and `unpackL` and `unpackU` are replaced by `unpack`. In addition, a type constraint is added to the type.

```
combineStepDown :: ( Pack stepC h l nStepC
                   , Pack stepU h m nStepU
                   , Pack stepL m l nStepL ) =>
                   (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepDown nextStep upr lwr = pack $
    \high -> let (med, nextUpr) = (unpack upr) high
                 (low, nextLwr) = (unpack lwr) med
             in  (low, nextStep nextUpr nextLwr)
```

From the type constraint, we can see how the type class approach solves the double unpack parameter problem. Three `Pack` constraints are present in the type. The last two state that parameter types `stepU` and `stepL` are instances of `Pack`, and the first constraint states that result type `stepC` is instance of `Pack`. From this we can see that the first two constraints arise from the use of `unpack` on the `upr` and `lwr` arguments (of types `stepU` and `stepL`), whereas the last constraint comes from the use of `pack`. In the implementation, this means that three dictionaries are passed, and that the two occurrences of `unpack` come from different dictionaries. In other words, the type system takes care of passing the double unpack functions.

The type class does have a small drawback. Due to the monomorphism restriction, Haskell only allows a function to have an overloaded type if it is defined with parameters on the lefthand side of the definition (ie. `f` x_1 ... x_n `=` ..., instead of `f = \`x_1 ... x_n`->` ...), or if it has an explicit type signature. Due to the type class, the inferred types of `lift` and `combine` are overloaded. However, `combine` has no parameters on the lefthand side of the definition, so it needs to be rewritten, or it needs a type signature.

Because the type of `combine` can be complex, as the order of the parameters depends on the direction of the layer functions, it is inconvenient to have to give the type signature. If a type signature in the architecture source is desired, the compiler can be asked for the type, which can then be pasted in the source.

To eliminate the need for an explicit type signature for `combine`, we explicitly add the upper and lower layer parameters to the definition. For `Simple`, `combine` becomes:

```
combine upper lower = fix (combineStepDown . combineStepUp) upper lower
```

A more general description of the construction of `combine` is given in the next section.

7.5 Final Library and Conclusions

Figure 8 contains the final definitions of the meta combinators together with the `Pack` type class, the `fix` combinator, and the `LayerFunction` type synonym. In order to describe and implement an architecture, a number of definitions are necessary that depend on the number of layer functions and hence cannot be included in the library. We give a general description of these definitions.

General use: The general case that we consider is of a layered architecture in which the layers have n layer functions. The record type `TheLayer` contains the layer functions. The type has a number of type variables. The $\mathtt{h}_i$ variables are the types that appear in the horizontal parameters of the layer, and the $\mathtt{a}_i$ and $\mathtt{r}_i$ are the types of the vertical arguments and result. The horizontal type variables need not be equal to the types of the horizontal parameters in the layer. See for example the type `Simple`. It has two horizontal type variables (`mapping` and `state`), but the horizontal parameters in the layer are `state` and `(mapping, state)`. As a result, the number of horizontal type variables (m) is not necessarily equal to the number of layer functions (n), and the types of the horizontal arguments and results in the `layerFunction` types below have to be denoted with `(...)`.

```
data TheLayer h₁ ... hₘ a₁ r₁ ... aₙ rₙ =
       TheLayer  { layerFunction₁ :: LayerFunction (...)  a₁ (...)  r₁
                   ...
                   layerFunctionₙ :: LayerFunction (...)  aₙ (...)  rₙ }
```

First, we define the `Step` type synonym that captures the explicit function type and rotation of parameters in the subsequent data definitions:

```
type Step next a₁ r₁ ... aₙ rₙ = (a₁ -> (r₁, next a₂ r₂ ... aₙ rₙ a₁ r₁))
```

Using `Step` the mutually recursive data definitions for each of the steps are straightforward. Each definition $\mathtt{Step}_i$ contains a reference to the next definition $\mathtt{Step}_{i+1}$, except for $\mathtt{Step}_n$, which contains a reference to the first step: $\mathtt{Step}_1$. Note that write the type definition with the Haskell record syntax to define the $\mathtt{step}_i$ selector functions.

```
newtype Step₁ a₁ r₁ ... aₙ rₙ = Step₁ {step₁ :: Step Step₂ a₁ r₁ ... aₙ rₙ}
newtype Step₂ a₁ r₁ ... aₙ rₙ = Step₂ {step₁ :: Step Step₃ a₁ r₁ ... aₙ rₙ}
...
newtype Stepₙ a₁ r₁ ... aₙ rₙ = Stepₙ {step₁ :: Step Step₁ a₁ r₁ ... aₙ rₙ}
```

The type synonym `Layer` is introduced for the type $\mathtt{Step}_1$:

```
type Layer a₁ ... a₂ₙ = Step₁ a₁ ... a₂ₙ
```

```
fix :: (a->a) -> a
fix a = let fixa = a fixa
         in fixa

type LayerFunction horArgs vertArg horRess vertRes =
       horArgs -> vertArg -> (vertRes, horRess)

class Pack step arg res nStep | step -> arg res nStep where
  pack :: (arg -> (res, nStep)) -> step
  unpack :: step -> arg -> (res, nStep)

liftStep :: Pack step vArg vRes nStep =>
            (hArgs -> vArg -> (vRes,hRess)) ->
            (hRess -> nStep) ->
            hArgs -> step
liftStep layerF next horArgs = pack $
    \vertArg -> let (vertRes, horRess) = layerF horArgs vertArg
                in  (vertRes, next horRess)

combineStepDown :: ( Pack stepC h l nStepC
                   , Pack stepU h m nStepU
                   , Pack stepL m l nStepL ) =>
                   (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepDown nextStep upr lwr = pack $
    \high -> let (med, nextUpr) = (unpack upr) high
                 (low, nextLwr) = (unpack lwr) med
             in  (low, nextStep nextUpr nextLwr)

combineStepUp :: ( Pack stepC l h nStepC
                 , Pack stepU m h nStepU
                 , Pack stepL l m nStepL ) =>
                 (nStepU -> nStepL -> nStepC) -> stepU -> stepL -> stepC
combineStepUp nextStep upr lwr = pack $
    \low -> let (med, nextLwr) = (unpack lwr) low
                (high, nextUpr) = (unpack upr) med
            in  (high, nextStep nextUpr nextLwr)
```

Figure 8: Final meta combinators for `lift` and `combine`

All `Step`$_i$ types are made instances of `Pack` with n instance declarations of the following form:

```
instance Step_i (Step_i a_1 r_1 ... a_n r_n) a_1 r_1 (Step_{i+1} a_2 r_2 ... a_n r_n a_1 r_1)
 where pack = Step_i
       unpack = step_i
```

The only thing left to do now, is to define the `lift` and `combine` combinators. For the lift combinator, we need to apply `liftStep` to each of the layer functions, compose the steps and apply `fix` to the composition.

```
lift :: TheLayer h_1 ... h_m a_1 r_1 ... a_n r_n -> Layer a_1 r_1 ... a_n r_n
lift theLayer =
  fix $ liftStep (layerFunction_1 theLayer)
      ...
      . liftStep (layerFunction_n theLayer)
```

The `combine` combinator is a composition of either `combineStepUp` or `combineStepDown` meta combinators to which `fix` is applied. The choice of `combineStepUp` or `combineStepDown` for a layer function is determined by the direction of the vertical data flow in that layer. Note that the `upper` and `lower` arguments are added because of the monomorphism restriction. The type of combine cannot be given in a general definition and is explained below.

```
combine :: Layer ... -> Layer ... -> Layer ...
combine upper lower =
  fix ( combineStep<Up/Down>
      ...
      . combineStep<Up/Down>
      ) upper lower
```

A general type definition of combine cannot be given, because the order of the type variables depends on the direction of the layer functions. For each pair of type variables, a_i and r_i in the `Layer ...` a_i r_i `...` types of the two arguments and the result, we have the following dependency. If the i-th layer function is downward:

```
Layer ... h m ... -> Layer ... m l ...  -> Layer ... h l ...
```

And if the layer function is upward:

```
Layer ... m h ... -> Layer ... l m ...  -> Layer ... l h ...
```

However, instead of computing the type by hand, the compiler can be asked for the type which can then be pasted in the source.

This concludes the definitions that are necessary to be able to describe an architecture.

Simple editor: As an example application of the library, we give the definitions that are required for implementing the simple editor architecture. Recall that the data type containing the layer functions is:

```
data Simple state mapping doc pres gest upd =
       Simple { present ::   LayerFunction state doc (mapping, state) pres
              , translate :: LayerFunction (mapping, state) gest state upd
              }
```

The simple editor layer contains two layer functions, so we need to define two step data types with corresponding `pack` and `unpack` functions. First we define the `Step` type synonym and the step data types. The layer has two layer functions, so the number of type variables is four.

```
type Step next a b c d = (a ->(b, next c d a b))

newtype PresStep doc pres gest upd =
            PresStep {presStep :: Step TransStep doc pres gest upd}
newtype TransStep gest upd doc pres =
            TransStep {transStep :: Step PresStep gest upd doc pres}
```

Each pair of steps must be an instance of `Pack`:

```
instance Pack (PresStep a b c d) a b (TransStep c d a b) where
  pack = PresStep
  unpack = presStep

instance Pack (TransStep a b c d) a b (PresStep c d a b) where
  pack = TransStep
  unpack = transStep
```

The definitions of `lift` and `combine` are now rather trivial:

```
lift simple =
  fix $ liftStep (present simple) . liftStep (translate simple)

combine =
  fix $ combineStepDown . combineStepUp
```

The definitions of the `main` and `editLoop` functions are the same as in Section 6 so they are not reproduced here.

Conclusions:

The simplicity of the definitions of the `lift` and `combine` combinators leads to the question whether it is possible to define generic `lift` and `combine` combinators with the use of a type class that contains the step specific aspects. This would eliminate the need to explicitly define the combinators. It would be sufficient to define the step data types and make them instances of the type class.

Although it is possible to define the type class, it turns out not te be very useful. The generic combinator takes an argument of a certain step type and contains a recursive call to itself on the next step type. Because of this recursive call, a type constraint on the step, for example that it is an instance of `Wrap`, must also hold for the next step. But since the recursive call generates a constraint for its next step too, the constraint must also hold for the step after the next step, and so on. As a result, in the type of the combinator we have to explicitly give the type constraints on each of the step types. However, this means that the type of the combinator depends on the number of steps, and we do not want to encode the number of steps in the library. Therefore, we will use the simpler approach of constructing `lift` and `combine` explicitly.

The meta combinator library has the same advantages as the direct parameter passing solution of Section 6, but at the same time, it is much easier to describe a specific architecture. The use of meta combinators makes the data flow clearer and reduces the chance of errors in the specification.

8 The Proxima editor

The motivation for using Haskell to describe editor architectures has come from the Proxima project [7]. Proxima is a generic incremental XML editor, which is being developed at Utrecht University. The editor is parameterized with a presentation sheet that specifies how a document is presented. Furthermore, it is parameterized with a computation sheet that specifies derived values, such as chapter numbers, or a table of contents. A user can only see and edit the final presentation of the document. Because the mapping of the document on the presentation is a stepwise process, and hence the translation from edit commands on the presentation to updates on the document as well, Proxima has a layered architecture.

Figure 9 schematically shows the layers of Proxima. The figure shows the layers (surrounded by rectangles) as well as the data values that arise in the process of mapping of the document on the presentation. Edit actions and incremental updates are not shown in the figure.

The *document* at the top of the architecture is mapped on the *rendering* at the bottom, which is shown to the user. The *evaluator* takes the document and a computation sheet and

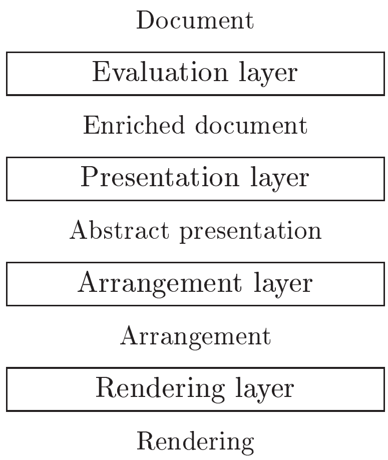

Figure 9: The layers of Proxima

produces the *enriched document*, which is the document together with all derived values. Similarly, the *presenter* takes the enriched document together with a presentation sheet, and computes the *abstract presentation*. Until this point, the presentation does not contain any absolute layout; all elements in the presentation are positioned relative to each other and do not yet have an absolute size.

The absolute coordinates are computed by the *arranger*. It maps the abstract presentation on an *arrangement*, which is a absolute positioning of all elements of the presentation. The arranger also takes care of hyphenation, line and page breaking. Finally, the arrangement is rendered as a bitmap by the *renderer*.

A prototype of the Proxima editor has been implemented with the architecture combinators of the previous section. The module that contains the architecture description is an actual part of the implementation of the prototype.

9 Conclusions

The combinators presented in this paper make it possible to specify layered editor architectures in a concise and transparent way. With a small number of definitions, a layered architecture can be described. The combinators have been tested with an actual architecture and have been heap profiled to ensure that no memory leaks are present.

Furthermore, because the architecture description language is embedded in the implementation language, the architecture of a system forms part of the implementation of the system. No translations from the ADL to the implementation language is required, and the implementation is guaranteed to comply with the architecture.

The combinator language in this paper is tailored to a specific kind of architectures: those of layered editors. Although we use the term editor in a broad sense, also including spreadsheets, e-mail agents, etc., further research should explore the possibilities of using Haskell to describe other kinds of architectures. Another area of research concerns the dynamic aspects of the architecture: how can constraints and invariants on the data be described and, if possible, verified?

References

[1] L. Augustsson. Cayenne - a language with dependent types. *Proceedings of the ICFP '98, ACM SIGPLAN Notices,* 34(1). 1999, pp. 239-50.

[2] P. Hudak. Modular Domain Specific Languages and Tools. In I*nternational Conference on Software Reuse, ICSR 98*, 1998

[3] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming,* E-SOP 2000, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.

[4] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering,* 26(1). 2000.

[5] S. Peyton Jones and J. Hughes. *Haskell 98: A Non-strict, Purely Functional Language.* February 1999. `http://www.haskell.org/onlinereport/`

[6] M.M. Schrage, J. Jeuring, D. Swierstra, and L. Meertens. Layered Software Architectures for Editors, 2002, In preparation

[7] M.M. Schrage, J. Jeuring, D. Swierstra, and L. Meertens. *Proxima, a generic incremental XML editor,* 1999. `http://www.cs.uu.nl/research/projects/proxima/`