Haskell as an Architecture Description Language

Martijn M. Schrage

S. Doaitse Swierstra

Technical Report UU-CS-2008-045
December 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Haskell as an Architecture Description Language

Martijn M. Schrage S. Doaitse Swierstra
Utrecht University
{martijn,doaitse}@cs.uu.nl

Abstract

We define a domain specific embedded language in Haskell for describing layered software architectures which maintain bidirectional dependencies. 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. Because, contrary to the situation with typical Architecture Description Languages, the description is part of the implementation of the system, the implementation is guaranteed to comply with the architecture, and the architecture is guaranteed to comply with the implementation.

1. Introduction

In this paper we develop a set of Haskell combinators for describing architectures of layered systems. Although designed with a layered editor in mind, we claim this approach to be applicable to many kinds of layered architectures. The combinators have been successfully used in the implementation of the generic editor Proxima (Schrage 2004), as well as for a system providing web-interfaces to databases.

Describing an architecture in an real programming language not only makes it possible to describe the interfaces of the components of the system, and how they are to be composed, but also enables us to smoothly extend this high-level description into a real implementation, without having to maintain several views on our system, with all its problems of diverging versions. Because the architecture description in Haskell 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 (Medvidovic and Taylor 2000), 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 architecture description languages from, amongst others, programming languages. However, Haskell offers possibilities for describing the main components of an architecture, while incorporating these components as part of the program itself. 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.

As argued by Hudak (Hudak 1998), higher-order typed functional languages offer excellent possibilities for embedding domain-specific languages. Embedding a domain-specific language facilitates reuse of syntax, semantics, implementation code, software tools, as well as look-and-feel. In this paper we develop 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.

In this paper we give three implementation models for layered architectures. First, in Section 2, we introduce a simplified layered editor architecture and, in section 3, 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 as a basis in Section 5 as a base to develop a more abstract combinator implementation that uses nested pairs. 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. The combinators from this section require two minimal function definitions for instantiating a specific architecture. In Section 8, we exploit the type system to automatically provide these definitions, based solely on the types of the layer interface. Section 9 shows how the architecture combinators are used to describe (and implement) the architecture of the Proxima editor. And, finally, Section 10 discusses future work and concludes.

2. A simple editor

The architecture description combinators in this paper have been designed for the generic presentation-oriented structure editor Proxima. Proxima supports editing on the document structure as well as on its presentation, which has given rise to its layered architecture. A typical feature of the architecture is that each layer maintains its own local state, which is used to store information that does not have a logical place on the other layers. An example of this is white space, which is part of the presentation of a program source, but does not fit well in the abstract syntax tree.

Because of the complexity of the actual Proxima architecture, we introduce a simple layered architecture for a presentation-oriented editor to explain the architecture description methods in the next sections. Although the architecture is simple, it contains the essential features of the Proxima architecture.

Figure 1 depicts the editing process for our simple editor. The editor keeps track of a document (`doc`), which is mapped onto a presentation (`pres`). The presentation process is split into n steps: $\texttt{present}_1 \ldots \texttt{present}_n$, each step takes care of a specific subtask, such as computing a set of layout alternatives, computing minimal and maximal sizes, negotiation between objects to be displayed about the available screen space, keeping track of white space which was explicitly entered by the user but which does not play a role at the document level and keeping track of explicit representation choices –toggling visibility, alignment, unfolding status– made by the user in the course of the editing process. At the bottom of the figure, the presentation is shown to a user, who provides an edit gesture (`gest`) in response. The edit gesture is mapped onto a document update (`upd`) by $\texttt{interpret}_n \ldots \texttt{interpret}_1$, which is then applied to the document. In the next cycle, the updated document is presented again. Note that some of the editing gestures

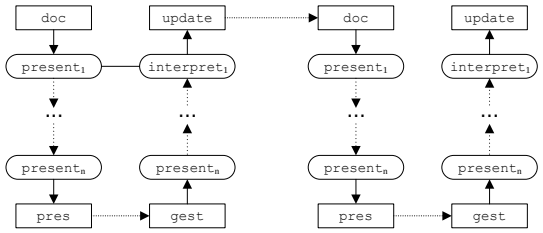

Figure 1. Two cycles in the editing process.

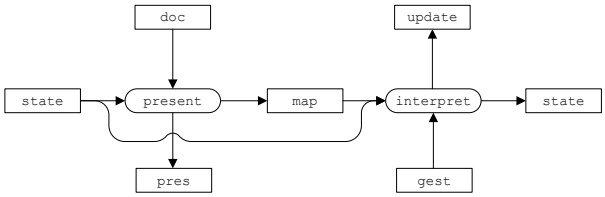

Figure 2. A single layer.

effectively turn out to be identity operations at the top level, since some of them will be handled at an intermediate level, and will result in an update of the state maintained at that level.

A layer consists of a pair of `present`$_i$ and `interpret`$_i$ functions, which we refer to as *layer functions*. Besides the vertical data flow for presentation and interpretation, we thus may also have horizontal data flow that stays within the layer. Horizontal data flow is used to maintain state in a layer; this state component is passed along between all computations taking place in a layer.

Figure 2 shows the data flow for a single layer with two examples of horizontal data flow. The result `map` of the function `present` is passed on to `interpret` and represents information about where things are mapped on the screen. Furthermore, a `state` parameter is passed to `present` as well as `interpret`, and may be updated by `interpret`. Note that the `state` parameter of `present` is the result of `interpret` in the previous edit step. Because of the sequential nature of the edit steps, we only consider horizontal data flow that goes from left to right.

3. A layer in Haskell

In the next sections, we explore the possibilities of describing the layered architecture from the previous section in Haskell. 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. Before discussing how to model the connections between the layers, we focus on the functions within a layer.

A layer function takes horizontal as well as vertical arguments and returns both horizontal and vertical results. To make the difference between horizontal and vertical data explicit, we introduce a type synonym for layer functions.

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

Each layer is represented by a record containing all the layer functions: in our case `present` and `interpret`. The types of the layer functions follow directly from Figure 2. If we put these functions directly in a record, we get:

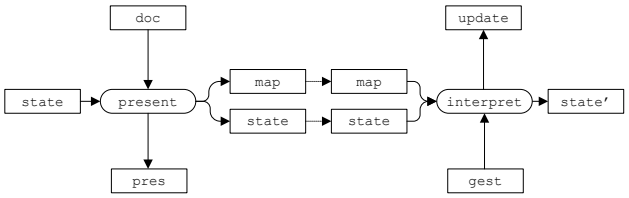

Figure 3. Data flow in a normalized layer.

```
data Simple state map doc pres gest upd =
       Simple { present   :: LayerFn state doc
                                     map pres
              , interpret :: LayerFn (map,state) gest
                                     state upd
              }
```

However, this is not entirely what we want. To simplify the horizontal connection between layer functions, we prefer a normalized data type in which the horizontal result type (`horRes`) of a layer function matches the horizontal argument type (`horArg`) of the next layer function. For our example, this implies that the horizontal result of `present` has the same type as the horizontal argument of `interpret` and vice versa (since the result of `interpret` is the argument of `present` in the next edit cycle). Figure 3 shows the data flow for the layer functions in the normalized type `Simple`. Because the conversion to a normalized type is straightforward, we do not show it here. The definition of the normalized `Simple` is:

```
data Simple state map doc pres gest upd =
       Simple { present   :: LayerFn state doc
                                     (map, state) pres
              , interpret :: LayerFn (map, state) gest
                                     state upd
              }
```

Although a `Simple` layer consists of two layer functions, the final combinator library presented in this paper abstracts over this number and can be used for layers with arbitrary numbers of layer functions.

4. Method 1: Explicitly connecting the components

Now that the layers have been modeled, we need to realize the vertical data flow by connecting the layer functions. The document must be fed into the layers at the top, yielding the presentation at the bottom, and similarly, the edit gesture must be fed into the bottom layer, yielding the document update. The most straightforward way of tying everything together is to explicitly write down the selection and application of each of the functions in each of the layers. This will be the first approach, followed by gradually more abstract approaches in sections 5 to 7.

We give an example edit loop that explicitly connects three layers: `layer1`, `layer2`, and `layer3` of type `Simple`. The data flow between the layer functions is shown in Figure 4. Note that we might look at the code below as a textual representation of this picture, and that vice versa we might compute such a picture out of the code. At the bottom of the figure, the presentation is shown to the user, and an edit gesture is obtained, which we represent in the code with two functions `showRendering :: Rendering -> IO ()` and `getGesture :: IO Gesture`. At the top of the figure, the document is updated, which we model with a function `updateDocument :: Update -> Document -> IO Document`. The Haskell code for the edit loop is:

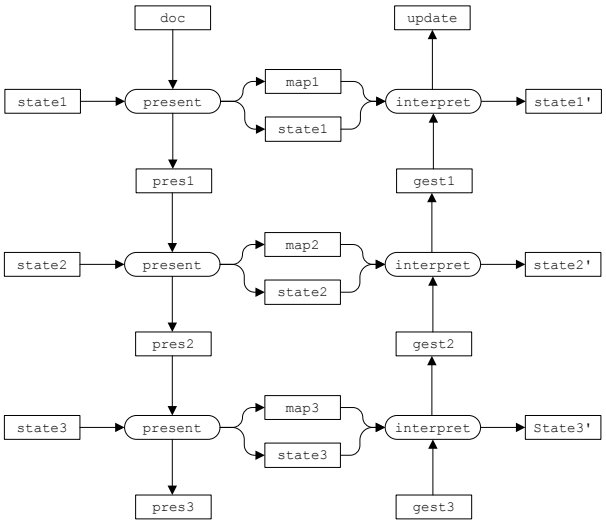

Figure 4. Data flow in and between layers.

```
editLoop (layer1, layer2, layer3) states doc =
  loop states doc where
  loop (state1, state2, state3) doc =
   do { let (pres1,(map1,state1')) =
              present layer1 state1 doc
      ; let (pres2,(map2,state2')) =
              present layer2 state2 pres1
      ; let (pres3,(map3,state3')) =
              present layer3 state3 pres2

      ; showRendering pres3
      ; gest3 <- getGesture

      ; let (gest2,state3'')  =
              interpret layer3 (map3, state3') gest3
      ; let (gest1,state2'')  =
              interpret layer2 (map2, state2') gest2
      ; let (update,state1'') =
              interpret layer1 (map1, state1') gest1

      ; let doc' = updateDocument update doc
      ; loop (state1'', state2'', state3'') doc'
      }
```

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

```
main layer1 layer2 layer3 =
 do { states <- initStates
    ; doc <- initDoc
    ; editLoop (layer1, layer2, layer3) 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 each implementing a different layer. 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 state parameters are horizontal parameters and that the presentation is a vertical parameter is not immediately clear from the program code. Moreover, explicitly encoding the standard patterns for upward and downward vertical parameters, increases the chance of errors. 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. However, type checking will not detect two equally typed variables accidentally being swapped. the goal of the coming sections is to gradually abstract from the current, very explicit description of the problem.

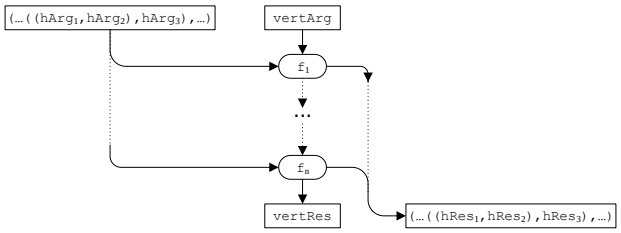

Figure 5. Horizontal nested pairs for a downward step.

5. Method 2: Nested pairs

In this section we abstract from the horizontal and vertical dataflow patterns in the edit loop of the previous section, by using combinators for combining layers. In the main loop, we call the layer functions of the combined layer, rather than explicitly calling each layer function in the main loop. The combinators also make the data flow more explicit. The direction of the vertical parameter is made apparent by the choice of combinator, rather than explicitly threading it through the function applications.

Similar to function composition ($f \cdot g$), we develop a `combine` combinator that takes two layers and returns a combined layer. The layer functions of the combined layer are 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 `combine` 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 `LayerFn horArgC vertArg horResC vertRes`. The parameters `horArgC` and `horResC` stand for the types of the collections of horizontal parameters and results. Figure 5 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 of the same type, we cannot use a list to represent the collections. Moreover, we wish to be able to determine at compile time whether the collection contains the required 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. Hence, we use a nested cartesian product to represent the horizontal parameters and results.

We only use left-associatively nested products in this section: $(\ldots((e_1,e_2),e_3),\ \ldots),\ e_n)$, although this will not be enforced by the combinators; as long as the structure of the argument and result products is the same, which is guaranteed by the way the `combine` combinator is used, the precise structure does not matter.

We first define two combinators for composing layer functions: a downward combinator `composeDown` (for `present`) and a upward combinator `composeUp` (for `interpret`). A downward vertical parameter passes through the higher layer first, whereas an

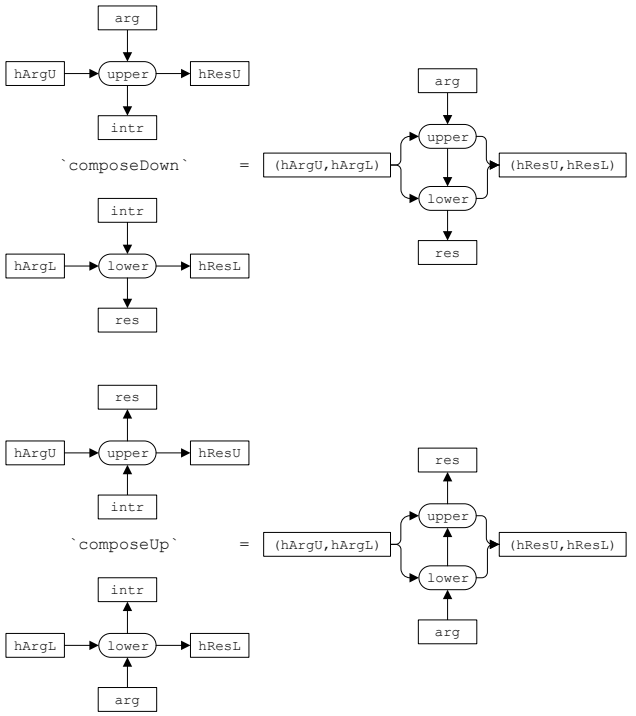

Figure 6. `composeDown` and `composeUp`

upward vertical parameter passes through the lower layer first. Figure 6 shows the data flow for the two combinators.

The combinator `composeDown` composes two layers `higher` and `lower` by feeding the intermediate vertical result of `h` into `l`. At the same time, the horizontal parameters for `higher` and `lower` 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 `higher` and `lower`.

```
composeDown :: LayerFn horArgU arg horResU intr ->
               LayerFn horArgL intr horResL res ->
               LayerFn (horArgU, horArgL) arg
                       (horResU, horResL) res
composeDown upper lower =
  \(horArgU, horArgL) arg ->
    let (interm, horResU) = upper horArgU arg
        (res, horResL)    = lower horArgL interm
    in  (res, (horResU,horResL))
```

The definition of `composeUp` is analogous to `composeDown`. Its type is:

```
composeUp :: LayerFn horArgU intr horResU res ->
             LayerFn horArgL arg horResL intr ->
             LayerFn (horArgU, horArgL) arg
                     (horResU, horResL) res
```

Combining layers

Using `composeUp` and `composeDown`, we can now define a combinator to combine `Simple` layers.

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 use type `Simple` to represent a combined layer, because the types for the horizontal parameters do not match. For `Simple`, the horizontal parameter of `present` has type `state`, and its result has type `(map, state)`. In contrast, the horizontal parameter of `present` in the composed layer is a nested pair of states and its result is a nested pair of `map` and `state` tuples.

We introduce a type `LayerC` as a more general version of `Simple`. Because we cannot easily denote a nested pair structure in a Haskell type declaration, we leave the structure of the horizontal parameters unspecified. The parameter `states` represents the nested pair of `state` values, and the parameter `mapsStates` represents the nested pair of `map` and `state` tuples.

```
data LayerC states mapsStates doc pres gest upd =
       LayerC { presentC ::
                   LayerFn states doc mapsStates pres
              , interpretC ::
                   LayerFn mapsStates 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
         , interpretC = interpret 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)
         , interpretC = composeUp (interpretC upper)
                                  (interpretC 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 { let (pres, mapsStates) =
                   presentC layers states doc

           ; showRendering pres
           ; gest <- getGesture

           ; let (update, states') =
                   interpretC layers mapsStates 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 layer1 layer2 layer3 =
 do { (state1, state2, state3) <- initStates
    ; doc <- initDoc
    ; let layers = lift layer1 `combine` lift layer2
                               `combine` lift  layer3
    ; editLoop layers ((state1, state2), state3) doc
    }
```

Conclusions

The nested pairs solution is more compositional than the approach of the previous section and most of the data flow is hidden

from the main loop. However, the 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. Moreover, the type of the composite layer is parameterized with all the types appearing in the layers, leading to large type signatures.

6. Method 3: Hidden parameters

In the previous section, the horizontal results that are computed on evaluation of a combined layer function are returned explicitly and passed as arguments to the next layer function. In this section we take an alternative approach, which we explain with an example. Recall that with the nested pairs method, each combined layer function returns a collection of horizontal results together with its vertical result:

```
...
let (pres, mapsStates) = presentC layers states doc
...
let (update, states') = interpretC layers mapsStates gest
...
```

In contrast, the hidden-parameter method does not return a collection of horizontal results, but a function that has already been partially applied to the horizontal results; the state is stored in the closure which is thus formed.

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

The code that is shown is not entirely accurate, but it gives the general idea. Together with the presentation, `presentStep` returns a function `interpretStep` for computing the document update. The layer functions in `interpretStep` have already been partially applied to the horizontal results from the presentation step. Thus, the horizontal parameters are now entirely hidden from the main loop, the main editor loop becomes more transparent and the type of a combined layer becomes simpler since the types for the horizontal parameters are internalized.

Type definitions

The type of the layer is described by the following type: `(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. Hence, we need to use a `newtype` declaration, with the disadvantage that values of the type have to be wrapped with constructor functions.

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

We now define two combinators for constructing and combining `Layer` values: `lift` converts a `Simple` layer to a hidden-parameter layer of type `Layer`, and `combine` combines two layers of type `Layer`. 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.

Definition of `lift`

The combinator `lift` takes a `Simple` layer and returns a `Layer`:

```
lift :: Simple state map doc pres gest upd ->
        state -> Layer doc pres gest upd
lift simple state = presStep state
 where presStep state = Layer $
         \doc ->  let (pres, (map,state)) =
                            present simple state doc
                  in  (pres, intrStep (map,state))
       intrStep (map,state) =
         \gest -> let (upd, state') =
                            interpret simple (map,state) gest
                  in  (upd, presStep state')
```

Besides `layer`, `lift` gets a second parameter, `init`, which is the initial value of the horizontal parameter. 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`. Thus, once the initial 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 between the layer functions, and also to the next edit cycle.

Definition of `combine`

To combine layers, we define a combinator `combine`, which gets two layers as arguments: a higher layer and a lower layer. The type of `combine` is:

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

The reason for the order of the type variables is that for each pair of variables, the first type is an argument type and the second type a result type. Hence, the first step in the combined layer is a function `high -> low`, which is the composition of a function `high -> med` in the higher layer and a function `med -> low` in the lower. On the other hand, the second step goes upward. Thus, the function `elow -> ehigh` in the combined layer is the reverse composition of functions `elow -> emed` and `emed -> ehigh` in the higher and lower layers

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`.

```
combine = presStep
 where presStep (Layer upr) (Layer lwr) = Layer $
           \high -> let (med, uprIntr) = upr high
                        (low, lwrIntr) = lwr med
                    in  (low, intrStep uprIntr lwrIntr)
       intrStep upr lwr =
           \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 contains references to the horizontal parameters. Furthermore, the combined layer is called `presentStep` instead of `layers` to reflect that it represents the presentation step of the computation.

```
editLoop (Layer presentStep) doc =
 do { let (pres , interpretStep) = presentStep doc

    ; showRendering pres
    ; gesture <- getGesture

    ; let (update, presentStep') = interpretStep 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 layer1 layer2 layer3) =
 do { (init1, init2, init3) <- initStates
    ; doc <- initDoc
    ; let layers = lift layer1 init1 `combine`
                   lift layer2 init2 `combine`
                   lift layer3 init3
    ; editLoop layers doc
    }
```

Conclusions

The hidden-parameter 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. Thus, both horizontal and vertical data flow are made more transparent.

7. Developing a library for architecture descriptions

In this section, we develop a small library for constructing `lift` and `combine` combinators for layered architectures having an arbitrary number of layer functions. We refer to these combinators as *meta combinators* because they are used to construct combinators.

The `lift` and `combine` combinators from the previous section are just one case of a layered architecture: a layer with two layer functions and hence two *step*s. Even though the combinators are straightforward, some code is duplicated, and small errors are easily made. Therefore, instead of a guideline on how to write `lift` and `combine` by hand, we prefer a small library of meta combinators for constructing these combinators. A further advantage of having a meta-combinator library is that instead of explicitly encoding the direction for each of the steps in the `combine` function, we can use the name of the meta combinator to reflect the direction in which the data flows (like with the composeUp/Down functions from the nested pairs method in Section 5).

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 `intrStep`.

We derive the meta combinators by starting with the combinators from the previous section and gradually factoring out all step-specific aspects. We end with a combinator that is constructed out of a collection of simple building blocks (or meta combinators).

7.1 Type definitions

The `Layer` type poses a problem if we want to construct a library for building `lift` and `combine` functions, since somehow its constructors need to be added to and removed by the combinators. One solution is to create a type class for the constructor and deconstructor functions, but this complicates the types and requires a user to provide an instance of this class. Therefore, we introduce a compositional representation of a layer type that makes use of simple types defined in the library.

If we inspect the `Layer` type from the Section 6, we see it is made up of two steps of the form `vArg -> (vRes, ...)`. We capture this in the following type:

```
newtype Step a b ns = Step (a -> (b, ns))
```

In order to compose steps, we define an infix, right-associative, type constructor `(:.:)`. The reason for right associativity will become apparent in Section 7.2.

```
infixr :.:
newtype (:.:) f g ns  = Comp (f (g ns))
```

We also define a `NilStep` as the starting point for a series of compositions:

```
newtype NilStep t = NilStep t
```

Now, for example, we can encode the type `Doc -> (Pres, Gest -> (Upd, next))` as `(Step Doc Pres :.: Step Gest Upd :.: NilStep) next`. To encode the feedback loop, we introduce a fixed-point type `Fix`:

```
newtype Fix f = Fix (f (Fix f))
```

With these type combinators, we can now express `Layer` in a compositional and point-free way:

```
type Layer doc pers gest upd =
  Fix (Step doc pres :.: Step gest upd :.: NilStep)
```

Because `Step`, `NilStep`, and `:.:` appear partially parameterized in the layer type, and `Fix` is recursive, all three types need to be introduced using `newtype` definitions. Hence, instead of having a single `Layer` constructor, `lift` and `combine` will be littered with constructors. This is not a problem, however, since the combinators we will derive in the next subsections take care of adding and removing these constructors.

The reason why we have an explicit `NilStep` with yet another constructor is that it causes the number of occurrences `(:.:)` to be the same as the number of steps, which will facilitate the removal of `Comp` constructors. Furthermore, as we will see in Section 8, the `NilStep` will also provide a base for recursive instances, preventing overlapping instances.

7.2 Derivation for `lift`

First we develop a meta combinator for the `lift` function. We start with the code for `lift` for a layer with two steps from Section 6. If we rename several variables and adapt the code for the new constructor-rich representation of the `Layer` type, we get:

```
lift simple state = step1 state
 where step1 hArg = Fix . Comp . Step $
        \vArg -> let (pres, hRes) =
                       present simple hArg vArg
                 in  (pres, step2 hRes)
       step2 hArg = Comp . Step $
         \vArg -> let (upd, hRes) =
                        interpret simple hArg vArg
                  in  (upd, lNilStep hRes)
       lNilStep hRes = NilStep $ step1 hRes
```

The definitions of the local functions `step1` and `step2` contain mutually-recursive references, thus hard-coding their position in the sequence of steps. We eliminate this positional information in the definitions by supplying the next step as a parameter to each function. The `lNilStep` no longer needs to be a local function.

```
lift :: Simple state map doc pres gest upd ->
        state -> Layer doc pres gest upd
lift simple state =
  step1 (step2 (lNilStep (lift simple))) state
 where step1 next hArg = Fix . Comp . Step $
        \vArg -> let (pres, hRes) =
                       present simple hArg vArg
```

```
                in  (pres, next hRes)
      step2 next hArg = Comp . Step $
        \vArg -> let (upd, hRes) =
                       interpret simple hArg vArg
                in  (upd, next  hRes)

lNilStep next hRes = NilStep $ next hRes
```

Next we capture the `Comp` and `Step` constructors and the lambda expression with the function `liftStep`. Here, it becomes apparent why composition is right associative, since each step has both a`Comp` constructor and a `Step` constructor. If composition were left-associative, the `Comp` constructors would end up between the `Fix` and `Step` constructors, and it would be harder to capture the pattern.

```
liftStep f next horArgs = Comp . Step $
  \vArg -> let (vertRes, horRes) = f horArgs vArg
           in  (vertRes, next horRes)
```

leading to the new definition of `lift`:

```
lift simple state =
  step1 (step2 (lNilStep (lift simple))) state
 where step1 next hArg =
         Fix $ liftStep (present simple) next hArg
       step2 next hArg =
         liftStep (interpret simple) next hArg
```

If we drop the `state` parameter on the first two lines and rewrite the function application as a composition, we get:

```
lift layer = (step1 . step2 . lNilStep) (lift layer)
  ...
```

Capturing the recursion pattern with the `fix` combinator:

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

we get our next version of `lift`:

```
lift simple = fix (step1 . step2 . lNilStep)
 where step1 next hArg = Fix $
         liftStep (present simple) next hArg
       step2 next hArg =
         liftStep (interpret simple) next hArg
```

Regardless the number of steps, there will only be one `Fix` constructor (in contrast to the number of `Comp` and `Step` constructors, which are equal to the number of steps). Hence, we can define a function `lfix` to add this constructor. The function `lfix` also composes the `lNilStep` with the steps.

```
lfix f = fix f' where f' n = Fix . (f . lNilStep) n

lift simple = lfix (step1 . step2)
 where step1 next hArg =
         liftStep (present simple) next hArg
       step2 next hArg =
         liftStep (interpret simple) next hArg
```

Now we got rid of the all the constructors in the local step definitions, we can drop the `next` and `args` parameters and give a point-free definition:

```
lift :: Simple state map doc pres gest upd ->
        state -> Layer doc pres gest upd
lift simple = lfix $ liftStep (present simple)
                   . liftStep (interpret simple)
```

`liftStep` works for an arbitrary number of steps

For an n-step layer, the definition of `lift` (using the method from Section 6) has n local step functions, each containing a reference to the next. For such a layer, we can perform exactly the same steps as for the 2-step `lift`. The resulting `lift` contains a composition of n `liftStep` applications. If we denote the step type constructors by $\texttt{Step}_i$ and the layer functions by $\texttt{layerFn}_i$, we get:

```
lift layer =  lfix $ liftStep (layerFn_1 layer)
              ...
              .      liftStep (layerFn_n layer)
```

7.3 Derivation for `combine`

The derivation of the meta combinators for `combine` is largely similar to the derivation for `lift`. We again start with the original definition of `combine` for a two-step layer, adapt it to the new `Layer` type and rename several variables:

```
combine upr lwr = step1 upr lwr
 where step1 (Fix (Comp (Step upr)))
             (Fix (Comp (Step lwr))) =
         Fix . Comp . Step $
         \high -> let (med, uprIntr) = upr high
                      (low, lwrIntr) = lwr med
                  in  (low, step2 uprIntr lwrIntr)
       step2 (Comp (Step upr)) (Comp (Step lwr)) =
         Comp . Step $
         \low -> let (med, lwrPres) = lwr low
                     (high, uprPres) = upr med
                 in  (high, cNilStep uprPres lwrPres)
       cNilStep (NilStep u) (NilStep l) =
         NilStep $ step1 u l
```

The explicit mutual recursion in the local functions is removed by passing the next step as a parameter, and rewriting the whole function as a fixed point. The `cNilStep` becomes a top-level function.

```
combine upr lwr = fix (step1 . step2 . cNilStep) upr lwr
 where step1 next (Fix (Comp (Step upr)))
                  (Fix (Comp (Step lwr))) =
         Fix . Comp . Step $
         \high -> let (med, uprIntr) = upr high
                      (low, lwrIntr) = lwr med
                  in  (low, next uprIntr lwrIntr)
       step2 next (Comp (Step upr)) (Comp ( Step lwr)) =
         Comp . Step $
         \low -> let (med, lwrPres) = lwr low
                     (high, uprPres) = upr med
                 in  (high, next uprPres lwrPres)

cNilStep next (NilStep u) (NilStep l) =
  NilStep $ next u l
```

Without the explicit recursive calls, we can capture the vertical data flow patterns with two functions `combineStepDown` and `combineStepUp`:

```
combineStepDown :: (f x -> g y -> h ns) ->
                   (Step a b :.: f) x ->
                   (Step b c :.: g) y ->
                   (Step a c :.: h) ns
combineStepDown next (Comp (Step upper))
                     (Comp (Step lower)) = Comp . Step $
  \h -> let (m ,upperf) = upper h
            (l, lowerf) = lower m
        in  (l, next upperf lowerf)

combineStepUp :: (f x -> g y -> h ns) ->
                 (Step b c :.: f) x ->
                 (Step a b :.: g) y ->
                 (Step a c :.: h) ns
combineStepUp next (Comp (Step upper))
                   (Comp (Step lower)) = Comp . Step $
```

```
  \l -> let (m, lowerf) = lower l
            (h, upperf) = upper m
        in  (h, next upperf lowerf)
```

If we use these two functions, and drop the parameters to `step2`, we get:

```
combine upr lwr = fix (step1 . step2 . cNilStep) upr lwr
 where step1 next (Fix upr) (Fix lwr) =
         Fix $ combineStepDown next upr lwr
       step2 = combineStepUp
```

The second step is now simply a `combineStepUp`, but the first step still contains a `Fix` constructor. In order to get rid of it, we first rewrite `combine` to make the pattern more apparent:

```
combine = fix (\n (Fix u) (Fix l) ->
  Fix $ (step1 . combineStepUp . cNilStep) n u l)
 where step1 next upr lwr =
         combineStepDown next upr lwr
```

Now we can define a function `cfix` that pattern matches on the arguments and adds a `Fix` to the result. Similar to `lfix`, it also adds the `cNilStep`.

```
cfix f = fix f'
 where f' n (Fix u) (Fix l) = Fix $ (f . cNilStep) n u l
```

which leaves us with:

```
combine upr lwr = cfix (step1 . step2) upr lwr
 where step1 next upr lwr =
         combineStepDown next upr lwr
       step2 = combineStepUp
```

Now, we can drop the parameters and replace to `step1` and `step2` by `combineStepDown` and `combineStepDown`. Thus, the final version of `combine` reads

```
combine :: Layer high med emed ehigh ->
           Layer med low elow emed ->
           Layer high low elow ehigh
combine = cfix (combineStepDown . combineStepUp)
```

Similar to `liftStep`, `combineStepDown` and `combineStepUp` do not depend on the number of steps. Hence, they can be used to construct `combine` for layers with an arbitrary number of layer functions.

Simple editor

The `main` function for the simple editor is the same as in Section 6. Only the `editLoop` function has a couple of changes to account for the new constructors. To make the code more symmetric, we define deconstructor functions `unStep :: (Step a r :.: g) t -> a -> (r, (g t))` and `unNil` (which is the selector function of `NilStep`, when declared as a record.)

```
unStep (Comp (Step step)) = step
unNil (NilStep step) = step

editLoop (Fix presentStep) doc =
 do { let (pres , interpretStep) =
            unStep presentStep $ doc

    ; showRendering pres
    ; gesture <- getGesture

    ; let (update, presentStep') =
            unStep interpretStep $ gesture

    ; let doc' = updateDocument update doc
    ;
    ; editLoop (unNil presentStep') doc'
    }
```

7.4 Adding a monad

The final modification we make to the library is to add a monad, in order to allow layer functions to perform IO operations. The type `LayerFn` is extended with an extra type variable `m` for the monad.

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

Consequentially, the `Step` type is also modified to account for the monadic result:

```
newtype Step a b m ns = Step (a -> m (b, ns))
```

At composition, the monad is passed to both arguments:

```
newtype (:.:) f g m ns  = Comp (f m (g m ns))
```

The `NilStep` does not actually use the monad argument:

```
newtype NilStep m t = NilStep t
```

For the fixed point, we introduce a type synonym `FixM`, which passes the monad to its type-function argument, and applies `Fix` to the result.

```
type FixM m f = Fix (f m)
```

The monadic version of the other code is largely similar to the non-monadic version. Basically, each let expression of the form

`let` x_1 `=` exp_1`; ...;` x_n `=` exp_n `in` $(hRes, vRes)$

is replaced by a monadic statement

`do {` x_1 `<-` exp_1`; ...;` x_n `<-` exp_n`; return` $(hRes, vRes)$ `}`

Furthermore, the type signatures for the pairs of horizontal and vertical results $(hRes,\ vRes)$ become `m` $(hRes,\ vRes)$. Because of the similarity between the two libraries, we only show the monadic `liftStep`:

```
liftStep :: (hArg -> vArg-> m (vRes, hRes)) ->
            (hRes -> ns) -> hArg -> Step vArg vRes ns
liftStep f next horArgs = Step $
  \vArg -> do { (vertRes, horRes) <- f horArgs vArg
              ; return (vertRes, next horRes)
              }
```

The functions `lfix`, `lcomp`, `cfix`, and `ccomp` are independent of the monad and are the same for both versions of the library.

7.5 Final Library and conclusions

Figure 7 contains the final monadic library. In order to describe and implement an architecture, we need to provide a `Layer` type, and definitions of `lift` and `combine`. We give a general description of these definitions.

General use

The general case that we consider is a layer with n layer functions. The record type `TheLayer m` h_1 `...` h_m a_1 r_1 a_2 r_2 `...` a_n r_n contains the layer functions. The variable `m` is the monad, variables h_i are the types that appear in the horizontal parameters of the layer, and the a_i and r_i are the types of the vertical arguments and results.

Because the types of the horizontal parameters are not necessarily single h_i variables, but tuples of these variables, we denote the horizontal parameters by $horArgs_i$ and $horRess_i$. As an example, consider the type `Simple`. Its horizontal type variables are `map` and `state`, but the types of the horizontal parameters are `state` and `(map, state)`.

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

type LayerFn m horArgs vertArg horRess vertRes =
       horArgs -> vertArg -> m (vertRes, horRess)


newtype FixM m f = Fix (f m)

infixr :.:

newtype (:.:) f g m ns  = Comp (f m (g m ns))

newtype NilStep m t = NilStep t

newtype Step a b m ns = Step (a -> m (b, ns))

unStep (Comp (Step step)) = step
unNil (NilStep step) = step


lfix f = fix f' where f' n = Fix . (f . lNilStep) n

lNilStep next hRes = NilStep $ next hRes

liftStep f next horArgs = Comp . Step $
  \vArg -> do { (vertRes, horRes) <- f horArgs vArg
              ; return (vertRes, next horRes)
              }


cfix f = fix f'
 where f' n (Fix u) (Fix l) = Fix $ (f . cNilStep) n u l

cNilStep next (NilStep u) (NilStep l) =
  NilStep $ next u l

combineStepDown next (Comp (Step upper))
                     (Comp (Step lower)) = Comp . Step $
  \h -> do { (m ,upperf) <- upper h
           ; (l, lowerf) <- lower m
           ; return (l, next upperf lowerf)
           }

combineStepUp next (Comp (Step upper))
                   (Comp (Step lower)) = Comp . Step $
  \l -> do { (m, lowerf) <- lower l
           ; (h, upperf) <- upper m
           ; return (h, next upperf lowerf)
           }
```

Figure 7. Final meta-combinator library

In general, the definition of `TheLayer` has this form:

```
data
  TheLayer m h1 ... hm a1 r1 a2 r2 ... an rn =
    TheLayer { LayerFn1 :: LayerFn m horArgs1 a1
                                     horArgs2 r1
             , LayerFn2 :: LayerFn m horArgs2 a2
                                     horArgs3 r2 }
             ...
             , LayerFnn :: LayerFn m horArgsn an
                                     horArgs1 rn }
```

We assume the layer is normalized, meaning that $horArgs_1 = horRess_n$ and $horArgs_{i+1} = horRess_i$. If the layer is not normalized, a simple wrapper function can be defined to convert the layer to a normalized layer (see Section 3).

Type definitions

For a layer record as defined above, the type definition for the `Layer` type used by the combinators is:

```
type Layer m a1 r1 a2 r2 ... an rn =
  FixM m (Step a1 r1 :.: a2 r2 :.: ... :.: Step an rn)
```

Definition of `lift` and `combine`

The definitions of `lift` and `combine` are straightforward. For `lift`, we need to apply `liftStep` to each of the layer functions, compose the steps with `lcomp`, and apply `lfix` to the composition.

```
lift :: Monad m =>
        TheLayer m h1 ... hm a1 r1 ... an rn ->
        Layer m a1 r1 ... an rn
lift theLayer =
  lfix $ liftStep (LayerFn1 theLayer)
       . liftStep (LayerFn2 theLayer)
       ...
       . liftStep (LayerFnn theLayer)
```

The `combine` combinator consists of n `combineStepUp/Down` meta combinators, composed with `ccomp`, after which `cfix` is applied. The direction of the vertical data flow determines the choice between `combineStepUp` and `combineStepDown` for each step. The exact type of `combine` depends on the direction of the meta combinators and is explained below.

```
combine :: Monad m =>
           Layer m ... -> Layer m ... -> Layer m ...
combine =
  cfix $ combineStepUp/Down
       ...
       . combineStepUp/Down
```

The type of `combine` depends on the direction of the vertical data flow in the layer. Consider the i-th pair of type variables in `Layer a1 r1 ... an rn`. Variable `ai` represents the vertical argument of layer function i, and `ri` the vertical result. If step i is an upward step, the variables at this position in the `Layer` types are related as follows in the type signature for `combine`:

```
combine :: Monad m =>
           Layer ... md h ... -> Layer ... l md ... ->
           Layer ... l h ...
```

On the other hand, for a downward layer function, we have:

```
combine :: Monad m =>
           Layer ... h md ... -> Layer ... md l ... ->
           Layer ... h l ...
```

7.6 Conclusions

The meta combinator library has the advantages of the hidden-parameter solution from 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. For a specific architecture, we only need to define a `Layer` type, and give simple definitions of `lift` and `combine`.

8. Type-class magic

The definitions of `lift` and `combine` that have to be provided manually for each specific architecture are almost uniquely determined by the layer type, which leads to the question whether we can use

type classes to construct generic versions of these functions. Indeed, this turns out to be possible if we encode the direction of each step in its type. In this section, we present the type classes and instances that do the job. For clarity, we use the non-monadic combinators from Section 7 as a base.

To encode the direction of a step, we extend the `Step` type with a phantom-type variable (Leijen and Meijer 1999).

```
newtype Step dir a b ns = Step (a -> (b, ns))
```

Two constructorless types encode the direction.

```
data Up
data Down
```

Definition of `genericLift`

A generic version of `lift` would take a variable number of layer functions, and returns a layer type . The number of layer functions is determined by the number of steps in the result type (which is determined by the context in which it is used.)

The structure of a generic version of `lift` in a pseudo-Haskell language would read:

```
genericLift ::
  lf1 -> .. -> lfn ->
  Fix (Step Up/Down <s1> :.: .. :.: Step Up/Down <sn>)
genericLift = \lf1 .. lfn ->
              lfix (liftStep lf1 . ..  . liftStep lfn)
```

In this code, we can identify a composition function that takes a varying number of functions (`liftStep lf1` to `liftStep lfn`) and return a composition. If we assume a function `compose`, that takes a representation of the number of steps (denoted by `<n>`) followed by `n` functions, we can rewrite `genericLift` to:

```
genericLift = \lf1 .. lfn ->
  lfix (compose <n> (liftStep lf1) .. (liftStep lfn))
```

Now, we can identify another pattern `\a1 an -> f (g (h a1) .. (h an))`. Because of the parentheses around `g` and its arguments, we cannot simply compose `f` and `g`. We assume a function `app`, which takes the representation of the steps `<n>`, and the two functions `f` and `g`. For `h`, `app` uses `liftStep`, which is not a parameter. If we take `f` to be `lfix`, and `g` to be `compose <n>`, we have a new version of `genericLift`:

```
genericLift = app <n> lfix (compose <n>)
```

The final non-Haskell part in the definition is the `<n>` expression. The number of steps is represented by the argument of the `Fix` type in the result of `genericLift`. Hence, we assume a function `resType`, which for functions of type `a1 .. an -> Fix (s1 :.: .. :.: sn)` returns `steps t` (note that `steps` has kind `* -> *`). With this last function, the definition of `genericLift` is no longer pseudo code, but actual Haskell:

```
genericLift = app (resType genericLift) lfix
                  (compose (resType genericLift))
```

Which leaves us the task of defining the type classes for constructing the right instances for `compose`, `app`, and `resType`.

For the variable-argument composition `compose`, we declare a class `Comp` with a single method `comp`, which takes a composition type, and a neutral element, and returns a composition function that takes as many arguments as the composition type has steps. The function `compose` is simply `comp` with `id` for the neutral element.

```
class Comp (cmp :: * -> *) r c | cmp -> r c where
  comp :: cmp t -> r -> c
```

```
instance Comp (NilStep) (b->res) (b->res)  where
  comp cmp r = r

instance Comp g (a->res) cmp =>
         Comp (f :.: g) (y->res) ((a->y) -> cmp) where
  comp cmp r = \ab -> comp (rightType cmp) (r.ab)

rightType :: (f :.: g) t -> g t
rightType = undefined

compose c = comp c id
```

The application function is a bit more complex. The class `App` has a method `app`, which takes a composition type, and two functions `f` and `fx`. The result is a function that has the same number of arguments as the composition type has steps, and which applies `fx` to each argument, and applies these arguments to the function `f`.

```
class App (cmp :: * -> *) f fx r | cmp f -> fx r  where
  app :: cmp t -> f -> fx -> r

instance App (NilStep) (a->b) a b  where
  app cmp f a = f a

instance App g (a->b) d e =>
         App (Step dr ar rs :.: g) (a->b)
             (((hRes -> g ns) -> hArg ->
               (Step dr vArg vRes :.: g) ns) ->d)
            (LayerFn hArg vArg hRes vRes ->e) where
  app cmp f fx = \lf -> (app (rightType cmp) f
                             (fx (liftStep lf)))
```

The code for `liftStep` is the same as in the previous section. Its type differs slightly due to the direction type variable, but since this is a phantom type, it does not show up in the values.

The last problem we need to tackle is how to obtain the composition type over which `compose` and `app` recurse. For this we declare a class `ResType` with a method `resType`, which yields the result type of its function argument. Since the result of `genericLift` is always of type `Fix ct`, we can define a base instance for `Fix ct`, in which `restype` returns `ct t`, and a recursive instance for `a -> f`, in which `restype` returns the result type of `f`. Since no values are actually computed here, we can give the method a default implementation of `undefined`.

```
class ResType f res | f -> res where
  resType :: f -> res
  resType = undefined

instance ResType (Fix ct) (ct t)

instance ResType f r => ResType (a -> f) r
```

Definition of `genericCombine`

The situation for `genericCombine` is somewhat simpler than for `genericLift`, since the function does not have a varying number of arguments. However, unlike `genericLift`, the step functions are based on the direction of the step. If we look at `genericCombine`, the general structure would be:

```
combine ::
  Fix (Step Up/Down .. :.: .. :.: Step Up/Down .. ) ->
  Fix (Step Up/Down .. :.: .. :.: Step Up/Down .. ) ->
  Fix (Step Up/Down .. :.: .. :.: Step Up/Down .. )
combine = cfix \$ combineStepUp/Down
                  ...
                . combineStepUp/Down
```

We assume a function that creates a composition of `n` combine steps, where the choice for an upward or a downward step is based

on the direction of the respective `Step` type in the composition type.

```
genericCombine = cfix (combineC (resType genericCombine))
```

The type class looks a bit unfriendly due to the presence of the `Step` type, which is necessary because of the dependence on the direction.

```
class Combine (cmp :: * -> *) t f | cmp t -> f where
  combineC :: cmp t -> f

instance Combine NilStep t ((x -> y -> t) ->
          (NilStep x) -> (NilStep y) -> NilStep t) where
  combineC _ = \next (NilStep x) (NilStep y) ->
                NilStep (next x y)

instance (Combine c ct ( (ut -> lt -> ct) ->
                         u ut -> l lt-> c ct) ) =>
         Combine (Step Down a r :.: c) ct
                 ((ut -> lt -> ct) ->
                  (Step Down a m :.: u) ut ->
                  (Step Down m r :.: l) lt ->
                  (Step Down a r :.: c) ct) where
  combineC cmp = \next u l ->
    combineStepDown (combineC (rightType cmp) next) u l
```

There is also an instance for `Step Up` but it is very similar to the instance for `Step Down`. The only difference is that `Down` is replaced by `up`, and that the parameter order for the upper and lower arguments `a m` and `m r` is replaced by `m r` and `a m`. Hence, we do not show it here.

Simple editor

With the two generic functions defined above, it is no longer necessary to manually define a `combine` function. For `lift`, it still makes sense to define a function that takes a layer as an argument and selects the functions from this layer to pass on to `genericLift`.

The required definitions for the simple editor are:

```
type Layer dc prs gst upd =
  Fix (Step Down dc prs :.: Step Up gst upd :.: NilStep)


lift :: Simple state map doc pres gest upd ->
                state -> Layer doc pres gest upd
lift smpl = genericLift (present smpl) (interpret smpl)

main layer1 layer2 layer3 =
 do { (init1, init2, init3) <- initStates
    ; doc <- initDoc

    ; let layers = lift layer1 init1 `genericCombine`
                   lift layer2 init2 `genericCombine`
                   lift layer3 init3
    ; editLoop layers doc
    }
```

The code for `editLoop` is the same as in Section 7.

The monadic version of the generic combinators is a straightforward extension, but has even more daunting types, so we will not present it here.

Although type errors become somewhat more complicated because of the overloaded types, the type class solution turns out to be quite useful. The monadic version has been successfully tested in the Proxima generic editor.

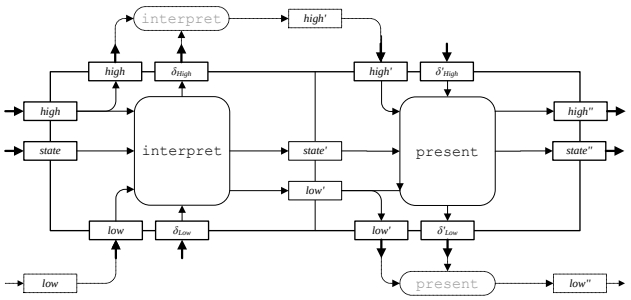

Figure 8. Data flow in a Proxima layer.

9. The Proxima editor

The Proxima editor uses the combinators from Figure 7 for the description of its architecture. The module that contains the architecture description is an part of the implementation of the prototype.

Although the precise data flow of a Proxima layer is beyond the scope of this paper, we show a general overview in Figure 8. The main difference with a `Simple` layer is that instead of mapping one data level onto another, a Proxima layer maps edit operations on a data level onto edit operations on the other level. Hence, each layer must also keep track of the actual data at each level. The edit operations are the deltas in the figure, whereas the data levels are the *high* and *low* values, which are threaded through the layer and switch between being vertical and horizontal parameters.

The data flow in the figure is encoded in the type definition for a Proxima layer. For the edit operations (the δ's in Figure 8), we use distinct types for edit operations going up (`editH` and `editL`) and edit operations going down (`editH'` and `editL'`). The reason for this distinction is that upward edit operations are often of a different nature than the downward ones. The code below is actual Proxima source code:

```
data Layer m state high low editH editL editH' editL' =
       Layer { interpret ::
                   LayerFn m (state, high) (low, editL)
                             (state, low) (high, editH)
             , present ::
                   LayerFn m (state, low) (high, editH')
                             (state, high) (low, editL')
             }
```

The `Layer` definition is part of the `Architecture` module, which imports the modules that define `interpret` and `present` for each layer. The definitions of the layer type as well as `lift` and `combine` for Proxima are the same as for the simple editor, except for the fact that `present` and `interpret` are swapped. The Proxima main edit loop is also virtually the same, except that it consists of more layers than the simple editor.

10. Conclusions and future work

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, clearly showing the data flow between the layers. The combinators have been heap profiled to ensure that no memory leaks are present, and have been used to implement the Proxima prototype as well as a database web-interface system.

Because the architecture description language is embedded in the implementation language, the architecture of a system forms part of the implementation of the system. We do not need to translate the architecture to an implementation, and hence, the imple-

mentation is guaranteed to comply with the architecture and vice versa.

According to Medvidovic and Taylor (Medvidovic and Taylor 2000), an architecture description language should describe the components of an architecture, the connectors, and the configuration. For the architecture combinators defined in this paper, we can identify these aspects as follows: the layer functions are the components; `lift` and `combine` are the connectors; and the applications of `lift` and `combine` determine the configuration.

In the paper we have assumed that once we call a (combined) step function, the information flows through all the layers. In a real editor this might not always be the case; if the user adds some extra white-space this might be recorded in the state at one of the intermediate levels, upon which the presentation process can be resumed. We can also envision a situation where information flows up and down a few times between two adjacent layers, until a situation is reached in which a change has to be propagated to yet another layer. Note that we actually only require that the upper protocol of a lower layer corresponds to the lower protocol of its upper layer; the updating of the document and providing the gesture at the bottom layer are examples of simple single-step protocols. We foresee however that we might use the type system to describe much more complicated protocols. An open question, and a matter of debate, is whether this should be done by exploiting the Haskell type system further, or whether one should move to more expressive type systems such as found in Agda (Norell 2007). Only experimenting and comparing solutions can give a definitive answer. Another area of research concerns how dynamic aspects of the architecture, such as invariants and constraints on the data, can be described and, if possible, verified.

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. For us the experiment to model the structure of the system using Haskell has been a successful experience, and we hope that this paper will inspire others to pursue the approach for different classes of architectures.

References

Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, 1999.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. ISSN 0098-5589.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, Oct 2004. URL `http://www.cs.uu.nl/research/projects/proxima`.