Haskell as an ADL for layered architectures

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

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.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages (e.g., description, interconnection, definition)

General Terms Domain-specific languages, Architecture

Keywords keyword1, keyword2

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, the method is applicable to layered architectures in general. The combinators have been successfully used for the implementation of the generic editor Proxima (Schrage 2004), as well as for a system that provides web-interfaces to databases.

Describing an 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 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 (ADLs) (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 ADL's from, amongst others, programming languages. However, Higher-order typed (HOT) functional programming languages, such as Haskell, Clean (Plasmeijer and van Eekelen 2001), and ML (Milner et al. 1997) 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.

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. 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.

Often, a program that makes use of a DSEL, contains many applications of the combinators, and hence the evaluation of the combinators forms a substantial part of the running time. Consider, for example, parsers or pretty printers. In contrast, the combinators in an architecture description are just 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 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.

2. A simple editor

We introduce a simple layered architecture for a presentation-oriented editor, which is used in the next sections to explain the architecture description methods. 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 components: $\texttt{present}_1 \ldots \texttt{present}_n$. 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 then mapped onto a document update (`upd`) by $\texttt{interpret}_n \ldots \texttt{interpret}_1$, after which it is performed on the document.

A layer consists of a pair of $\texttt{present}_i$ and $\texttt{interpret}_i$ functions, which we refer to as *layer functions*. Besides the vertical data flow for presentation and interpretation, we may also have horizon-

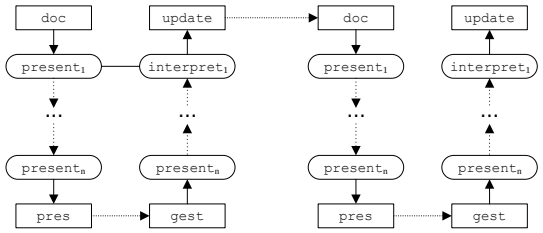

Figure 1. Two cycles in the editing process.

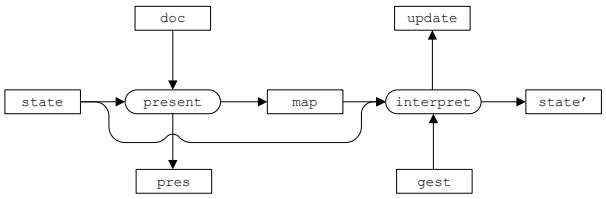

Figure 2. A single layer.

tal data flow that stays within the layer. Horizontal data flow concerns results from a layer function that are passed to a layer function it the same layer, possibly in a next edit step.

Figure 2 shows the data flow for a single layer with two examples of horizontal data flow. The `map` result of `present` is passed to `interpret` and represents information about the presentation mapping. 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 following sections, we explore the possibilities of implementing the layered architecture from the previous section in 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. Thus, we eliminate the extra step of translating the architecture description to a program, which is a possibe source for 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. In the next sections, we present three methods for modeling the connections, but first we focus on the layer functions.

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)
```

We model a layer with a record that contains the layer functions. For the example editor, we have the two layer functions: `present` and `interpret`. The types of the layer functions follow directly

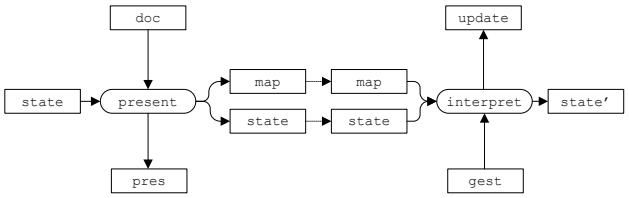

Figure 3. Data flow in a normalized layer.

from Figure 2. If we would put these functions directly in a record, we would get:

```
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
              }
```

4. Method 1: Explicitly connecting the components

Now that the layers have been modeled, we need to realize the 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 the selection and application of each of the functions in each of the layers.

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. At the bottom of the figure, the presentation is shown to the user, and an edit gesture is obtained, which we represent 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 corresponding Haskell code for the edit loop is:

```
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')) =
```

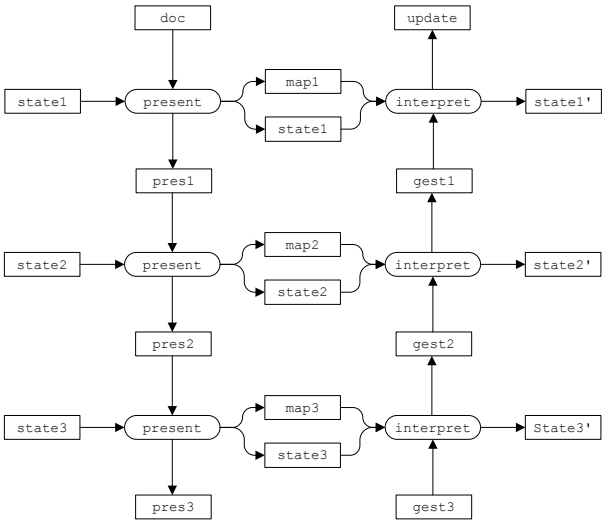

Figure 4. Data flow in and between layers.

```
        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 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, 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.

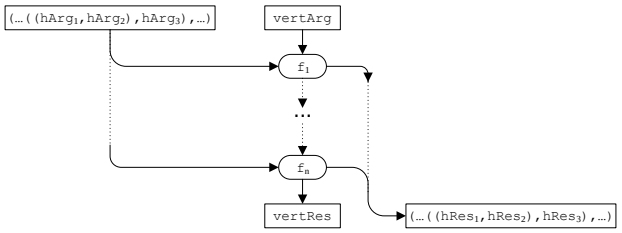

Figure 5. Horizontal parameters in nested pairs.

5. Method 2: Nested pairs

In this section we abstract from the horizontal and vertical data-flow 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 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 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 `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 right number of elements. A tuple or castesian 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 consisting of only pairs to represent the horizontal parameters and results.

Although it will not be enforced by the combinators, we only use left-associatively nested products in this section: `(...((`e_1`,`e_2`),`e_3`),` `...,` e_n`)`. However, 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 exact structure does not matter.

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

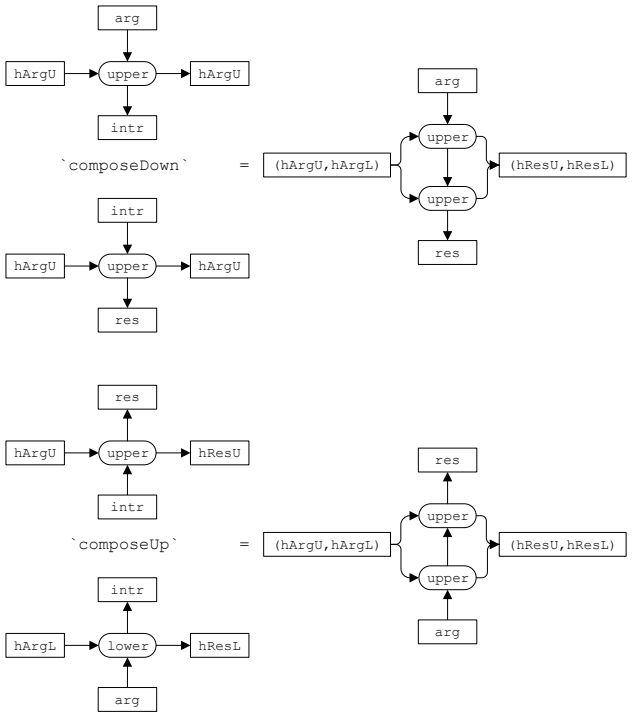

Figure 6. `composeDown` and `composeUp`

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 `composeUp`. 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 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 the 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.

The type `LayerC` is 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 much 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 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.

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

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. 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. Thus, the main editor loop becomes more transparent and the type of a combined layer is simpler since the type variables for the horizontal parameters are hidden.

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. 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 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, `state`, 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 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.

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 { (state1, state2, state3) <- initStates
   ; doc <- initDoc
   ; let layers = lift layer1 state1 `combine`
                  lift layer2 state2 `combine`
                  lift layer3 state3
   ; 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 with 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 steps. Even though the combinators are straightforward, 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 desirable to have a small library of meta combinators for constucting these combinators. Another advantage of a meta-combinator library is that instead of explicitly encoding the direction for each of the steps in the `combine` function, we can use a meta combinator whose name reflects the direction (similar to 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`.

The method we use for the derivation of the meta combinators is to start with the combinators from the previous section and gradually factorize out all step-specific aspects. In the end, we have a combinator that is constructed by using straightforward applications 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 requires a user to provide an instance of this class, as well as complicate the types and introduce problems with the monomorphism restriction. Therefore, we introduce a compositional representation of the layer types that only makes use of 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 can capture such a step with 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` that is the base case for a composition:

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

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

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

With these types, we can give a compositional point-free definition for `Layer`:

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

Because `Step`, `NilStep`, and `:.:` appear partially parameterized in the layer type, and `Fix` is recursive, all three types need to be newtypes. Hence, instead of one `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 compositions (`:.:`) to be the same as the number of steps, which will facilitate the removal of `Comp` constructors.

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. We eliminate the mutual recursion 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
```

Now, we can 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 aComp 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)
```

The new lift reads

```
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)
  ...
```

After which we can capture the recursion pattern with the fix combinator:

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

Thus, we get:

```
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 to 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 layers with arbitrary numbers of steps

For an n-step layer, the definition of lift (using the method from Section 6) has n local step functions, each one 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 have:

```
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, and adapt 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 symmetrical, 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 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, so here it is a phantom type (**?**).

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

For the fixed point, the monad is passed to its argument and to the recursive call.

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

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. *

TODO: explain this more?

7.5 Type-class magic

Defs lift and combine are simple and depend completely on type. It turns out that we can define them automatically with a class
type Step changes

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

data Up
data Down

generic lift = \a1 .. an -> lfix $ liftStep a1   . liftStep an
generic lift = \a1 .. an -> lfix $ compose  a1 .. an
```

First, we construct a class `Comp` that provides us with a variable argument composition, by recursion over the composition type:

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

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

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

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

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)))
```

A final problem is how to obtain the composition type. It is the result

The solution is a type class that gives the type of the result of a function. (result must be fix)

```
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
```

genericLift is defined by putting together all the components.

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

Combine is simpeler, also a composition, but no args, so no app needed.

```
combine = cfix \$ combineStepUp/Down . ... . combineStepUp/Down

-- combine
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

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

TODO: maybe not show Up instance?

```
*

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

Now everything is in the library! Just give a type and a fix and it works. main looks like this now:

```
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)
```

```
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 Fix m f = Fix (f m (Fix m f))

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

```
main layer1 layer2 layer3 =
 do { (state1, state2, state3) <- initStates
    ; doc <- initDoc

    ; let layers = lift layer1 state1 `genericCombine`
                   lift layer2 state2 `genericCombine`
                   lift layer3 state3
    ; editLoop layers doc
    }
```

The monadic version of the type classes is straightforward but even nastier to look at, so we will not present it here.

7.6 Final Library and conclusions

Figure 7 contains the final monadic library. 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 a layer with n layer functions. The record type `TheLayer` $\mathtt{h}_1$ ... $\mathtt{h}_m$ $\mathtt{a}_1$ $\mathtt{r}_1$ $\mathtt{a}_2$ $\mathtt{r}_2$ ... $\mathtt{a}_n$ $\mathtt{r}_n$ contains the layer functions. 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 results.

Because the types of the horizontal pararameters are not necessarily single $\mathtt{h}_i$ 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)`.

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

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

We assume the layer is normalized, which means 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

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

Definition of `lift` and `combine` *

TODO: account for monad

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 :: TheLayer h1 ... hm a1 r1 ... an rn ->
        Layer 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 :: Layer ... -> Layer ... -> Layer ...
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` $\mathtt{a}_1$ $\mathtt{r}_1$ ... $\mathtt{a}_n$ $\mathtt{r}_n$. Variable $\mathtt{a}_i$ represents the vertical argument of layer function i, and $\mathtt{r}_i$ 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 :: Layer ... m h ... -> Layer ... l m ... ->
           Layer ... l h ...
```

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

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

*

TODO: show final editor code again? (only very small difference because of monad)

7.7 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`.

The fact that we still need to write some code by hand leads to the question whether it is possible to define a type class that allows us to `lift` and `combine` as its member functions. With such a type class, and a layer type that contains an encoding of the direction of the vertical data flow, we would no longer need to explicitly define the combinators.

For `lift`, this type class can be defined, although it does require the explicit base case `TNil` for type composition (see Section **??**). For `combine`, the situation is a lot more complicated due to the different data-flow directions in each of the steps. If we encode the data flow in the `Step` type (i.e. by using types `StepUp` and `StepDown`), it may be possible to define a type class for `combine`. Whether this is the case remains further research. * However, it is questionable whether this is actually worth the effort. The code for the meta combinators would turn much more complex, whereas the gains of a single `lift` and `combine` function are not that much, given the fact that they only need to be defined once and will typically remain constant.

TODO: Het lijkt er alleen op dat ik die klasse bijna gevonden heb, wat het hele paper in de war zou gooien..

8. Related work

Why this is nicer than an architecture description language.

9. Conclusions

The combinators presented in this paper make it possible to specify layered editor architectures in a concise and transparent way, making the data flow within the layer. 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 implementation is guaranteed to comply with the architecture.

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.

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 how dynamic aspects of the architecture, such as invariants and constraints on the data, can be described and, if possible, verified.

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. URL `citeseer.ist.psu.edu/hudak98modular.html`.

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. doi: http://dx.doi.org/10.1109/32.825767.

Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0262631814.

R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. URL `http://www.cs.kun.nl/ clean/contents/contents.html`.

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/thesis.pdf`.