Compositional Model-Views with Generic Graphical User Interfaces

Peter Achten, Marko van Eekelen, and Rinus Plasmeijer

Department of Software Technology, University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
Telephone: +31 24 365 2644, Fax: +31 24 365 2525,
peter88@cs.kun.nl, marko@cs.kun.nl, rinus@cs.kun.nl

Abstract. Creating GUI programs is hard even for prototyping purposes. Using the model-view paradigm makes it somewhat simpler since the model-view paradigm dictates that the model contains no GUI programming, as this is done by the views. Still, a lot of GUI programming is needed to implement the views.
We present a new method for constructing GUI applications that fits well in the model-view paradigm. Novel in our approach is that the views also contain no actual GUI programming. Instead, views are constructed in a fully compositional way by defining a model of the view. We use a technique developed earlier to generate the GUI part. We show how the method supports flexibility, compositionality and incremental change by introducing abstract components in the view models.

Keywords: Graphical User Interfaces, Generic and Functional Programming.

1 Introduction

The design of high quality user interfaces is an iterative process that has a great demand for rapid prototyping and flexible incremental change of versions of the UI under construction [14]. In practice, writing effective Graphical User Interfaces (GUI) with programming toolkits for even small programs (500 lines of code) is a complicated task. This is caused by two major obstacles:

A. The programmer needs to be skilled in the API of the used library and the tools that help him in his task (such as resource editors).
B. GUI programs tend to tie up the logic of the application with the realization of its user interface.

In this paper we show how contemporary functional language techniques using generic programming and strong type system features (existential types and rank-2 polymorphism) can be employed to obtain a programming toolkit that eliminates these obstacles. Even though the used techniques are advanced, the resulting API of this toolkit is concise, and the method-of-use is not hard, as will be demonstrated in this paper.

The system described in this paper fits well in the well-known *model-view* paradigm [12], introduced by Trygve Reenskaug in the language Smalltalk (the paradigm was then named the *model-view-controller* paradigm). In our approach the data type plays the *model* role, and the *views* are derived automatically from the generic decomposition of values of that type. The *controller* role is dealt with by both the automatically derived communication infrastructure and the views (as they need to handle user actions).

In our method we will eliminate obstacle **A** by using *Graphical Editor Components* [3]. A $GEC_{\mathtt{m}}$ is an interactive editor (the *view*) to edit values of arbitrary data type `m` (the *model*) in a type-safe way. Using generic programming techniques, the view is automatically derived from the type `m` of the model. Hence, the programmer does not need to know about GUIs. One might gather that this is also sufficient to eliminate obstacle **B**, but this is not the case. The obstacle is still present in two ways:

B.1. The type of the model not only represents the data that is used by the application logic, but at the same time it represents the information that is needed to automatically generate the intended view. In this sense, the view is not well separated from the model.

B.2. A different editor can only be specified by defining a different type. Consequently, changing views incrementally implies changing types which in its turn implies further changes reducing flexibility.

In this paper, **B.1** is dealt with by imposing a strict separation of concerns of the model. Instead of one model, the programmer defines a *data model and* a *view model* and their relation in the form of two conversion functions. Then, the *GEC* system can be used to derive the intended GUI from the view model. **B.2** is dealt with by introducing *abstract* views (*AGEC*s) that can be used as components of the view model. Due to the power of abstraction *AGEC*s are fully compositional.

The resulting system encourages an incremental methodology of programming GUIs. For rapid prototyping purposes, one starts with identical types for the data model and view model and the trivial *identity* transformation functions. Then, one can start to change views incrementally by changing instances of abstract component views in the view model.

The language that we have used is Clean, but it should be noted that the approach is applicable to other functional languages (with other I/O libraries) that support the above mentioned features as well, for instance Generic Haskell [9]. In principle, this can be done with any I/O library but using the Haskell Object I/O library [2, 5] will minimize the effort of porting the system.

This paper is organized as follows. In Sect. 2 we recapitulate the concept of a *GEC*. Using these *GEC*s as basic building blocks, we show in Sect. 3 how we eliminate obstacles **B.1** and **B.2**, giving us the intended system. We present related work in Sect. 4 and conclude and point to future work in Sect. 5.

2 The Concept of a Graphical Editor Component

In [3] we introduced the concept of a Graphical Editor Component, a $GEC_{\mathtt{t}}$. A $GEC_{\mathtt{t}}$ is an editor for values of type `t`. It is provided with an initial value of type `t` and it is guaranteed that an application user can only use the editor to create values of type `t`. At all times, a $GEC_{\mathtt{t}}$ contains a value of type `t`.

A $GEC_{\mathtt{t}}$ is generated with a *generic* function [10, 4]. A generic function is a meta description on the structure of types. For any concrete type `t`, the compiler is able to automatically derive an instance function of this meta description for the given type.

Currently, we support all Clean types, with exception of function and abstract types. The power of a generic scheme is that we obtain an editor for free for any data type. This makes the approach particularly suited for *rapid prototyping*.

Before explaining *GEC*s in more detail, we need to point out that Clean uses an explicit environment passing style [1] for I/O programming. This style is supported by the uniqueness type system [6] of Clean. Because *GEC*s are integrated with Clean Object I/O, the I/O functions that are presented in this paper are state transition functions on the program state (`PSt st`). The program state represents the external world of an interactive program, tailored for GUI operations. In this paper the identifier `env` is a value of this type. In the Haskell variant of Object I/O, a state monad is used instead. The uniqueness type system of Clean ensures single threaded use of the environment. Uniqueness type attributes that actually appear in the type signatures are not shown in this paper, in order to simplify the presentation.

2.1 Creating $GEC_{\mathtt{t}}$s

A $GEC_{\mathtt{t}}$ is a graphical editor component to edit values of type `t`. These editors are *created* with the generic function `gGEC`. This function takes a *definition* (`GECDef t env`) of a $GEC_{\mathtt{t}}$ and *creates* the $GEC_{\mathtt{t}}$ object in the environment. It returns an *interface* (`GECInterface t env`) to that $GEC_{\mathtt{t}}$ object. It is a `(PSt ps)` transition function because `gGEC` modifies the environment.

```
generic gGEC t :: GECFunction t (PSt ps)

:: GECFunction t env :== (GECDef t env) → env → (GECInterface t env,env)
```

A $GEC_{\mathtt{t}}$ is defined by `GECDef t env` which consists of two elements. The first is a value of type `t` which will be the initial value of the editor. The second is a call-back function of type `t → env → env`. The editor must know what parts of the program are interested in changes of the current value that are done by the user. This information is provided by its 'context' in the form of this call-back function. The editor uses this function when the user has changed the current value of the editor.

```
:: GECDef           t env :== (t, CallBackFunction t env)
:: CallBackFunction t env :== t → env → env
```

The `GECInterface t env` is a record that contains all *methods* that the 'context' can use to handle the newly created $GEC_{\mathtt{t}}$.

```
:: GECInterface t env =  { gecGetValue :: env → (t,env)
                         , gecSetValue :: t → env → env }
```

The method `gecGetValue` can be used to obtain the currently stored value of type `t` from the $GEC_{\mathtt{t}}$ component. The method `gecSetValue` can be used to set a new value in the corresponding $GEC_{\mathtt{t}}$. `GECInterface` contains several other useful methods for a program that are not shown above. These are methods to open and close the created $GEC_{\mathtt{t}}$ and to show or hide its appearance.

The appearance of a standard $GEC_{\mathtt{t}}$ is illustrated by the following example. Assume that the programmer has defined the type `Tree a` as shown below and consider the following application of `gGEC`:

```
::  Tree a = Node (Tree a) a (Tree a) | Leaf

gGEC (Node Leaf 1 Leaf, const id) env
```

This creates a $GEC_{\texttt{Tree Int}}$ which displays the indicated initial value (see Fig. 1). The application user can manipulate this value in any desired order thus producing new values of type `Tree Int`. Each time a new value is created, the call-back function is applied automatically. The call-back function of this first example (`const id`) has no effect. The shape and lay-out of the tree being displayed adjusts itself automatically. Default values are generated by the editor when needed.

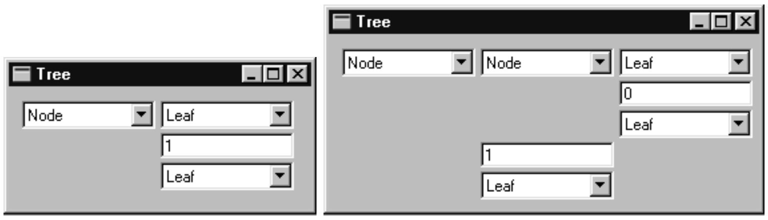


Fig. 1. The initial Graphical Editor Component for a tree of integers (Left) and a changed one (Right: with the pull-down menu the upper `Leaf` is changed into a `Node`).

2.2 Self-adjusting Graphical Editor Components

In [3] a number of examples are given to show how graphical editor components can be combined relying on the call-back mechanism and method invocation. In this paper we only use one particular form of combination, namely that of an editor that *itself* reacts to edit operations by the user. In this way, an editor can be *self-correcting*: any property on edit values of type `a` that is expressable by means of a function `f :: a → a` can be considered to be an invariant on the editor. As an example, we can construct a *sorted-list* editor by applying the `sort` function to all edited values.

Self-adjusting editors can be created with the concise function `selfGEC`:

```
selfGEC :: (a → a) a (PSt ps) → (PSt ps) | gGEC{|a|}
selfGEC f va env = new_env
where
    (thisGEC,new_env) = gGEC (f va, λnva. thisGEC.gecSetValue (f nva)) env
```

The function `selfGEC`, when applied to a function `f :: a → a` and value `va :: a`, creates a GEC_{a} with initial value (`f va`). The call-back function of this GEC_{a} is quite remarkable. At each change of value the editor re-applies `f` to the new value of type `a` and sets it as the actual new value of *itself*. Notice that, due to the explicit environment passing style, it is trivial in Clean to connect GEC_{a} to itself. In Haskell's monadic I/O one needs to tie the knot with `fixIO`.

2.3 Customizing Graphical Editor Components

The generic definition of `gGEC` enables the system to derive a GEC_{t} for arbitrary values of type `t`. Occasionally one needs to deviate from the standard GEC_{t} because it does not suit the requirements of the particular application. In [3] we show that this can be done by defining special instances for the types that need to be customized. This has been demonstrated for the ubiquitous *counter* example. In Fig. 2 the self-correcting code (`updCntr`) and model type (`Counter`) is given. The default $GEC_{\texttt{Counter}}$ (shown at the bottom in Fig. 2) is a mirror image of the generic representation of the `Counter` model. It works as intended, and we get it for free. Unfortunately, its view is a counterexample of a good-looking counter.

```
updCntr :: Counter → Counter
updCntr (n,Up)   = (n+1,Neutral)
updCntr (n,Down) = (n-1,Neutral)
updCntr any      = any

:: Counter :== (Int,UpDown)
:: UpDown   =  Up | Down | Neutral
```

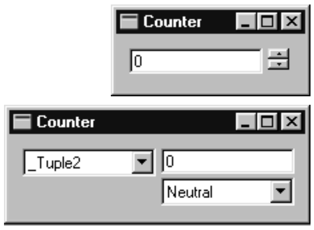


Fig. 2. Two $GEC_{\texttt{Counter}}$s created by `selfGEC updCntr (0,Neutral)`. The standard one (bottom) and a customized one (top).

The changes that are required to obtain the customized editor are to define new generic instances for `(,)` (hide the constructor and place its arguments next to each other) and `UpDown` (display  instead of ).

Although in a slightly artificial way, this example demonstrates the obstacles **B.1** and **B.2** that we intend to remove. The increment/decrement behaviour that is captured with the `UpDown` type also fixes the derived GUI. The only ways to change the GUI are to use another type for this behaviour or to customize the editor for that type, as shown above.

3 Compositional Graphical Editor Components

Using the generic `gGEC` function, we automatically get an editor for any data type we invent. This is great for rapid prototyping. However, the appearance of the editor that we get for free in this way, might not resemble what we have in mind. We have explained in Sect. 2.3 that an editor can be customized for a specific type by defining a specialized instantiation of the generic function `gGEC` for that type. For certain basic types e.g. representing buttons and the like, this is exactly what we want. We also want to be able to create new editors from existing ones in a compositional way. Editors are automatically generated given a concrete type and a value of that type. The only way we can change this is by defining specialized editors for certain types. We need to invent a new way to realize abstraction and composition based on this specialization mechanism. In the following sections we show step by step how this can be accomplished.

First, we show in Sect. 3.1 how a program can be split such that a clear separation can be made between editor dependent and editor independent code. This makes it possible to choose any editor just by making changes to the editor dependent code, while we never have to make any changes to the editor independent code. Next, in Sect. 3.2 we show how to construct self-contained editors that take care of their own update and conversion behaviour. Finally, in Sect. 3.3 we turn these self-contained editors into *abstract reusable* editors, thus encapsulating all information about their implementation and behaviour. However, *abstract* types seem to be at odds with generic programming. We show in Sect. 3.3 how we have managed to solve this problem.

Although the solution requires high-level functional and generic language constructs, it should be emphasized that editors remain very straightforward to use. In this section we construct a running example to illustrate the technique. The code fragment **(1)** below shows the *data model*. Code fragments appear as framed pieces of `code`. The data model is a record of type `MyDataModel`. The intention is that whenever the user edits one of the fields `value1` or `value2`, then these new values are summed and displayed in field `sum`. This behaviour is defined by `updDataModel`. We want to emphasize that the types and code are 'carved in stone': they do not change in the rest of this paper.

```
:: MyDataModel
     = {value1 :: Int, value2 :: Int , sum :: Int}

initDataModel (v1,v2)
     = {value1 = v1, value2 = v2, sum = v1 + v2}

updDataModel :: MyDataModel → MyDataModel
updDataModel rec = { rec & sum = rec.value1 + rec.value2 }
```
(1)

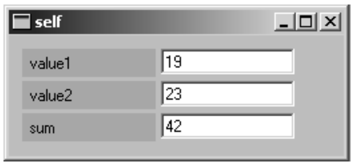
The 'functional record update' notation `{`r `&` f_0 `=` v_0 `, . . . ,` f_n `=` v_n`}` creates a new record value in which all fields have the same value as in r except the updated fields $f_0 \ldots f_n$.

3.1 Separation of Concerns by Separating Types

First of all, we want to accomplish a good separation of concerns. Ideally, it should be possible to concentrate on the functionality of the program without worrying about the actual shape of the editors. If one is not happy with the standard editor, it should be possible to construct the appropriate editor later without being forced to modify code that is not shape-related.

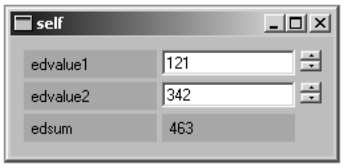
Using the function `selfGEC` we can immediately get a $GEC_{\texttt{MyDataModel}}$ for free for testing purposes. Below we show what the $GEC_{\texttt{MyDataModel}}$ GUI looks like when created by the function `standardEditor`. Each time the application user changes a value with the editor, the function `updDataModel` is applied and a new sum is calculated and displayed.

```
standardEditor
  = selfGEC updDataModel
            (initDataModel (0,0))
```



Now suppose that we do not like the look of this standard editor very much, and want a different one. This is `myEditor` shown below in code fragment **(2)**. Again, this code is 'carved in stone'. We want to reuse the counters of the previous section for editing the two value fields. As the sum is calculated given these two values, we do not want the sum value to be editable at all.

```
myEditor to from
  = selfGEC (to o updDataModel o from)
            (to (initDataModel (0,0)))
```
(2)



Since editors are created fully automatically just by looking at the type, the only way to obtain the desired editor is by using suitable data types. For the counters we use the `Counter` type. We assume that we have a specialized basic editor for the type `Display a`: this editor shows any value of type `a`, but the value cannot be changed in the editor. We combine these types in a new type to obtain the desired editor.

As we said before, we do not want to change the code fragments **(1)** and **(2)**. For this reason we make a clear distinction between the *model* of the *data* (`MyDataModel`) and the *model* of the *view* used to generate the editor we want (`MyViewModel`). Conversion functions between these two models need to be defined (`toMyViewModel` and `fromMy-ViewModel`).

This strict separation of concerns removes obstacle **B.1**: the data model has nothing to do with the means of visualization; this is done by the view model.

We can now easily express in the function `myEditor` how the *view* and *model* are connected. To glue them together we just need two conversion functions `to` and `from` the editor domain. We convert the initial value `initDataModel` to the view domain and create an editor. After a change being made with the editor we convert the new values back to the data model domain, apply the algorithm `updDataModel`, and convert the result back to the view domain such that it can be displayed and edited again.

Consequently, we obtain a running editor by applying:

```
myEditor toMyViewModel fromMyViewModel
```
(3)

The editor we get is completely determined by the type of `MyViewModel` and the definition of the conversion functions. If we want another editor, we only have to change this type and/or these conversion functions, all other code remains the same.

```
:: MyViewModel = { edvalue1 :: Counter          // an updown counter
                 , edvalue2 :: Counter          // an updown counter
                 , edsum    :: Display Int }    // non-editable integer value

toMyViewModel :: MyDataModel → MyViewModel
toMyViewModel rec     = { edvalue1 = toCounter rec.value1
                        , edvalue2 = toCounter rec.value2
                        , edsum    = toDisplay rec.sum }

fromMyViewModel :: MyViewModel → MyDataModel
fromMyViewModel edrec = { value1   = fromCounter (edrec.edvalue1)
                        , value2   = fromCounter (edrec.edvalue2)
                        , sum      = fromDisplay edrec.edsum }

toCounter    n    = (n,Neutral)
fromCounter (n,_) = n

:: Display a = Display a
```

```
toDisplay   x           = Display x
fromDisplay (Display x) = x
```

In this way we have created a separate layer on top of the *unchanged* existing program. Unfortunately, we did not really reach the desired compositional behaviour. By choosing another data type one does obtain another editor for free that *looks* the way we want, but one does not automatically get the desired self-contained *behaviour* with it. For instance, we have used the type `Counter` in the definition of `MyViewModel`. The generated editor displays a counter, but it does not take care of the updates of the counter. This is clearly not what we want. We have to invent a type from which self-contained editors can be generated.

3.2 Defining Self-Contained Editors

If we want to reuse an existing editor, it is not enough to reuse its type. We also want to reuse its functionality: each editor should take care of its own update. For this purpose we need a type in which we can store the functionality of an editor. If want to create a view `v` on a domain model `d`, we need to be able to replace a standard editor for type `d` by a self-contained editor for some isomorphic type `v`. Furthermore, since we generally also have to perform conversions between these types, we like to store them as well, such that each editor can take care of its own conversions. Finally, it is generally useful to take into account the old value of `v` when converting from `d` since editors may have an internal state.

Therefore we define a new type, `ViewGEC d v`, in which we can store the update and conversion functions, and we define a specialized version of our generic editor `gGEC` for this type (`gGEC{|ViewGEC|}`). The definitions are given below. Notice that in `gGEC{|ViewGEC|}` two additional parameters appear: `gGECd` and `gGECv`. This is caused by the fact that generic functions in Clean are kind-indexed functions. As `ViewGEC d v` is of kind $\star \to \star \to \star$, the generic function has two additional parameters, one for type `d` and one for type `v`.

The `ViewGEC` editor does the following. The value of type `d` is stored in the `ViewGEC` record, but a `d`-editor (`gGECd`) for it is not created. Taking the old value of `v` into account, the `d`-value is converted to a `v`-value using the conversion function `d_oldv_to_v :: d → (Maybe v) → v`. For this `v`-value we do generate a generic `v`-editor (`gGECv`) to store and edit the `v`-value.

Whenever the application user creates a new `v`-value with this editor, the call-back function of the `v`-editor is called (`viewCallback`) and the `update_v :: v → v` function is applied. This is similar to applying `selfGEC update_v` to the corresponding new value of type `v`. The resulting new `v`-value is shown in the `v`-editor again, and it is converted back to an `d`-value as well, using the function `v_to_d :: v → d`. This new `d`-value is then stored in the `ViewGEC` record in the `d_val` field, and the call-back function for the `ViewGEC` editor is called (`viewGECCallback`). The new `d`-value can be inspected in the program as if a new `d`-value was created with a standard generic `d`-editor.

```
:: ViewGEC d v = { d_val        :: d
                 , d_oldv_to_v  :: d → (Maybe v) → v
                 , update_v     :: v → v
                 , v_to_d       :: v → d }
```

```
mkViewGEC :: d (d → v) (v → v) (v → d) → ViewGEC d v
mkViewGEC d fdv fvv fvd = { d_val         = d
                          , d_oldv_to_v   = fdvv
                          , update_v      = fvv
                          , v_to_d        = fvd }
where
    fdvv d Nothing  = fdv d
    fdvv _ (Just v) = v

gGEC{|ViewGEC|} gGECd gGECv (viewGEC, viewGECCallback) env
    = ({ gecSetValue = viewSetValue vInterface
       , gecGetValue = viewGetValue vInterface },new_env)
where
    (vInterface,new_env) = gGECv (viewGEC.d_oldv_to_v viewGEC.d_val Nothing
                                 ,viewCallback vInterface
                                 ) env

    viewCallback vInterface new_v env
        = viewGECCallback {viewGEC & d_val = new_d} new_env
    where
        new_upd_v = viewGEC.update_v new_v
        new_env   = vInterface.gecSetValue new_upd_v env
        new_d     = viewGEC.v_to_d new_upd_v

    viewSetValue vInterface new_viewGEC env
       = vInterface.gecSetValue new_v new_env
    where
        newb            = new_viewGEC.d_oldv_to_v new_viewGEC.d_val (Just old_v)
        (old_v,new_env) = vInterface.gecGetValue env

    viewGetValue vInterface env
        = ({viewGEC & d_val = viewGEC.v_to_d current_v},new_env)
    where
        (current_v,new_env) = vInterface.gecGetValue env
```

The concrete behaviour of the generated `ViewGEC` editor now not only depends on the type, but also on the concrete information stored in a value of type `ViewGEC`. Now it becomes very easy to define self-contained reusable editors, such as a counter editor, shown below. The corresponding editor takes care of the conversions and the update. The `displayGEC` does a trivial update (`id`entity) and also takes care of the required conversions.

```
counterGEC :: Int → ViewGEC Int Counter
counterGEC i = mkViewGEC i toCounter updCntr fromCounter

displayGEC :: a → ViewGEC a (Display a)
displayGEC x = mkViewGEC x toDisplay id fromDisplay
```

Making use of these new self-contained editors we can repair and even simplify our previous editor definition. To replace it, we only have to provide a new definition of `MyViewModel` and of the conversion functions `toMyViewModel` and `fromMyViewModel`. All other definitions remain the same.

```
:: MyViewModel  = { edvalue1 :: ViewGEC Int Counter
                  , edvalue2 :: ViewGEC Int Counter
                  , edsum    :: ViewGEC Int (Display Int) }

toMyViewModel :: MyDataModel → MyViewModel
toMyViewModel rec = { edvalue1 = counterGEC rec.value1
                    , edvalue2 = counterGEC rec.value2
                    , edsum    = displayGEC rec.sum }

fromMyViewModel :: MyViewModel → MyDataModel
fromMyViewModel edrec = { value1 = edrec.edvalue1.d_val
                        , value2 = edrec.edvalue2.d_val
                        , sum    = edrec.edsum.d_val }
```

In the definition of `toMyViewModel` we can now simply choose any suited self-contained editor. Each editor handles the needed conversions and updates itself automatically. To obtain the value we are interested in, we just have to address the `d_val` field.

The example shows that we have obtained the compositional behaviour that we wanted to have. One problem remains. If we would replace a self-contained editor by another in `toMyViewModel`, all other code remains the same. However, we do have to change the type of `MyViewModel`. In this type it is completely visible what kind of editor has been used. The abstraction would be complete if we also manage to create an abstract data type for our self-contained editors.

3.3 Abstract Self-Contained Editors

The concrete value of type `ViewGEC d v` is used by the generic mechanism to generate the desired self-contained editors. The `ViewGEC d v` type depends on the type of the editor `v` that is being used. Put in other words, the type still reveals information about the implementation of editor `v`. This is undesirable for two reasons: one can not exchange views without changing types, and the type of composite views reflects their composite structure. For these reasons, we want a type that *abstracts* from the concrete editor type `v`.

However, if we manage to hide these types, how can the generic mechanism generate the editor for it? The compiler can only generate an editor for a given concrete type, not for an abstract type of which the content is unknown. The solution is as follows. When the abstraction is being made, we *do* know the contents and its type. Hence, we can store the *generic editor function* (of type `GECFunction`, see Sect.2.1) in the abstract data structure itself where the abstraction is being made. The stored editor function can be applied later when we really need to construct the editor. Therefore, we define an abstract data structure (`AGEC d`) in which we store the `ViewGEC d v` and its corresponding generic `gGEC` function for `v`. Technically this requires a type system that supports existentially quantified types as well as rank-2 polymorphism.

```
:: AGEC d = ∃.v: AGEC (ViewGEC d v)
                      (∀.ps: GECFunction (ViewGEC d v) (PSt ps))

mkAGEC :: (ViewGEC d v) → AGEC d | gGEC{|⋆|} v
mkAGEC viewGEC = AGEC viewGEC (gGEC{|⋆ → ⋆ → ⋆|} undef gGEC{|⋆|})

gGEC{|AGEC|} = ...  // similar to gGEC{|ViewGEC|}, but apply function stored in AGEC
```

The function `mkAGEC` creates the desired `AGEC` given a `viewGEC`. Looking at the type of `AGEC`, the generic system can deduce that the editor to store has to be a generic editor for type `ViewGEC d v`. To generate this editor, the generic system by default requires an editor for type `d` and type `v` as well. *We* know that in this particular case we do not use the `d`-editor at all. We can tell this to the generic system by making use of the fact that generic functions in Clean are kind indexed. The system allows us, if we wish, to explicitly specify the editors for type `d` (`undef`) and type `v` (`gGEC{|⋆|}`) to be used by the editor for `ViewGEC` (`gGEC{|⋆ → ⋆ → ⋆|}`). In this case we know that we do not need an editor for type `d` (hence the `undef`), and use the standard generic editor for type `v`. The overloading context restriction in the type of `mkAGEC` (`| gGEC{|⋆|} v`) states that for making an `AGEC d` out of a `ViewGEC d v` only an editor for type `v` is required.

We also have to define a specialized version of `gGEC` for the `AGEC` type. The corresponding generated editor applies the stored editor to the stored `ViewGEC`.

The types and kind indexed generic programming features we have used here may look complicated, but for the programmer an abstract editor is easy to make. To use a self-contained editor of type `v` as editor for type `d`, a `ViewGEC d v` has to be defined. Note that the editor for type `v` is automatically derived for the programmer by the generic system! The function `mkAGEC` stores them both into an `AGEC`. The functions `counterAGEC` and `displayAGEC` show how easy `AGEC`'s can be made. One might be surprised that the overloading context for `displayAGEC` still requires a `d`-editor (`| gGEC{|⋆|} d`). This is caused by the fact that in this particular case type `d` is used in the definition of type `Display`.

```
counterAGEC :: Int → AGEC Int
counterAGEC i = mkAGEC (counterGEC i)

displayAGEC :: d → AGEC d | gGEC{|⋆|} d
displayAGEC x = mkAGEC (displayGEC x)
```

We choose to export `AGEC d` as a Clean abstract data type. This implies that code that uses such an abstract value can not apply record selection to access the `d` value. For this purpose we provide the following obvious projection functions to retrieve the `d`-value from an `AGEC d` (`^^`) and to store a new `d`-value in an existing `AGEC d` (the infix operator `^=`).

```
(^^) :: (AGEC d) → d                          // Read current value
(^^) (AGEC viewGEC gGEC) = viewGEC.d_val

(^=) infixl :: (AGEC d) d → (AGEC d)      // Set new value
(^=) (AGEC viewGEC gGEC) nval = AGEC {viewGEC & d_val=nval} gGEC
```

The inclusion of the abstract type `AGEC d` together with its instance of `gGEC`, and the access functions provides the additional strength to the toolkit that is needed to successfully eliminate obstacle **B.2**.

We can now refine the three definitions of our running example for the last time and 'carve it in stone' as well in code fragment **(4)**. All other code fragments remain unchanged. The complete code is formed by the code fragments **(1)**. . . **(4)** (with a few auxiliary functions).

```
:: MyViewModel       = { edvalue1 :: AGEC Int
                       , edvalue2 :: AGEC Int
                       , edsum    :: AGEC Int }

toMyViewModel :: MyDataModel → MyViewModel
toMyViewModel rec     = { edvalue1 = counterAGEC rec.value1
                        , edvalue2 = counterAGEC rec.value2
                        , edsum    = displayAGEC rec.sum }

fromMyViewModel :: MyViewModel → MyDataModel
fromMyViewModel edrec = { value1 = ^^ edrec.edvalue1
                        , value2 = ^^ edrec.edvalue2
                        , sum    = ^^ edrec.edsum }
```
(4)

The advantage we have obtained now is that, if we want to pick another editor, we only have to tell which one to pick in the definition of `toMyViewModel`. The types used in `MyViewModel` all remain the same (`AGEC Int`), no matter which editor is chosen. Also the definition of `fromMyViewModel` remains unaffected. It is instructive to compare the final definition with the one at the end of Sect. 3.2.

3.4 Abstract Editors Are Compositional

In order to show the compositional nature of abstract editors, we first turn the running example into an abstract editor, say `sumAGEC` :: `AGEC Int`. In disguise, it can be used *itself* as an `Int`-editor. Following the scheme introduced above, this is done as follows:

```
sumAGEC :: Int → AGEC Int                          // see counterAGEC (3.3)
sumAGEC i = mkAGEC (sumGEC i)
where sumGEC :: Int → ViewGEC Int MyViewModel  // see counterGEC (3.2)
      sumGEC i = mkViewGEC i to upd from
      where to   = toMyViewModel o toMyData
            from = fromMyData    o fromMyViewModel
            upd  = toMyViewModel o updDataModel o fromMyViewData

            toMyData   i = initData (0,i)
            fromMyData d = d.sum
```

Now `sumAGEC`, `counterAGEC`, and `displayAGEC` are interchangeable components. If we want to experiment with variants of the running example, we simply pick the instance of our choice in the `toMyViewModel` function. This is displayed in Fig. 3.

We are setting up a library of abstract components. One of these library functions `idAGEC` (which takes a value of any type and promotes it to an abstract editor component for that type) is used in the example above. With this library it will be possible to

Alternative definition of `toMyViewModel`:

```
toMyViewModel1 rec
   = { edvalue1 = idAGEC      rec.value1
     , edvalue2 = idAGEC      rec.value2
     , edsum    = displayAGEC rec.sum }

toMyViewModel2 rec
   = { edvalue1 = idAGEC      rec.value1
     , edvalue2 = counterAGEC rec.value2
     , edsum    = displayAGEC rec.sum }

toMyViewModel3 rec
   = { edvalue1 = counterAGEC rec.value1
     , edvalue2 = sumAGEC     rec.value2
     , edsum    = displayAGEC rec.sum }
```

Corresponding GUI:

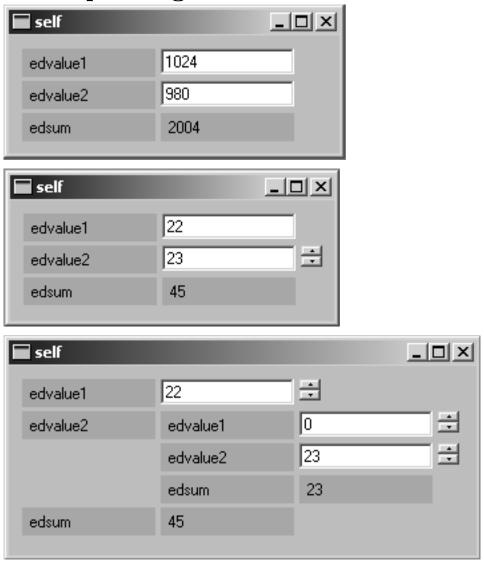


Fig. 3. *Plug-and-play* your favourite abstract editors to experiment with the running example. The only code that changes is the function `toMyViewModel`.

rapidly create GUIs in a declarative style. This can be very useful e.g. for prototyping, education, tracing and debugging purposes.

4 Related Work

Our model-view approach has several interesting features that are not present in the standard approach [12]. Firstly, because views are derived automatically, a programmer in our system does not need to explicitly 'register' nor program views. Instead, the view is *specified* by the programmer by means of the types that are used in the model. This specification, which is defined in terms of data types only, is used by the generic system to derive the actual GUI. Secondly, views can be customized via overruling instance declarations of arbitrary types. Finally, the most distinguishing feature of our model-view approach is the nature of both the model and the views. The generic framework dissects the offered type of the model into the set of generic types, each of which is mapped to an interactive model-view unit. Put in other words, our approach can truly be called model-view *all the way*.

Frameworks for the model-view paradigm in a functional language use a similar value-based approach (Claessen *et al* [8]), or an event-based version [11]. In both cases, the programmer needs to explicitly handle view registration and manipulation. In our framework, the information-flow follows the structure that is derived by the generic decomposition of the model value. This suggests that we could have based our abstract GUI definitions on a stream-based solution such as Fudgets [7]. However, stream based approaches are known to impose a much too rigid coupling between the stream based communication and the GUI structure resulting in a severe loss of flexibility and maintainability. For this reason, we have chosen to use a system with a call-back mechanism as the interface of its GUI components.

Martijn Schrage [13] also employs generic programming techniques to produce views in the *Proxima* PhD-project. It is specifically geared towards the design and development of a generic presentation-oriented XML-editor. Wolfram Kahl has developed a first version of editor combinators [17]. With editor combinators text-based structure editors can be defined and composed in a way which is similar to parser combinators.

We know of no other declarative work for describing general purpose GUIs that achieves a similar abstraction level with such a complete separation of model and view.

5 Conclusions and Future Work

We have introduced a technique for programming GUIs with the following properties:

- The programmer can *separate* application logic from view logic by defining a separate data model and view model.
- Using abstract *GEC*s (in which a complete model-view is encapsulated) the programmer can *incrementally* change the view model without modifying its type.
- The programmer can use a library of *abstract GEC*s to construct GUIs by *composition* without knowing anything about standard GUI libraries.
- The *ease of programming* with abstract GECs makes it very suited for use in education, tracing, debugging and rapid prototyping.

The Clean language that we have used in this project is a *functional* language with strong support for types, including *existential* types and *rank-2 polymorphism*. We rely essentially on *generic programming* with *kind indexed* types. The GUI part is implemented op top of the Object I/O library of Clean. The system could also have been realized in Generic Haskell using the Haskell Object I/O library.

Currently, function types are excluded from the system. We plan to include arbitrary function values by reusing the *Esther* system [15] which relies on Clean's support for *dynamics* [16]. Furthermore, we will investigate the expressive power of our graphical editor components by setting up a library for abstract GECs and by performing case studies, experimenting with multiple views and multiple models.

Acknowledgements

We like to thank Pieter Koopman and Arjen van Weelden for their valuable comments.

References

1. P. Achten and R. Plasmeijer. Interactive Functional Objects in Clean. In Clack, Hammond, and Davie, editors, *The 9th International Workshop on the Implementation of Functional Languages, IFL 1997, Selected Papers*, volume 1467 of *LNCS*, pages 304–321. St.Andrews, UK, Springer, 1998.
2. Achten, Peter and Peyton Jones, Simon. Porting the Clean Object I/O Library to Haskell. In M. Mohnen and P. Koopman, editors, *The 12th International Workshop on the Implementation of Functional Languages, IFL 2000, Selected Papers*, volume 2011 of *LNCS*, pages 194–213. Aachen, Germany, Springer, 2001.

3. Achten, Peter, van Eekelen, Marko and Plasmeijer, Rinus. Generical Graphical User Interfaces. In Greg Michaelson and Phil Trinder, editors, *Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03*, LNCS. Edinburgh, UK, Springer, 2003. To appear.
4. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, Sept. 2002.
5. Angelov, Krasimir Andreev. ObjectIO for Haskell. Description and Sources at www.haskell.org/ObjectIO/, Applications at /free.top.bg/ka2_mail/, 2003.
6. E. Barendsen and S. Smetsers. *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter 2, Graph Rewriting Aspects of Functional Programming, pages 63–102. World Scientific, 1999.
7. M. Carlsson and T. Hallgren. FUDGETS - a graphical user interface in a lazy functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture, FPCA '93*, Kopenhagen, Denmark, 1993.
8. K. Claessen, T. Vullinghs, and E. Meijer. Structuring Graphical Paradigms in TkGofer. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 251–262, Amsterdam, The Netherlands, 9-11 June 1997. ACM Press.
9. D. Clarke and A. Löh. Generic Haskell, Specifically. In J. Gibbons and J. Jeuring, editors, *Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48, Schloss Dagstuhl, July 2003. Kluwer Academic Publishers. ISBN 1-4020-7374-7.
10. Hinze, Ralf. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.
11. W. Karlsen, Einar and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *Implementation of Functional Languages, Selected Papers*, volume 1467 of *LNCS*, pages 285–303, St.Andrews, Scotland, 1997. Springer.
12. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
13. Schrage M. *Proxima, a presentation-oriented XML editor.* PhD thesis, University of Utrecht. To appear in 2004.
14. B. Shneiderman. *Designing the User Interface: strategies for effective human-computer interaction.* Addison Wesley, 2nd edition, 1992.
15. A. van Weelden and R. Plasmeijer. A functional shell that dynamically combines compiled code. In P. Trinder and G. Michaelson, editors, *Selected Papers Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL'03*. Heriot Watt University, Edinburgh, Sept. 2003. To appear.
16. M. Vervoort and R. Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 101–117. Springer, Sept. 2003.
17. Wolfram Kahl, Oliver Braun, Jan Scheffczyk. Editor Combinators - A First Account. Technical Report Nr. 2000-01, Fakultät für Informatik, Universität der Bundeswehr München, 2000.