Generic Graphical User Interfaces

Peter Achten, Marko van Eekelen, and Rinus Plasmeijer

Department of Software Technology, University of Nijmegen, The Netherlands
peter88@cs.kun.nl, marko@cs.kun.nl, rinus@cs.kun.nl

Abstract. It is important to be able to program GUI applications in a fast and easy manner. Current GUI tools for creating visually attractive applications offer limited functionality. In this paper we introduce a new, easy to use method to program GUI applications in a pure functional language such as Clean or Generic Haskell. The method we use is a refined version of the model-view paradigm.
The basic component in our approach is the Graphical Editor Component (GEC_τ) that can contain *any* value of *any* flat data type τ and that can be freely used to display and edit its value. GEC_τs can depend on others, but also on themselves. They can even be mutually dependent. With these components we can construct a flexible, reusable and customizable editor. For the realization of the components we had to invent a new generic implementation technique for interactive applications.

1 Introduction

Making an attractive Graphical User Interface (GUI) for an application is not an easy task. One can of course use the GUI library offered by the operating system (Windows, Mac, Linux). These GUI libraries are powerful (they determine what is possible on a particular OS), but the offered level of abstraction is in general rather low. Therefore, most people will prefer to use a visual editor as offered by many commercial programming environments. Such tools are very user friendly at the expense of offering limited functionality. One still has to combine the graphical elements made with the visual editor with the program code for handling the actual GUI events. Inherently, graphical representations that depend on run-time data cannot be drawn in advance. Summarizing, a visual editor is a good tool for certain simple GUI applications, but for more complicated ones one still has to struggle with low level programming code.

For dealing with more complicated applications in a simpler way, we want to define GUIs on a higher level of abstraction. Modern, pure functional programming languages enable the definition and construction of high abstraction levels. The Object I/O library [1] is probably the largest GUI library available for pure functional languages. GUI applications can be defined in Object I/O in a platform independent way. The Object I/O library has been defined in Clean and is currently available for Windows and MacOSX. A subset has been ported to Linux [11]. Object I/O has been ported to Haskell by Peter Achten and Simon Peyton Jones [2] and Krasimir Angelov [4]. Recently, it has become possible to combine

Haskell programs with Clean programs [12, 10]. Hence, in various ways Object I/O is nowadays available for a large community of pure functional programmers.

In Clean, impressive GUI applications have been made using Object I/O: e.g. the Clean IDE (including a text editor and a project manager), 2D-platform games, and the proof assistant Sparkle. The latter application in particular demonstrates the expressive power of Object I/O.

We have experienced that GUI elements such as dialogs, menus, and simple windows are relatively easy to define on a high level of abstraction. An application like the proof assistant Sparkle requires much more knowledge of the Object I/O primitives. In Sparkle [9], an action in one window (e.g. the completion of the proof of a theorem) has many consequences for the information displayed in the other windows (e.g. the list of completed theorems). Sparkle in this respect resembles the behavior of applications that are made with the well-known *model-view* paradigm [18]. The message passing primitives of Object I/O can handle such a complicated information flow. However, the learning curve to program such complicated mutual influences is rather steep.

Clearly, we need better tools to construct GUI applications on a high level of abstraction. However, we require that these tools are not as restrictive as standard GUI builders. Furthermore, we want to be able to create GUI applications in a versatile way. On the one hand it must be easy to combine standard components and on the other hand these components should be easily customized to adapt to our wishes.

In this paper we fulfill this need by introducing a new way for constructing GUI elements that respond to the change of other GUI elements. Proper customization requires a rigid separation of value versus visualization, and has lead to a refined version of the model-view paradigm (but note that at this moment only one view is supported). The basic idea is the concept of a Graphical Editor Component (a GEC_τ) with which one can display and edit values of *any* flat[1] type τ. Any change in a value is directly passed to all other GECs that depend on the changed value. Using generic programming techniques, a GEC_τ for a concrete (user defined) type τ can be generated automatically. Apart from defining the data type and customization definitions almost no additional programming effort is needed. For applications in which the particular *look* is less of an issue, it is even sufficient to provide only the data type. This makes it an excellent tool for *rapid prototyping*. All low level communication that is required to accomplish the information flow between the graphical elements, is taken care of by the system. The proposed technique is *universal*, *customizable* and *compositional*. Moreover, it is a novel application of generic programming [13, 8].

In Sect. 2 we present the basic idea of a GEC_τ. Sect. 3 shows how these GEC_τs can be combined to construct more complicated GUIs. In Sect. 4 we reveal how a GEC_τ is implemented using generic programming techniques. Sect. 5 explains how a GEC_τ can be customized to display its components in an alternative way. Thereafter, we will discuss related work. Finally, we draw conclusions and point out future work.

[1] A *flat* type is a type that does not contain any function types.

2 The concept of a Graphical Editor Component

We want to be able to make complicated GUI applications with minimal programming effort. The basic building block of our method is a customizable Graphical Editor Component (GEC_τ), which we can generate automatically for any flat type τ. More precisely, a GEC_τ is a generated function that contains a value of type τ and creates a visual component that:

1. can be used by a programmer to automatically display *any* value of type τ;
2. can be used by the application user to view and edit a value of type τ;
3. can be customized by a programmer such that its elements can be displayed in an alternative way;
4. can communicate any value change made by the user or by the program to any other component that depends on that change.

It is important to note that a GEC_τ is a very general component: in languages such as Clean and Haskell, every expression represents a value of a certain type. Since a GEC_τ can display any value of type τ, it can also be used to display any object (expression) of that type. Each GEC_τ is completely tailored to its type τ. It guarantees that each value edited is well-typed.

2.1 Interactive functional programming

Before continuing, we have to make a few remarks for people unfamiliar with functions that perform I/O in Clean. (Generic programming in Clean is explained in Sect. 4.1.) Object I/O uses an explicit environment passing style [1] supported by the uniqueness type system [5] of Clean. Consequently, any function that does something with I/O (like `mkGEC` in Sect. 2.2) is an explicit state transition function working on a program state (`PSt st`) returning at least a new program state. (In this paper the identifier `env` will be a value of this type.) The uniqueness type system of Clean will ensure single threaded use of such a state. In the Haskell variant of Object I/O, a state monad is used instead. Uniqueness type attributes that actually appear in the type signatures are not shown in this paper, in order to simplify the presentation.

2.2 Creating GEC_τs

In this section we explain in general terms what the generic function to create GEC_τs, `mkGEC`, does. In the next section we will show how `mkGEC` can be used to connect different GEC_τs.

In order to create a GEC_τ one only has to apply the generic function `mkGEC` which has the following type:

```
generic mkGEC t :: [GECAttribute] t (CallBackFunction t (PSt ps)) (PSt ps)
                                              ->2 (GEC t (PSt ps), PSt ps)
```

[2] The Clean type `a b -> c` is equivalent to the Haskell type `a -> b -> c`.

```
:: CallBackFunction t env :== t -> env -> env
```

Hence, in order to call `mkGEC` the following arguments have to be provided:

- a `GECAttribute` list controlling behavioral aspects. In this paper we restrict it to two data constructors: `OutputOnly` and `BelowPrev`,
- an initial value of type `t`,
- a call-back function defined by the programmer that will be called automatically each time the value of type `t` is edited by the user or by the program,
- the current unique state of the program.

The function `mkGEC` returns

- a record (`GEC`) containing methods that can be used to handle the newly created $GEC_{\mathtt{t}}$ component for type `t`, and
- the new unique program state (as usual for I/O handling functions in Clean).

```
:: GEC t env = { gecGetValue :: env -> (t,env)
               , gecSetValue :: t -> env -> env
               }
```

The `GEC` record that is returned contains several other useful methods for a program that are not shown above. These are methods to open and close the created GEC_τ or to show or hide its appearance. For application programmers the methods `gecGetValue` and `gecSetValue` are the most interesting. The method `gecGetValue` can be used to obtain from the GEC_τ component the currently stored value of type τ. The method `gecSetValue` can be used to *set* a *new* value in the corresponding GEC_τ.

When the user of the application changes the content of a GEC_τ, the corresponding call-back function will be automatically called with the new value. This call-back function can be used to store new values in other GEC_τs using the `gecSetValue` method of these GEC_τs as will be demonstrated in Sect. 3.

The appearance of a standard GEC_τ 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 `mkGEC`:

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

mkGEC [] (Node Leaf 1 Leaf) identity[3] env
```

This defines a window containing the $GEC_{\mathtt{Tree\ Int}}$ displaying the indicated initial value (see Fig. 1). The application user can edit this initial value in any desired order thus producing new values of type `Tree Int`. Each time a new value is created, the call-back function `identity` is called automatically. In this example this has no effect (but see Sect. 3). The shape and lay-out of the tree

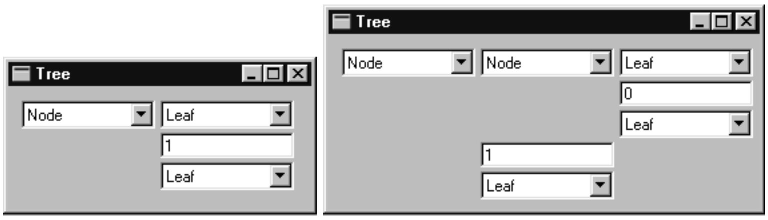


Fig. 1. The initial Graphical Editor Component for a tree of integers (Left) and a changed one (Right: with the pull-down menu the upper `Leaf` is changed into a `Node`).

being displayed adjusts itself automatically. Default values are made up by the editor when needed.

Notice that a GEC_τ is strongly typed. Only well-typed values of type τ can be created with a GEC_τ. Therefore, with a $GEC_{\texttt{Tree Int}}$ the user can only create values of type `Tree Int`. If the user makes a mistake, for instance by typing an arbitrary string into the integer box, the previous displayed integer value is restored. Any created editor also includes many other features for free, such as: automatic scroll facilities, an automatic hint box showing the type of any displayed item, and the option to hide and show any part of a data structure. All of this is generated completely given a type τ.

3 Combining Graphical Editor Components

In this section we will give some small examples how GEC_τs can be combined. A simple combination scheme is the following. If one GEC_τ B depends on a change made in a GEC_σ A, then one can pass GEC_τ B to the call-back function of A. Each time A is changed, its call-back function is called automatically and as a reaction it can set a new value in B by applying a function of type $\sigma \rightarrow \tau$. Below, conform this scheme, two such GEC_τs are created in the function `apply2GECs` such that the call-back function of `GEC_A` employs `GEC_B`.

```
apply2GECs :: (a -> b) a (PSt ps) -> (PSt ps)
apply2GECs f va env
   #[4] (GEC_B, env) = mkGEC []                         (f va)  identity     env
   #  (GEC_A, env) = mkGEC [OutputOnly,BelowPrev]    va  (set GEC_B f) env
   = env

set[5] :: (GEC b (PSt ps)) (a -> b) a (PSt ps) -> (PSt ps)
set gec f nva env = gec.gecSetValue (f nva) env
```

[3] In several examples `identity` will be used as a synonym for `const id`.

[4] The `#`-notation of Clean has a special scope rule such that the same variable name can be used for subsequent non-recursive `#`-definitions.

[5] This function `set` will also be used in other examples.

With these definitions, `apply2GECs toBalancedTree [1,5,2,8,3,9] env`, results in two GEC_τs. One for a `[Int]`, and one for a `Tree Int`. Assuming that `toBalancedTree` is a function that transforms a list into a balanced tree, any change made by the user to the displayed list[6] will automatically result into a corresponding re-balanced tree being displayed (see Fig. 2). In order to emphasize the functional dependency of the tree editor, it can not be edited by the user (controlled by the `GECAttribute OutputOnly`.)

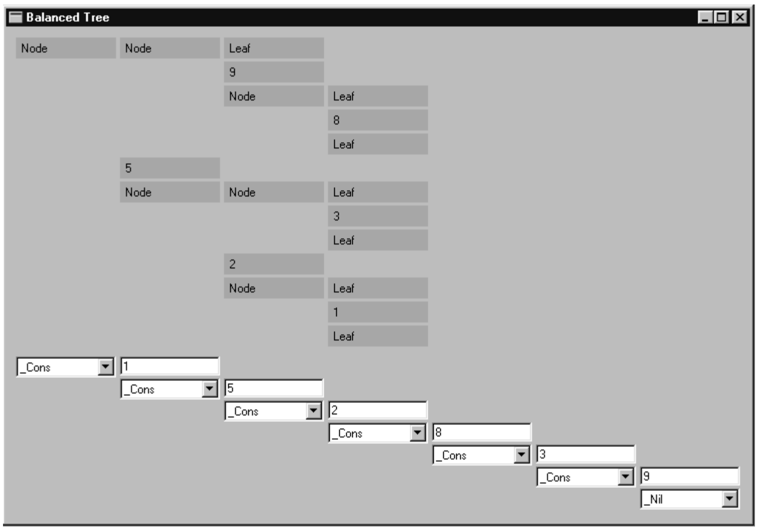


Fig. 2. A window that contains a non-editable $GEC_{\texttt{Tree Int}}$ which shows the effect of applying the function `toBalancedTree` after any user editing performed on $GEC_{\texttt{[Int]}}$.

As this example demonstrates, combining GEC_τs is very easy. One GEC_τ can have an effect on arbitrary many GEC_τs including itself! Consider:

```
selfGEC :: (a -> a) a (PSt ps) -> (PSt ps)
selfGEC f va env = new_env
where[7] (thisGEC,new_env) = mkGEC [] (f va) (set thisGEC f) env
```

Initially, this function displays the effect of applying a given function `f` to a given value `va` of type `a`. Any change a user makes using the editor automatically causes a re-evaluation of `f` to the new value thus created. Consequently, `f` has to be a function of type `a` $\rightarrow$ `a`. For example, one can use `selfGEC` to display and edit a balanced tree. Now, each time the tree is edited, it will re-balance *itself*.

[6] A Clean list is internally represented with the constructors `_Nil` and `_Cons`.

[7] The `#`-notation can not be used here since the definition of `selfGEC` is recursive.

Notice that, due to the explicit environment passing style, it is trivial in Clean to connect a GEC_τ to itself. In Haskell's monadic I/O one needs to tie the knot with `fixIO`.

In a similar way one can define mutually dependent GEC_τs. Take the following definition of `mutualGEC`.

```
mutualGEC :: a (a -> b) (b -> a) (PSt ps) -> (PSt ps)
mutualGEC va a2b b2a env = env2
where (GEC_B,env1) = mkGEC []            (a2b va) (set GEC_A b2a) env
      (GEC_A,env2) = mkGEC [BelowPrev]       va   (set GEC_B a2b) env1
```

This function displays two GEC_τs. It is given an initial value `va` of type `a`, a function `a2b :: a` $\rightarrow$ `b`, and a function `b2a :: b` $\rightarrow$ `a`. The `GEC A` initially displays `va`, while `GEC B` initially displays `a2b va`. Each time one of the GEC_τs is changed, the other will be updated automatically. The order in which changes are made is irrelevant. For example, the application `mutualGEC {euros = 3.5} toPounds toEuros` will result in an editor that calculates the exchange between pounds and euros (see Fig. 3) and vice versa.

```
exchangerate = 1.4

:: Pounds = {pounds :: Real}
:: Euros  = {euros  :: Real}

toPounds :: Euros -> Pounds
toPounds {euros} = {pounds = euros / exchangerate}

toEuros :: Pounds -> Euros
toEuros {pounds} = {euros = pounds * exchangerate}
```

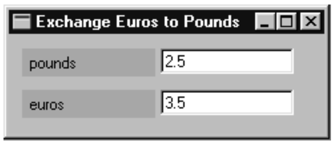


Fig. 3. Mutually dependent $GEC_{\texttt{Pounds}}$ and $GEC_{\texttt{Euros}}$ in one window.

The example of Fig. 3 may look a bit like a tiny spreadsheet, but it is essentially different since standard spreadsheets don't allow mutual dependencies between cells. Notice also the separation of concerns: the way GEC_τs are coupled is defined completely separate from the actual functionality.

4 Implementation Design of GEC_τs

Although the implementation of the system is not very big (1800 loc for the implementation modules, including comments and tracing statements) it is not possible to describe all relevant implementation aspects in this paper. We restrict ourselves to the key aspects of the design.

4.1 Generic Functions

Recently, generic functions have been added to Clean [3]. A generic function is an ultimate reusable function that allows reflection on the structure of any data in a type safe way. It is not a single function, but actually a special kind of overloaded function. One might be tempted by the idea to design some suitable universal type, and define generic functions on values of this type, and convert the changed value back to the actual type. However, it is a fundamental property of the language that (without some kind of reflection) there cannot exist a single universal type with which values of any type can be represented.

Instead, the types used by generic function definitions approximate the universal type idea. They do not constitute a single type but a *family of types*. Each concrete user defined type is represented by a different combination of members of this generic type family. Such a particular representation by itself has a type that depends on the combination of values of the generic types that is used.

Hence, to define a generic function, instances have to be defined for a finite number of types, the *generic types*, out of which any value of any type in the language can be constructed.

```
:: Unit       = Unit
:: Either a b = Left a | Right b
:: Pair a b   = Pair a b
:: TypeCons a = TypeCons InfoT a
:: DataCons a = DataCons InfoC a
```

The generic types consist of the basic types (`Bool`, `Int`, `Real`, ..., which are used to represent themselves), `Unit` (to represent a zero arity data constructor), `Pair` (product type, used to combine arguments of data constructors), and `Either` (the sum type to indicate which data constructor of a certain type is used). Furthermore, there are two special additional types `TypeCons` and `DataCons`. They contain additional information (in `InfoT` and `InfoC`) about the name and arity of the original type and data constructors. This is useful for making generic functions that can parse or print. We will need them to display the values in our graphical editor.

With a collection of generic types, values of any user-defined type can be represented, e.g. `[1]::[Int]` is represented by `TypeCons _List (Left (DataCons _Cons (Pair 1 (TypeCons _List Right (DataCons _Nil Unit)))))`, see also Fig. 4 on page 10.

Once defined by the programmer, a generic function can be applied to values of any concrete (user defined) type. The compiler will automatically add conversion functions (*bimaps*) that will transform the concrete type to the corresponding combination of generic types. Furthermore, the generic types returned by the generic function are converted back again to the actual type demanded.

In order to be able to deal with (mutual) recursive types, it is *vital* that these transformations are done in a lazy way (see [14] and [3]).

Generic functions are very useful for defining work of a general nature. Because generic functions can be specialized for any specific concrete type as well, they can also be customized easily. So far, the technique has been successfully used to define functions like equality, map, foldr, as well as for the construction of various parsers and pretty printers. Also, generic programming techniques play an important role in the implementation of automatic test systems [17]. The use of generic programming techniques for the creation of GUI applications has to our knowledge never been done before.

4.2 Creating a GEC_τ with generic functions

A GEC_τ component basically is an interactive editor for data of type τ, which can be edited in any order. A special property of such an editor is that all data elements have to be visualized and still have to be modifiable *as well*.

For our kind of interactive programs generic functions cannot be used in the standard way (in which a user type is converted lazily to the generic representation after which the generic function is applied and the result is converted back again). We need a variant in which the generic representation is not discarded but persists somehow, such that interactions (and corresponding conversions to the user types) can take place.

Consequently, we have to create a family of objects of different types to accommodate the generic representation. One solution might be to create a family of functional heaps (like Haskell `MVars` [19]). This family of heaps should then be used to create the required communication infrastructure. Instead we have used *receiver objects* [1], a facility of Object I/O. A receiver object (or, in short, *receiver*) has an internal state in the same way as functional heaps do, but they are more flexible because one can attach an arbitrary number of methods to them that also have access to the world environment. In this way, receivers enable an object-oriented style of programming. With these methods we have implemented the communication infrastructure.

The methods of a receiver that 'manages' a `t` value are invoked via Object I/O message passing functions. These require an identification value of abstract type `(GECId t)`. These are generated with the function `openGECId :: (PSt ps) → (GECId t,PSt ps)`. For the earlier mentioned `gecGetValue` and `gecSetValue` methods of the `GEC` record corresponding functions have been defined that handle the details of message passing. These are `` gecGetValue` :: (GECId t) (PSt ps) → (t,PSt ps) `` and `` gecSetValue` :: (GECId t) t (PSt ps) → (PSt ps) `` respectively. The `GEC` record methods are simply curried applications of these message passing functions.

Using receivers, we can create a family of objects that store the corresponding values of the generic type family, that communicate with their children using their typed identifiers, and that communicate with their parent using the callback interface. The topology of the receiver objects is similar to the generic representation.

In Fig. 4 we show the objects that are created to represent the value `[1]` generically. Notice the similarity between the generic representation and the topology of the receiver objects created. For this particular value, 11 receiver objects are initially created. The figure reveals that, compared to the generic

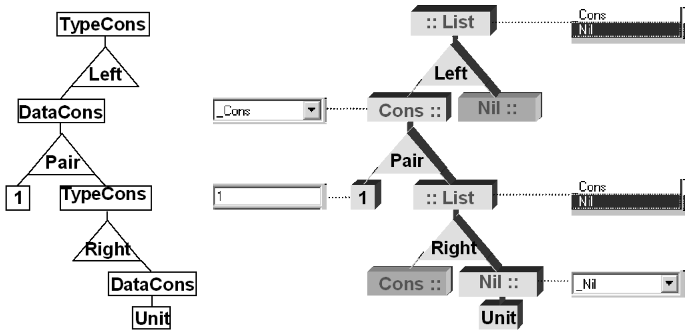


Fig. 4. The representation of `[1]` using generic types (Left), and the topology of the created receiver objects (see page 9) representing the same expression (Right).

representation, two additional (inactive) receivers (indicated by dark grey boxes, one marked `Nil ::` and one marked `Cons ::`) have been created. The reason is that in order to allow a user to change a constructor to another one, the infrastructure to handle that other constructor must already be available.

In general, a lot of communication takes place when the application user operates the editor. For instance, when a leaf value in the tree of receivers is changed, all spine receiver objects that depend on it will be informed. The required underlying communication infrastructure is far from trivial, but we only have to define it once. Each receiver object of a certain (generic) type furthermore requires a view to display and edit a value of that type. How this view can be adapted, is explained in Sect. 5.

Once the editor is created for a certain type and the corresponding views have been created, the application user can use the views to edit the value. For the implementation this means that receivers have to be created dynamically as well. If the application user makes a larger structure (e.g. `[1,2]`) out of a smaller one (e.g. `[1]`), we have to increase the number of receivers accordingly. It is possible that a receiver is not needed anymore because the user has chosen some other value. Receivers are only deleted when the whole editor is closed. Until then we mark unused receivers as inactive but remember their values. We simply close the views of these inactive receivers and reopen them when the user regrets his decision. In this way the application user can quickly switch from one value to another and backwards without being forced to retype information.

When a user switches from one data constructor to another, we have to create default values. For instance, if a `Nil` is changed into a `Cons` using an editor for a list of integers, we will generate a default integer (`0`) and a default list of integers (`Nil`). We use a generic function to make up such a default value. A programmer can easily change this by specializing this generic default function for a particular type.

5 Customizing Graphical Editor Components

5.1 The counter example

No paper about GUIs is complete without the counter example. To make such a counter we need an integer value and an up-down button. The counter has to be increased or decreased each time the corresponding button is pressed. Notice that our editors only react to changes made in the displayed data structure. Consequently, if the application user chooses `Up` two times in a row, the second `Up` will only be noticed if its value was not `Up` before. Therefore, we need a three state button with a neutral position (`Neutral`) that can be changed to either `Up` or `Down`. After being pressed it has to be reset to the neutral position again. The counter can be created automatically by applying `selfGEC updCntr (0,Neutral)`, using the concise definitions in Fig. 5.

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

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

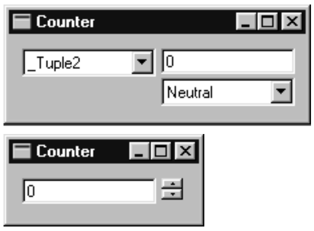
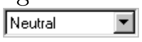


Fig. 5. Two $GEC_{\texttt{Counter}}$s. The standard one (on top) and a customized one.

The definition of the counter, the model, is intuitive and straightforward. The generated $GEC_{\texttt{Counter}}$ works as intended, and we get it for free. Unfortunately, its view is a counterexample of a good-looking counter (bottom window in Fig. 5). In this case we want to hide the tuple data constructor editor, and place the integer box next to the button editor. And finally we prefer to display  instead of  that we generate as default editor for values of the type `UpDown`.

5.2 Full customization

One of the goals in this project was to obtain fully customizable GUI editors. Here, we explain how this can be done.

Each receiver object that is generically created by the instances of `mkGEC` requires a graphical editor definition that will tell how a value of that type can be displayed and edited. This is done by a set of 'mirror' functions that takes two additional parameters. These are a fresh receiver identification value, and a view definition function. Below this is shown for the generic `Pair` case, the other generic cases proceed analogously:

```
generic mkGEC t :: [GECAttribute] t (CallBackFunction t (PSt ps)) (PSt ps)
                                                  -> (GEC t (PSt ps),  PSt ps)
mkGEC{|Pair|} gx gy as v f env
    # (pairId,env) = openGECId env
    = pairGEC pairId (pairGUI pairId) gx gy as v f env
```

The view definition function is of type `(PSt ps)` $\rightarrow$ `(GECGUI t (PSt ps), PSt ps)`: an environment based function in order to allow it to allocate the necessary resources. Hence, the view of a $GEC_{\mathtt{t}}$ is defined by a $\mathtt{GECGUI}_{\mathtt{t}}$. A $\mathtt{GECGUI}_{\mathtt{t}}$ on an environment `env` is a record with the following functions:

1. `guiLocs` reserves GUI screen space for the subcomponents of `t`. It is a function that returns a list of GUI locations[8], given its own location. For this reason it has type `Location` $\rightarrow$ `[Location]`.
2. `guiOpen` creates the view and the reserved GUI screen estate, given its own location. Hence, it must be an action. It has type `Location` $\rightarrow$ `env` $\rightarrow$ `env`.
3. `guiUpdate` defines how a new value set from the outside[9] has to be displayed. It has the obvious type `t` $\rightarrow$ `env` $\rightarrow$ `env`.
4. `guiClose` closes the view (usually a trivial Object I/O close action, which is the inverse operation of 2 above). It has the action type `env` $\rightarrow$ `env`.

For each instance of the generic family one such 'mirror' function and corresponding view definition has been predefined. For any (user defined) type, an editor is constructed automatically by combining these generic editors. The specialization mechanism of Clean can be used to change any part of a GEC_{τ} since a specialized definition for a certain type overrules the default editor for that type. Therefore, to create a good-looking counter one only has to redefine the editor for `(,)` and `UpDown`. Because the `UpDown` editor behaves as a basic value editor, its definition uses the mirror function for basic types, which is `basicGEC`:

```
mkGEC{|UpDown|} as v f env
  # (updownId,env) = openGECId env
  = basicGEC updownId (updownGUI (gecSetValue` updownId)) as v f env
```

The actual definition of `updownGUI` is 50 loc. Half of the code is 'ordinary' Object I/O code to implement the `guiOpen` function. The other functions are one-liners (this is typical).

To show the effect of the customized representation, we construct a slightly more complicated example. Below, in the record structure `DoubleCounter` we store two counters and an integer value, which will always display the sum of the two counters. Notice that the programmer can specify the wanted behavior just by applying `updCntr` on each of the counters. The sum is calculated by taking the sum of the resulting counter values. One obtains a very clear specification without any worry about the graphical representation or edit functionality. All

[8] The number of elements is actually dependent on the kind: zero for $\star$, one for $\star \rightarrow \star$, two for $\star \rightarrow \star \rightarrow \star$ etc.

[9] Note that the other direction is simply a parameter of the editor component.

one has to do is to apply `selfGEC updDoubleCntr` to the initial value `{counter1 = (0,Neutral), counter2 = (0,Neutral), sum = 0}`.

```
:: DoubleCounter = { counter1 :: Counter
                   , counter2 :: Counter
                   , sum      :: Int
                   }

updDoubleCntr::DoubleCounter -> DoubleCounter
updDoubleCntr cntr
    = { counter1 = newcounter1
      , counter2 = newcounter2
      , sum      = fst newcounter1 + fst newcounter2
      }
where newcounter1 = updCntr cntr.counter1
      newcounter2 = updCntr cntr.counter2
```

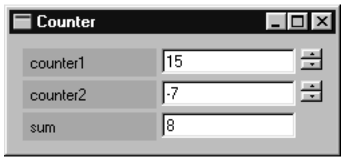


Fig. 6. An editor with two counters and a resulting sum.

For large data structures it may be infeasible to display the complete data structure. Customization can be used to define a GEC_τ that creates a view on a finite subset of such a large data structure with buttons to browse through the rest of the data structure. This same technique can also be used to create GEC_τs for lazy infinite data structures. For these infinite data structures customization is a must since clearly they can never be fully displayed.

6 Related Work

The system described in this paper is a refined version of the well-known *model-view* paradigm [18], introduced by Trygve Reenskaug in the language Smalltalk (then named as 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). Because views are derived automatically, a programmer in our system does not need to explicitly 'register' nor program views. Views can be customized via overruling instance declarations of arbitrary types.

A distinguishing feature of our approach is the distributed nature of both the model and the views. The model is distributed using the generic decomposition of the model value. The subvalues are stored in the receivers. This implies that it should be relatively easy to distribute this framework over a collection of distributed interactive processes. The view is distributed as well, as each view of a generic component is responsible for showing that particular generic instance only. Its further responsibility is to define GUI space for the subcomponents.

Frameworks for the model-view paradigm in a functional language use a similar value-based approach as we do (Claessen *et al* [7]), or an event-based version [15]. 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 used a stream-based solution such as FUDGETS [6]. However, stream based approaches are known to impose a much to 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 call-back mechanism as the interface of our GEC_τ components.

The Vital project [16] has similar goals as our project. Vital is an interactive graphical environment for direct manipulation of Haskell-like scripts. Shared goals are: direct manipulation of functional expressions (Haskell expressions vs. flat values), manipulation of custom types, views that depend on the data type (*data type styles*), *guarded* data types (we use `selfGEC`), and the ability to work with infinite data structures. Differences are that our system is completely implemented in Clean, while the Vital system has been implemented in Java. This implies that our system can handle, by construction, all flat Clean values, and all values are obviously well-typed. In addition, the purpose of a GEC_τ is to edit values of type τ, while the purpose of a Vital session is to edit Haskell scripts.

7 Conclusions and Future Work

Graphical Editor Components are built on top of the Object I/O GUI library. The Object I/O library offers a lot of functionality (e.g. one can define menus, draw objects, make timers, etcetera) and it is also very flexible, but at the expense of a steep learning curve. Graphical Editor Components offer a more limited functionality: a customizable view and editor for any type. The customization abilities make it possible to incorporate the functionality of Object I/O. It does not exclude Object I/O; it is fully integrated with it. One can still use windows, dialogs, menus, timers, and so on. The abstraction layer offered by the Graphical Editor Components is much higher. The learning curve is short and flat because one basically has to learn only the `mkGEC` function instead of approximately 500 Object I/O functions. The arguments of `mkGEC` are similar to most Object I/O functions. The most important advantages are:

- for any value of any flat type one gets an editor for free;
- the editor can be used to give type safe input to the application;
- any output or intermediate result can be displayed;
- visualization is separated from the value infrastructure and is customizable by redefining the components that need to be displayed differently;
- editors can be combined including mutually dependent editors.

The presented method offers a good separation of concerns. The specification of the wanted functionality of a component is completely separated from the specification of its graphical representation. One even obtains a default graphical representation for free. This makes it an excellent tool for rapid prototyping. Also one can abstract from the way components are connected. As a result, complicated interactive applications can be created without a lot of understanding of graphical I/O handling.

Editors can be used for programming GUI objects, from simple dialogs to complicated spreadsheets. They can also be used for tracing and debugging.

The automatic generation of components was only possible thanks to the generic programming facilities of Clean. The interactive nature of the components caused some interesting implementation problems. We had to store the generic representation in special objects (Object I/O receivers). We also hit on a disturbing limitation of the current implementation in Clean: one cannot overload generic functions in their generic type. For this reason we introduced the 'mirror' functions in Sect. 5. We intend to solve this.

We plan to extend the system with support for non-flat types. This will require the ability to deal with function types. An interesting direction of research seems to be to use the Esther shell of the experimental operating system Famke, which deals with creating and composing functions in an interactive and dynamically typed way. This shell is written in Clean as well [20].

Furthermore, we will investigate the expressive power of our graphical editor components by conducting experiments and case studies.

Acknowledgements

The authors would like to thank Arjen van Weelden, Pieter Koopman, Ronny Wichers Schreur and the anonymous referees 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. 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*, pages 168–186. Älvsjö, Sweden, Sept. 2002.
4. Angelov, Krasimir Andreev. ObjectIO for Haskell. Description and Sources at www.haskell.org/ObjectIO/, Applications at /free.top.bg/ka2_mail/, 2003.
5. 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.

6. 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.
7. 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.
8. 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.
9. M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *Selected Papers from the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2001. Springer.
10. Divinszky P. Haskell - Clean Compiler. ELTE, Budapest, 2003. Software at aszt.inf.elte.hu/~fun_ver/2003/software/HsCleanAll2.0.2.zip.
11. Fulgham, Brent. The Clean ObjectIO Library under Linux (Gtk+). Description at people.debian.org/~bfulgham/clean_examples, May 2003. Sources at people.debian.org/~bfulgham/clean/objectio-linux.tar.gz.
12. Hegedus H. Haskell to Clean Front End. Master's thesis, ELTE, Budapest, Hungary, 2001.
13. 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.
14. Hinze, Ralf and Peyton Jones, Simon. Derivable Type Classes. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *ENTCS*. Montreal, Canada, Elsevier Science, 2001.
15. 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.
16. Keith Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
17. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic Automated Software Testing. 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 84–100. Springer, 2003.
18. 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.
19. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St.Petersburg Beach, Florida, 1996. ACM.
20. A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. 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 215–231. Springer, Sept. 2002.