Automatic Generation of Editors for Higher-Order Data Structures

Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden
Nijmegen Institute for Information and Computer Science, Nijmegen University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{peter88, marko, rinus, arjenw}@cs.kun.nl

Abstract. With generic functional programming techniques, we have eased GUI programming by constructing a programming toolkit with which one can create GUls in an abstract and compositional way, using type-directed Graphical Editor Components (GECs). In this toolkit, the programmer specifies a GUI by means of a data model instead of low-level GUI programming. In earlier versions of this toolkit, the data model must have a first-order type. In this paper we show that the programming toolkit can be extended in two ways, such that the data model can contain higher-order data structures. We added support for dynamic polymorphic higher-order editors using the functional shell Esther. By combining the earlier developed techniques of generic GECs, abstract editors, we also added statically typed higher-order editors. In principle this solution extends our GUI programming toolkit with the full expressive power of functional programming languages.

1 Introduction

In the last decade, Graphical User Interfaces (GUls) have become the standard for user interaction. Programming these interfaces can be done without much effort when the interface is rather static, and for many of these situations excellent tools are available. However, when there is more dynamic interaction between interface and application logic, such applications require tedious manual programming in any programming language. Programmers need to be skilled in the use of a large programming toolkit.

The goal of the Graphical Editor project is to obtain a concise programming toolkit that is abstract, compositional, and type-directed. Abstraction is required to reduce the size of the toolkit, compositionality reduces the effort of putting together (or altering) GUI code, and type-directed automatic creation of GUls allows the programmer to focus on the data model. In contrast to visual programming environments, programming toolkits can provide ultimate flexibility, type safety, and dynamic behavior within a single framework. We use a pure functional programming language (Clean [20]) because functional programming languages have proven to be very suited for creating abstraction layers on top of each other. Additionally, they have strong support for type definitions and type safety.
Our programming toolkit utilizes the *Graphical Editor Component* (*GEC*) [6] as universal building block for constructing GUIs. A *GEC* is a graphical editor for values of any *monomorphic first-order* type \( t \). This type-directed creation of *GECs* has been obtained by *generic programming* techniques [8, 17, 16]. Generic programming is extremely beneficial when applied to composite custom types. With generic programming one defines a family of functions that depend on the *structure* of types. Although one structural element is the *function type* constructor \((-\)}\), it is fundamentally impossible to define a generic function that edits these higher-order values directly, because pure functional programs cannot look inside functions without losing referential-transparency (for instance by distinguishing \( \lambda x \to x+1 \) from \( \lambda x \to 1+x \)).

In this paper we extend the *GEC* toolkit in two ways, such that it can construct higher-order value editors. The first extension uses run-time *dynamic typing* [1, 19], which allows us to include them in the *GEC* toolkit, but this does not allow type-directed GUI creation. It does, however, enable the toolkit to use polymorphic higher-order functions and data structures. The second extension uses compile-time static typing, in order to gain monomorphic higher-order type-directed GUI creation of *abstract* types. It uses the *abstraction mechanism* of the *GEC* toolkit [7].

Both extensions require a means of using functional expressions, entered by the user, as functional values. Instead of writing our own parser/interpreter/type inference system we use the *functional Esther shell* [22], which provides type checking at the command line and can use compiled functions from disk. These functions can have arbitrary size and complexity, and even interface with the imperative world. Esther makes extensive use of dynamic types. Dynamic types turn arbitrary (polymorphic, higher-order) data structures (for instance of type \([Int \to Int]\) or \((Tree \, a) \to a\)) into a *first-order* data structure of type *Dynamic* without losing the original type.

Contributions of this paper are:

- We provide type-safe *expression* editors, which are needed for higher-order value editors. We obtain, as a bonus, the ability to edit first-order values using *expressions*. Another bonus: within these expressions one can use compiled functions from disk, incorporating *real world* functionality.
- The programming toolkit can now create polymorphic dynamically typed, and monomorphic statically typed, higher-order value editors.
- The programming toolkit is type-safe and type-directed.

This paper is structured as follows. Section 2 contains an overview of the first-order *GEC* toolkit. In Sect. 3 we present the first extension, in which we explain how Esther incorporates *expressions* as *functional values* using dynamic types. We present in Sect. 4 the second extension, and explain how we obtain higher-order type-directed GUI creation using the *abstraction mechanism* of the *GEC* toolkit. Section 5 gives examples of the new system that illustrate its expressive power. We discuss related work in Sect. 6 and conclude in Sect. 7.
Finally, a note on the implementation and the examples in this paper. The project has been realized in Clean. Familiarity with Haskell [18] is assumed, relevant differences between Haskell and Clean are explained in footnotes. The GUI code is mapped to Object I/O [4], which is Clean’s library for GUIs. Given sufficient support for dynamic types, the results of this project can be transferred to Generic Haskell [12], using the Haskell [18] port of Object I/O [3]. The complete code of all examples (including the complete GEC implementation in Clean) can be downloaded from http://www.cs.kun.nl/~clean/gec.

2 The GEC Programming Toolkit

With the GEC programming toolkit [6], one constructs GUI applications in a compositional way using a high level of abstraction. The basic building block is the Graphical Editor Component (GEC). It is generated by a generic function, which makes the approach type-directed.

Before explaining GECs in more detail, we need to point out that Clean uses an explicit multiple environment passing style [2] for I/O programming. As GECs 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 ps). 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. The uniqueness type system [9] of Clean ensures single threaded use of the environment. To improve the readability, uniqueness type attributes that actually appear in the type signatures are not shown. Furthermore, the code has been slightly simplified, leaving out a few details that are irrelevant for this paper.

Graphical Editor Components. A $GEC_t$ is an editor for values of type t. It is generated with a generic function [16, 8]. A generic function is a meta function that works on a description of the structure of types. For any concrete type t, the compiler is able to automatically derive an instance function of this generic function for the type t. The power of a generic scheme is that we obtain an editor for free for any monomorphic data type. This makes the approach particularly suited for rapid prototyping.

The generic function $gGEC$ creates GECs. It takes a definition $(GECDef t env)$ of a $GEC_t$ and creates the $GEC_t$ object in the environment. It returns an interface $(GECInterface t env)$ to that $GEC_t$ object. The environment env is in this case (PSt ps), since $gGEC$ uses Object I/O.

$\text{generic}^1 gGEC t :: (GECDef t (PSt ps)) (PSt ps) \rightarrow (GECInterface t (PSt ps), PSt ps)^2$

The $(GECDef t env)$ consists of three elements. The first is a string that identifies the top-level Object I/O element (window or dialog) in which the editor

---

1 $\text{generic } f t :: T(t)$ introduces a generic function $f$ with type scheme $T(t)$. Keywords are type-set in **bold**.

2 Clean separates function arguments by whitespace, instead of $\rightarrow$.
must be created. The second is the initial value of type \( t \) of the editor. The third is a callback function of type \( t \rightarrow \text{env} \rightarrow \text{env} \). This callback function tells the editor which parts of the program need to be informed of user actions. The editor uses this function to respond to changes to the value of the editor.

\[
\text{GECDef } t \text{ env} ::= (\text{String}, t, \text{CallBackFunction } t \text{ env})
\]

The (GECInterface \( t \text{ env} \)) is a record that contains all \textit{methods} of the newly created \( GEC_t \).

\[
\text{GECInterface } t \text{ env} = \{ \text{gecGetValue} :: \text{env} \rightarrow (t, \text{env}) , \text{gecSetValue} :: t \rightarrow \text{env} \rightarrow \text{env} \}^5
\]

The gecGetValue method returns the current value, and gecSetValue sets the current value of the associated \( GEC_t \) object. Programs can be constructed combining editors by tying together the various gecSetValues and gecGetValues. We are working on an arrow combinator library that abstracts from the necessary plumbing [5]. For the examples in this paper, it is sufficient to use the following tying function:

\[
\text{selfGEC} :: \text{String } (t \rightarrow t) t (\text{PSt ps}) \rightarrow (\text{PSt ps}) \uparrow^6 \text{gGEC}[x] \uparrow t
\]

\[
\text{selfGEC} s f v \text{ env} = \text{env1}
\]

where \( \langle\{\text{gecSetValue}\}, \text{env1}\rangle = \text{gGEC}[x] \uparrow (s, f v, \lambda x \rightarrow \text{gecSetValue} (f x)) \uparrow \text{env} \)

Given an \( f \) of type \( t \rightarrow t \) on the data model of type \( t \) and an initial value \( v \) of type \( t \), \text{selfGEC} gui f v creates the associated \( GEC_t \) using gGEC (hence the context restriction). \text{selfGEC} creates a feedback loop that sends every edited output value back as an input to the same editor, after applying the function \( f \).

\textbf{Example 1:} The standard appearance of a GEC is given by the following program that creates an editor for a \textit{self-balancing} binary tree:

\begin{verbatim}
module Editor
import StdEnv, StdIO, StdGEC

Start :: *World \rightarrow *World
Start world = startIO MDI Void myEditor world

myEditor :: (PSt ps) \rightarrow (PSt ps)
myEditor = selfGEC "Tree" balance (Node Leaf 1 Leaf)

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

\end{verbatim}

In this example, we create a \( GEC_{\text{Tree Int}} \) which displays the indicated initial value \( \text{Node Leaf 1 Leaf} \) (upper screen shot). The user can manipulate this

---

3 Type definitions are preceded by :::

4 ::= introduces a synonym type.

5 \( \{f_0 :: t_0, \ldots, f_n :: t_n\} \) denotes a record with field names \( f_i \) and types \( t_i \).

6 In a function type, \( \uparrow \) introduces all overloading class restrictions.
value in any desired order, producing new values of type *Tree Int* (e.g., turning
the upper *Leaf* into a *Node* with the pull-down menu). Each time a new value
is created or edited, the feedback function *balance* is applied. *balance* takes an
argument of type *Tree a* and returns the tree after balancing it. The shape and
lay-out of the tree being displayed adjusts itself automatically. Default values
are generated by the editor when needed.

Note that the only things that need to be specified by the programmer are
the initial value of the desired type, and the feedback function. In all remaining
examples, we only modify *myEditor* and the type for which an instance of *gGEC*
derives.

The tree example shows that a *GEC* explicitly reflects the structure of type
*t*. For the creation of GUI applications, we need to model both specific GUI ele­
ments (such as buttons) and layout control (such as horizontal, vertical layout).
This has been done by *specializing *gGEC* [6]* for a number of types that either rep­
resent GUI elements or layout. Here are the types and their *gGEC* specialization
that are used in the examples in this paper:

:: Display a = Display a // a non-editable GUI: e.g., [HelpText].
:: Hide a = Hide a // an invisible GUI, useful for state.
:: UpDown = UpPressed | DownPressed | Neutral // a spin button: #.

### 3 Dynamically Typed Higher-Order GECs

In this section we show how to extend *GECs* with the ability to deal with
functions and expressions. Because functions are opaque, the solution requires
a means of interpreting functional expressions as functional values. Instead of
writing our own parser/interpreter/type inference system we use the *Esther* shell
[22] (Sect. 3.1).

Esther enables the user to enter expressions (using a subset of *Clean*) that are
dynamically typed, and transformed into values and functions using compiled
code. It is also possible to reuse earlier created functions, which are stored on
disk. Its implementation relies on the *dynamic type system* [1, 19, 23] of *Clean*.

The shell uses a text-based interface, and hence it makes sense to create a
special string-editor (Sect. 3.2), which converts any string into the corresponding
dynamically typed value. This special editor has the same power as the Esther
command interpreter and can deliver any dynamic value, including higher-order
polymorphic functions.

#### 3.1 Dynamics in Clean

A *dynamic* is a value of static type *Dynamic*, which contains an expression as well
as a representation of its static type, e.g., *dynamic 42 :: Int, dynamic map fst
:: ∀a b: [(a, b)] → [a]*. Basically, dynamic types turn every (first and higher-
order) data structure into a first-order structure, while providing run-time access
to the original type and value.
Function alternatives and case patterns can match on values of type `Dynamic`. Such a pattern match consists of a value pattern and a type pattern, e.g., `[4, 2] :: [Int]`. The compiler translates a pattern match on a type into a run-time type unification. If the unification is successful, type variables in a type pattern are bound to the offered type. Applying dynamics at run-time will be used to create an editor that changes according to the type of entered expressions (Sect. 3.2, Example 2).

```haskell
dynamicApply :: Dynamic Dynamic → Dynamic
dynamicApply (f :: a → b) (x :: a) = dynamic f x :: b
dynamicApply df dx = dynamic "Error" :: String
```

dynamicApply tests if the argument type of the function `f`, inside its first argument, can be unified with the type of the value `x`, inside the second argument. dynamicApply can safely apply `f` to `x`, if the type pattern match succeeds. It yields a value of the type that is bound to the type variable `b` by unification, wrapped in a dynamic. If the match fails, it yields a string in a dynamic.

Type variables in type patterns can also relate to type variables in the static type of a function. A `*` behind a variable in a pattern associates it with the same type variable in the static type of the function.

```haskell
matchDynamic :: Dynamic → t | TC t
matchDynamic (x :: t*) = x
```

The static type variable `t`, in the example above, is determined by the static context in which it is used, and imposes a restriction on the actual type that is accepted at run-time by `matchDynamic`. The function becomes overloaded in the predefined `TC` (type code) class. This makes it a type dependent function [19].

The dynamic run-time system of Clean supports writing dynamics to disk and reading them back again, possibly in another program or during another execution of the same program. This provides a means of type safe communication, the ability to use compiled plug-ins in a type safe way, and a rudimentary basis for mobile code. The dynamic is read in lazily after a successful run-time unification. The amount of data and code that the dynamic linker links is, therefore, determined by the evaluation of the value inside the dynamic.

```haskell
writeDynamic :: String Dynamic env → (Bool, env) | FileSystem env
readDynamic :: String env → (Bool, Dynamic, env) | FileSystem env
```

Programs, stored as dynamics, have Clean types and can be regarded as a typed file system. We have shown that `dynamicApply` can be used to type check any function application at run-time using the static types stored in dynamics. Combining both in an interactive ‘read expression – apply dynamics – evaluate and show result’ loop, already gives a simple shell that supports the type checked run-time application of programs to documents. The `composeDynamic` function below, taken from the Esther shell, applies dynamics and infers the type of an expression.

```haskell
composeDynamic :: String env → (Dynamic, env) | FileSystem env
showValueDynamic :: Dynamic → String
```
composeDynamic expr env parses expr. Unbound identifiers in expr are resolved by reading them from the file system. In addition, overloading is resolved. Using the parse tree of expr and the resolved identifiers, the dynamicApply function is used to construct the (functional) value v and its type τ. These are packed in a dynamic v :: τ and returned by composeDynamic. In other words, if env ′ ← expr :: τ and ![expr]_{env} = v then composeDynamic expr env = (v :: τ, env). The showValueDynamic function yields a string representation of the value inside a dynamic.

3.2 Creating a GEC for the Type Dynamic

With the composeDynamic function, an editor for dynamics can easily be constructed. This function needs an appropriate environment to access the dynamic values and functions (plug-ins) that are stored on disk. The standard (PSt ps) environment used by the generic gGEC function (Sect. 2) is such an environment. This means that we can simply use composeDynamic in a specialized editor to offer the same functionality as the command line interpreter. Instead of Esther’s console we use a String editor as interface to the application user. In addition we need to convert the provided string into the corresponding dynamic. We therefore define a composite data type DynString and a specialized gGEC-editor for this type (a GECDynString) that performs the required conversions.

:: DynString = DynStr Dynamic String

The choice of the composite data type is motivated mainly by simplicity and convenience: the string can be used by the application user for typing in the expression. It also stores the original user input, which cannot be extracted from the dynamic when it contains a function.

Now we specialize gGEC for this type DynString. The complete definition of gGEC[DynString] is given below.

gGEC{DynString} (gui,DynStr _ expr,dynStringUpdate) env

$\equiv$ (stringGEC, env) = gGEC(|*) (gui, expr, stringUpdate dynStringUpdate) env

= (\{ gecSetValue = dynSetValue stringGEC.gecSetValue
  , gecGetValue = dynGetValue stringGEC.gecGetValue \}, env)

where dynSetValue stringSetValue (DynStr _ expr) env

= stringSetValue expr env

dynGetValue stringGetValue env

$\equiv$ (nexpr, env) = stringGetValue env

$\equiv$ (ndyn, env) = composeDynamic nexpr env

= (DynStr ndyn nexpr, env)

dynStringUpdate dynStringUpdate nexpr env

$\equiv$ (ndyn, env) = composeDynamic nexpr env

= dynStringUpdate (DynStr ndyn nexpr) env

The created GECDynString displays a box for entering a string by calling the standard generic gGEC[^İZ] function for the value expr of type String, yielding a

[^7] This is Clean’s ‘do-notation’ for environment passing.
The DynString-editor is completely defined in terms of this String-editor. It only has to take care of the conversions between a String and a DynString. This means that its gecSetValue method dynSetValue simply sets the string component of a new DynString in the underlying String-editor. Its gecGetValue method dynGetValue retrieves the string from the String-editor, converts it to the corresponding Dynamic by applying composeDynamic, and combines these two values in a DynString-value. When a new string is created by the application user, the callback function stringUpdate is evaluated, which invokes the callback function dynStringUpdate (provided as an argument upon creation of the DynString-editor), after converting the String to a DynString.

It is convenient to define a constructor function mkDynStr that converts any input expr, that has value v of type τ, into a value of type DynString guaranteeing that if v :: τ and \[expr\] = v, then (DynStr (v :: τ) expr) :: DynString.

\[mkDynStr :: a \rightarrow \text{DynString} \mid TC a\]

\[mkDynStr x = \text{let } dx = \text{dynamic } x \text{ inDynStr } dx \text{ (showValueDynamic } dx)\]

**Example 2:** We construct an interactive editor that can be used to test functions. It can be a newly defined function, say \(\lambda x \rightarrow x^2\), or any existing function stored on disk as a Dynamic. Hence the tested function can vary from a small function, say factorial, to a large complete application.

```plaintext
:: MyRecord =
{ function :: DynString
, argument :: DynString
, result :: DynString }

myEditor = selfGEC "test" guiApply (initval id 0)
where
  initval f v =
{ function = mkDynStr f 
, argument = mkDynStr v 
, result = mkDynStr \(f \cdot v\) }

guiApply r =
{ function = DynStr \(f :: a \rightarrow b\) 
, argument = DynStr \(v :: a\) 
} = \{r & result = mkDynStr \(f \cdot v\)\}

\[\text{guiApply } r = r\]
```

The type MyRecord is a record with three fields, function, argument, and result, all of type DynString. The user can use this editor to enter a function definition and its argument. The selfGEC function will ensure that each time a new string is created with the editor "test", the function guiApply is applied that provides a new value of type MyRecord to the editor. The function guiApply tests, in a similar way as the function dynamicApply (see Sect. 3.1), whether the type of the supplied function and argument match. If so, a new result is calculated. If not, nothing happens.

This editor can only be used to test functions with one argument. What happens if we edit the function and the argument in such a way that the result is not a plain value but a function itself? Take, e.g., as function the twice function.

\[x =: e\] binds x to e.

\[\{r \& f_0=v_0, \ldots, f_n=v_n\}\] is a record equal to r, except that fields \(f_i\) have value \(v_i\).
\( \lambda x \to f \,(f\,x) \), and as argument the increment function \((+\ 1)\). Then the result is also a function \( \lambda x \to ((+)\ 1)\ ((+)\ 1\ x) \). The editor displays \(<\text{function}>\) as result. There is no way to pass an argument to the resulting function.

With an editor like the one above, the user can enter expressions that are automatically converted into the corresponding Dynamic value. As in the shell, unbound names are expected to be dynamics on disk. Illegal expressions result in a Dynamic containing an error message.

To have a properly higher-order Dynamic application example, one needs an editor in which the user can type in functions of arbitrary arity, and subsequently enter arguments for this function. The result is then treated such that, if it is a function, editors are added dynamically for the appropriate number of arguments. This is explained in the following example.

**Example 3:** We construct a test program that accepts arbitrary expressions and adds the proper number of argument editors, which again can be arbitrary expressions. The number of arguments cannot be statically determined and has to be recalculated each time a new value is provided. Instead of an editor for a record, we therefore create an editor for a list of tuples. Each tuple consists of a string used to prompt to the user, and a DynString-value. The tuple elements are displayed below each other using the predefined list editor `vertlistAGEC` and access operator `^`, which will be presented in Sect. 4.1. The `selfGEC` function is used to ensure that each change made with the editor is tested with the guiApply function and the result is shown in the editor.

```plaintext
myEditor = selfGEC "test" (guiApply o (^))
  (vertlistAGEC [show "expression " 0])
where
guiApply [f=:_,(DynStr d _)):args]
  = vertlistAGEC [f:check (fromDynStr d) args]
where
  check (f::a -> b) [arg=:_,(DynStr (x::a) _)):args]
    = [arg : check (dynamic f x) args]
  check (f::a -> b) _ = [show "argument " "??"
  check (x::a) _ = [show "result " x]

  show s v = (Display s,mkDynStr v)
```

The key part of this example is formed by the function `check` which calls itself recursively on the result of the dynamic application. As long as function and argument match, and the resulting type is still a function, it will require another argument which will be checked for type consistency. If function and argument do not match, "??" is displayed, and the user can try again. As soon as the resulting type is a plain value, it is evaluated and shown using the data constructor `Display`, which creates a non-editable editor that just displays its value. With this editor, any higher-order polymorphic function can be entered and tested.
4 Statically Typed Higher-Order GECs

The editors presented in the previous section are flexible because they deliver a Dynamic (packed into the type DynString). They have the disadvantage that the programmer has to program a check, such as the check function in the previous example, on the type consistency of the resulting Dynamics.

In many applications it is statically known what the type of a supplied function must be. In this section we show how the run-time type check can be replaced by a compile-time check, using the abstraction mechanism for GECs. This gives us a second solution for higher-order data structures that is statically typed, which allows, therefore, type-directed generic GUI creation.

4.1 Abstract Graphical Editor Components

The generic function gGEC derives a GUI for its instance type. Because it is a function, the appearance of the GUI is completely determined by that type. This is in some cases much to rigid. One cannot use different visual appearances of the same type within a program. For this purpose abstract GECs (AGEC) [7] have been introduced. An instance of gGEC for AGEC has been defined. Therefore, an AGECd can be used as a GECd, i.e., it behaves as an editor for values of a certain domain, say of type d. However, an AGECd never displays nor edits values of type d, but rather a view on values of this type, say of type v. Values of type v are shown and edited, and internally converted to the values of domain d. The view is again generated automatically as a GECv. To makes this possible, the ViewGEC d v record is used to define the relation between the domain d and the view v.

:: ViewGEC d v
   = { d_val :: d // initial domain value
       , d_oldv_to_v :: d → (Maybe v) → v // convert domain value to view value
       , update_v :: v → v // correct view value
       , v_to_d :: v → d } // convert view value to domain value

It should be noted that the programmer does not need to be knowledgeable about Object I/O programming to construct an AGECd with a view of type v. The specification is only in terms of the involved data domains. The complete interface to AGECs is given below.

:: AGEC d // abstract data type
mkAGEC :: (ViewGEC d v) → AGEC d | gGEC[|x|] v
(\^\^) :: (AGEC d) → d // Read current domain value
(\^\_\^) infixl :: (AGEC d) d → AGEC d // Set new domain value

The ViewGEC record can be converted to the abstract type AGEC, using the function mkAGEC above. Because AGEC is an abstract data type we need access functions to read (^\^) and write (^\_\^) its current value. AGECs allow us to define arbitrarily many editors gec_2 :: AGEC_3 that have a private implementation of type GEC_4. Because AGEC is abstract, code that has been written for editors that manipulates some type containing AGEC_4, does not change when the value
of type \( \text{AGEC}_d \) is exchanged for another \( \text{AGEC}_d \). This facilitates experimenting with various designs for an interface without changing any other code.

We built a collection of functions creating abstract editors for various purposes. Below, we summarize only those functions of the collection that are used in the examples in this paper:

- \( \text{vertlistAGEC :: } [a] \rightarrow \text{AGEC} [a] \mid \text{gGECH}[*] a \quad \text{// all elements displayed in a column} \)
- \( \text{counterAGEC :: } a \rightarrow \text{AGEC} a \mid \text{gGECH}[*] \), IncDec a \quad \text{// a special number editor} \)
- \( \text{hidAGEC :: } a \rightarrow \text{AGEC} a \quad \text{// identity, no editor} \)
- \( \text{displayAGEC :: } a \rightarrow \text{AGEC} a \mid \text{gGECH}[*] a \quad \text{// identity, non-editable editor} \)

The \textit{counter} editor below is a typical member of this library.

\[
\text{counterAGEC :: } a \rightarrow \text{AGEC} a \mid \text{gGECH}[*] \), IncDec a
\]
\[
\text{counterAGEC } j = \text{mkAGEC} \{ \text{d_val = } j \), \text{d_oldv_to_v = } \lambda i \rightarrow (i, \text{Neutral}) \), \text{update_v = } \text{updateCounter, v_to_d = } \text{fst} \}
\]

\[
\text{where updateCounter } (n, \text{UpPressed}) = (n+\text{one}, \text{Neutral})
\]
\[
\text{updateCounter } (n, \text{DownPressed}) = (n-\text{one}, \text{Neutral})
\]
\[
\text{updateCounter } (n, \text{Neutral}) = (n, \text{Neutral})
\]

A programmer can use the counter editor as an integer editor, but because of its internal representation it presents the application user with an edit field combined with an up-down, or spin, button. The \text{updateCounter} function is used to synchronize the spin button and the integer edit field. The right part of the tuple is of type UpDown (Sect. 2), which is used to create the spin button.

### 4.2 Adding Static Type Constraints to Dynamic GECs

The abstraction mechanism provided by \( \text{AGEC} \)s is used to build type-directed editors for higher-order data structures, which check the type of the entered expressions dynamically. These statically typed higher-order editors are created using the function \text{dynamicAGEC}. The full definition of this function is specified and explained below.

\[
\text{dynamicAGEC :: } d \rightarrow \text{AGEC} d \mid \text{TC} d
\]
\[
\text{dynamicAGEC } x = \text{mkAGEC} \{ \text{d_val = } x \), \text{d_oldv_to_v = } \text{toView} \), \text{update_v = } \text{updView x, v_to_d = } \text{fromView x} \}
\]

\[
\text{where toView newx Nothing} = \text{let } dx = \text{mkDynStr newx in } (dx, \text{hidAGEC dx})
\]
\[
\text{toView } _{\text{Just oldx}} = \text{oldx}
\]
\[
\text{fromView } _{\text{Just oldx}} = \text{case } "\text{oldx of } \text{DynStr } (x::d^\prime)_{\text{.}} \rightarrow x
\]
\[
\text{updView } _{\text{Just oldx}} = \text{case } "\text{oldx of } \text{DynStr } (x::d^\prime)_{\text{.}} \rightarrow x
\]
\[
\text{updView } _{\text{Just oldx}} = \text{hidAGEC newx)
\]

\[
\text{updView } _{\text{Just oldx}} = ("\text{oldx, oldx})
\]

The abstract Dynamic editor, which is the result of the function \text{dynamicAGEC} initially takes a value of some statically determined type \( d \). It converts this value
into a value of type `DynString`, such that it can be edited by the application user as explained in Sect. 3.2. The application user can enter an expression of arbitrary type, but now it is ensured that only expressions of type `d` are approved.

The function `updView`, which is called in the abstract editor after any edit action, checks, using a type pattern match, whether the newly created dynamic can be unified with the type `d` of the initial value (using the `-notation in the pattern match as explained in Sect. 3.1). If the type of the entered expression is different, it is rejected\(^\text{10}\) and the previous value is restored and shown. To do this, the abstract editor has to remember the previously accepted correctly typed value. Clearly, we do not want to show this part of the internal state to the application user. This is achieved using the abstract editor `hidAGEC` (Sect. 4.1), which creates an invisible editor, i.e., a store, for any type.

**Example 5:** Consider the following variation of Example 2:

```plaintext
:: MyRecord a b = { function :: AGEC (a -> b) 
    , argument :: AGEC a 
    , result :: AGEC b }

myEditor = selfGEC "test" guiApply (initval ((+) 1.0) 0.0)
where
  initval f v = { function = dynamicAGEC f 
    , argument = dynamicAGEC v 
    , result = displayAGEC (f v) }
  guiApply myrec=:{ function = af, argument = av }
    = {myrec & result = displayAGEC ((**af) (**av))}
```

The editor above can be used to test functions of a certain statically determined type. Due to the particular choice of the initial values `((+) 1.0 :: Real -> Real and 0.0 :: Real)`, the editor can only be used to test functions of type `Real -> Real` applied to arguments of type `Real`. Notice that it is now statically guaranteed that the provided dynamics are correctly typed. The `dynamicAGEC`-editors take care of the required checks at run-time and they reject ill-typed expressions. The programmer therefore does not have to perform any checks anymore. The abstract `dynamicAGEC`-editor delivers a value of the proper type just like any other abstract editor.

The code in the above example is not only simple and elegant, but it is also very flexible. The `dynamicAGEC` abstract editor can be replaced by any other abstract editor, provided that the statically derived type constraints (concerning `f` and `v`) are met. This is illustrated by the next example.

**Example 6:** If one prefers a counter as input editor for the argument value, one only has to replace `dynamicAGEC` by `counterAGEC` in the definition of `initval`:

\(^{10}\) There is currently no feedback on why the type is rejected. Generating good error messages as in [15] certainly improves the user interface.
The `dynamicAGEC` is typically used when expression editors are preferred over value editors of a type, and when application users need to be able to enter functions of a statically fixed monomorphic type.

One can create an editor for any higher-order data structure $\tau$, even if it contains polymorphic functions. It is required that all higher-order parts of $\tau$ are abstracted, by wrapping them with an $AGEC$ type. Basically, this means that each part of $\tau$ of the form $a \rightarrow b$ must be changed into $AGEC (a \rightarrow b)$. For the resulting type $\tau'$ an edit dialog can be automatically created, e.g., by applying `selfGEC`. However, the initial value that is passed to `selfGEC` must be monomorphic, as usual for any instantiation of a generic function. Therefore, editors for polymorphic types cannot be created automatically using this statically typed generic technique. As explained in Sect. 3.2 polymorphic types can be handled with dynamic type checking.

5 Applications of Higher-Order $GEC$s

The ability to generate editors for higher-order data structures greatly enhances the applicability of $GEC$s. Firstly, it becomes possible to create applications in which functions can be edited as part of a complex data structure. Secondly, these functions can be composed dynamically from earlier created compiled functions on disk. Both are particular useful for rapid prototyping purposes, as they can add real-life functionality.

In this section we discuss one small and one somewhat larger application. Even the code for the latter application is still rather small (just a few pages). The code is omitted in this paper due to space limitations, but it can be found at http://www.cs.kun.nl/~clean/gec. Screen shots of the running applications are given in Appendix A.

An Adaptable Calculator. In the first example we use $GEC$ to create a 'more or less' standard calculator. The default look of the calculator was adapted using the aforementioned $AGEC$ customization techniques. Special about this calculator is that its functionality can be easily extended at run-time: the application user can add his or her own buttons with a user defined functionality. In addition to the calculator editor, a $GEC$ editor is created, which enables the application user to maintain a list of button definitions consisting of button names with corresponding functions. Since the type of the calculator functions are statically known, a statically typed higher-order $GEC$ is used in this example. The user can enter a new function definition using a lambda expression, but it is also possible to open and use an earlier created function from disk. Each time the list is changed with the list editor, the calculator editor is updated and adjusted accordingly. For a typical screen shot see Fig. 1.
A Form Editor. In the previous example we have shown that one can use one editor to change the look and functionality of another. This principle is also used in a more serious example: the form editor. The form editor is an editor with which electronic forms can be defined and changed. This is achieved using a meta-description of a form. This meta-description is itself a data structure, and therefore, we can generate an editor for it. One can regard a form as a dedicated spreadsheet, and with the form editor one can define the actual shape and functionality of such a spreadsheet. With the form editor one can create and edit fields. Each field can be used for a certain purpose. It can be used to show a string, it can be used as editor for a value of a certain basic type, it can be used to display a field in a certain way by assigning an abstract editor to it (e.g., a counter or a calculator), and it can be used to calculate and show new values depending on the contents of other fields. For this purpose, the application user has to be able to define functions that have the contents of other fields as arguments. The form editor uses a mixed mode strategy. The contents of some fields can be statically determined (e.g., a field for editing an integer value). But the form editor can only dynamically check whether the argument fields of a specified function are indeed of the right type. The output of the form editor is used to create the actual form in another editor which is part of the same application. By filling in the form fields with the actual value, the application user can test whether the corresponding form behaves as intended. For a typical screen shot see Fig. 2.

6 Related Work

In the previous sections we have shown that we can create editors that can deal with higher order data structures. We can create dynamically typed higher-order editors, which have the advantages that we can deal with polymorphic higher order data structures and overloading. This has the disadvantage that the programmer has to check type safety in the editor. The compiler can ensure type correctness of higher-order data structures in statically typed editors, but they can only edit monomorphic types. Related work can be sought in three areas:

Grammars Instead of Types: Taking a different perspective on the type-directed nature of our approach, one can argue that it is also possible to obtain editors by starting from a grammar specification instead of a type. Such toolkits require a grammar as input and yield an editor GUI as result. Projects in this flavor are for instance the recent Proxima project [21], which relies on XML and its DTD (Document Type Definition language), and the Asf+Sdf Meta-Environment [10] which uses an Asf syntax specification and Sdf semantics specification. The major difference with such an approach is that these systems need both a grammar and some kind of interpreter. In our system higher-order elements are immediately available as a functional value that can be applied and passed to other components.
GUI Programming Toolkits: From the abstract nature of the GEC toolkit it is clear that we need to look at GUI toolkits that also offer a high level of abstraction. Most GUI toolkits are concerned with the low level management of widgets in an imperative style. One well-known example of an abstract, compositional GUI toolkit based on a combinator library is Fudgets [11]. These combinators are required for plumbing when building complex GUI structures from simpler ones. In our system far less plumbing is needed. Most work is done automatically by the generic function $g_{GEC}$. The only plumbing needed in our system is for combining the $GEC$-editors themselves. Furthermore, the Fudget system does not provide support for editing function values or expressions.

Because a $GEC_t$ is a $t$-stateful object, it makes sense to have a look at object oriented approaches. The power of abstraction and composition in our functional framework is similar to mixins [13] in object oriented languages. One can imagine an OO GUI library based on compositional and abstract mixins in order to obtain a similar toolkit. Still, such a system lacks higher-order data structures.

Visual Programming Languages: Due to the extension of the GEC programming toolkit with higher-order data structures, visual programming languages have come within reach as application domain. One interesting example is the Vital system [14] in which Haskell-like scripts can be edited. Both systems allow direct manipulation of expressions and custom types, allow customization of views, and have guarded data types (like the self$GEC$ function). In contrast with the Vital system, which is a dedicated system and has been implemented in Java, our system is a general purpose toolkit. We could use our toolkit to construct a visual environment in the spirit of Vital.

7 Conclusions

With the original $GEC$-toolkit one can construct GUI applications without much programming effort. This is done on a high level of abstraction, in a fully compositional manner, and type-directed. It can be used for any monomorphic first-order data type. In this paper we have shown how the programming toolkit can be extended in such a way that $GEC$s can be created for higher-order data structures. We have presented two methods, each with its own advantage and disadvantage.

We can create an editor for higher-order data structures using dynamic typing, which has as advantage that it can deal with polymorphism and overloading, but with as disadvantage that the programmer has to ensure type safety at run-time. We can create a editor for higher-order data structures using the static typing such that type correctness of entered expressions or functions is guaranteed at compile-time. In that case we can only cope with monomorphic types, but we can generate type-directed GUIs automatically.

As a result, applications constructed with this toolkit can manipulate the same set of data types as modern functional programming languages can. The system is type-directed and type safe, as well as the GUI applications that are constructed with it.
References


A Screen Shots of Example Applications

![A screen shot of the adaptable calculator. Left the editor for defining button names with the corresponding function definitions. Right the resulting calculator editor](image-url)
Fig. 2. A screen shot of the form editor. The form editor itself is shown in the upper left window, the corresponding editable spreadsheet-like form is shown in the other