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@cs.kun.nl, marko@cs.kun.nl, rinus@cs.kun.nl, 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 GUIs in an abstract and compositional way, using type-directed Graphical Editor Components (*GEC*s). 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 types.* 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 (GUIs) 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 GUIs 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 [19]) 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 $GEC_{\texttt{t}}$ is a graphical editor for values of any *monomorphic first-order* type `t`. This type-directed creation of *GEC*s has been obtained by *generic programming* techniques [8, 16, 15]. 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.

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, 18], 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 types. 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 requires 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* [21], which provides type checking at the command line and can use compiled functions from disk. Esther makes extensive use of dynamic types, which turn arbitrary (higher-order) types into *first-order* types 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 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 [17] 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 *GEC*s in more detail, we need to point out that Clean uses an explicit multiple environment passing style [2] for I/O programming. As *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 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 $GEC_{\mathtt{t}}$ is an editor for values of type `t`. It is generated with a *generic* function [15, 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 data type. This makes the approach particularly suited for *rapid prototyping*.

The generic function `gGEC` creates *GEC*s. It 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. The environment `env` is in this case (`PSt ps`), since `gGEC` uses Object I/O.

```
generic¹ gGEC t :: (GECDef t (PSt ps)) (PSt ps)
                                    → (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 must be created. The second is the initial value of type `t` of the editor. The third is a callback function of type `t → env → 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.

[1] **generic** $f\ t :: T(t)$ introduces a generic function f with type scheme $T(t)$.
[2] Clean separates function arguments by whitespace, instead of `->`.

```
::[3] GECDef t env :==[4] (String,t,CallBackFunction t env)
::  CallBackFunction t env :== t → env → env
```

The (`GECInterface t env`) is a record that contains all *methods* of the newly created $GEC_{\texttt{t}}$.

```
:: GECInterface t env = { gecGetValue :: env → (t,env)
                        , gecSetValue :: t → env → env }[5]
```

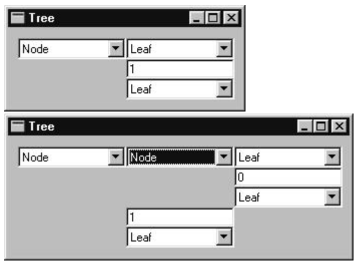
The `gecGetValue` method returns the current value, and `gecSetValue` sets the current value of the associated $GEC_{\texttt{t}}$ object. Programs can be constructed combining editors by tying together the various `gecSetValues` and `gecGetValues`. We are working on an arrow combinator library that abstracts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

```
selfGEC :: String (t → t) t (PSt ps) → (PSt ps) |[6] gGEC{|*|} t
selfGEC s f v env = env1
where ({gecSetValue},env1) = gGEC{|*|} (s,f v,λx → gecSetValue (f x)) env
```

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

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

```
module Editor
import StdEnv, StdIO, StdGEC

Start :: *World → *World
Start world = startIO MDI Void myEditor world

myEditor = selfGEC "Tree" balance (Node Leaf 1 Leaf)

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



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

[3] Type definitions are preceded by `::`.
[4] `:==` introduces a synonym type.
[5] $\{f_0$ `::` t_0`,` … `,` f_n `::` $t_n\}$ denotes a record with field names f_i and types t_i.
[6] In a function type, `|` introduces all overloading class restrictions.

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` is derived.

The tree example shows that a *GEC*$_{\texttt{t}}$ explicitly reflects the structure of type `t`. For the creation of GUI applications, we need to model both specific GUI elements (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 represent 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., .
:: 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 *GEC*s 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 [21] (Sect. 3.1).

Esther enables the user to enter expressions (using a subset of Clean) that are dynamically typed, and transformed to 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, 18, 22] 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) type into a first-order type, 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 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).

```
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.

```
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 [18].

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.

```
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.

```
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** $v :: \tau$ and returned by composeDynamic. In other words, **if** $env \vdash expr :: \tau$ and $[\![expr]\!]_{env} = v$ **then** composeDynamic *expr env* $= (v :: \tau,\ env)$. 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 $GEC_{\texttt{DynString}}$) 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
   ♯[7](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
         ♯ (nexpr,env) = stringGetValue env
         ♯ (ndyn, env) = composeDynamic nexpr env
         = (DynStr ndyn nexpr,env)
      stringUpdate dynStringUpdate nexpr env
         ♯ (ndyn,env) = composeDynamic nexpr env
         = dynStringUpdate (DynStr ndyn nexpr) env
```

The created $GEC_{\texttt{DynString}}$ displays a box for entering a string by calling the standard generic `gGEC{|⋆|}` function for the value `expr` of type `String`, yielding a `stringGEC`. 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`.

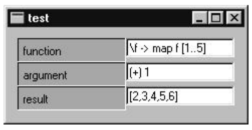
[7] This is Clean's 'do-notation' for environment passing.

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 $v :: \tau$ and $[\![expr]\!] = v$, then (`DynStr` $(v{::}\tau)$ *expr*) :: `DynString`.

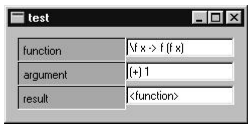
```
mkDynStr :: a → DynString | TC a
mkDynStr x = let dx = dynamic x in DynStr dx (showValueDynamic dx)
```

Example 2: We construct an interactive editor that can be used to test functions. It can be a newly defined function, say `λx → 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.

```
:: 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 v) }
   guiApply  r=:⁸{ function = DynStr (f::a → b) _
                 , argument = DynStr (v::a)     _ }
               = {r &⁹ result = mkDynStr (f v)}
   guiApply  r = r
```


test

function	\f -> map f [1..5]
argument	(+) 1
result	[2,3,4,5,6]


test

function	\f x -> f (f x)
argument	(+) 1
result	<function>

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 `λf x → f (f x)`, and as argument the increment function `((+) 1)`. Then the result is also a function `λx → ((+) 1) ((+) 1 x)`. The editor displays `<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.

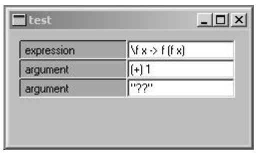
[8] x `=:` e binds x to e.

[9] {r `&` f_0`=`v_0`, ..., `f_n`=`v_n} is a record equal to r, except that fields f_i have value v_i.

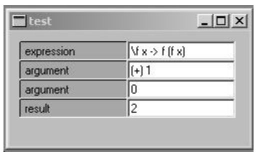
To have a properly higher-order dynamic application example one needs an editor in which the user can type in functions of arbitrary arity, and subsequently enter arguments for this function. The result is then treated such that, if it is a function, editors are added dynamically for the appropriate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

```
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)
```


test

expression	\f x -> f (f x)
argument	(+) 1
argument	"??"


test

expression	\f x -> f (f x)
argument	(+) 1
argument	0
result	2

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 the resulting type is a plain value, it is evaluated and shown using the predefined function `display`, which creates a non-editable editor that just displays its value. As soon as a type mismatch is detected, a question mark is displayed to prompt the user to try again. With this editor, any higher-order polymorphic function can be entered and tested.

4 Statically Typed Higher-order *GEC*s

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 *GEC*s. This gives us a second solution for higher-order types 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 GEC*s (*AGEC*) [7] have been introduced. An instance of `gGEC` for *AGEC* has been defined. Therefore, an $AGEC_{\mathtt{d}}$ can be used as a $GEC_{\mathtt{d}}$, i.e., it behaves as an editor for values of a certain *domain*, say of type `d`. However, an $AGEC_{\mathtt{d}}$ never displays nor edits values of type `d`, but rather a *view* on values of this type. Values of type `v` are shown and edited, and internally converted to the values of domain `d`. The view, say of type `v`, is again generated automatically as a $GEC_{\mathtt{v}}$. 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 $AGEC_{\mathtt{d}}$ with a view of type `v`. The specification is only in terms of the involved data domains. The complete interface to *AGEC*s is given below.

```
:: AGEC d                                   // abstract data type
mkAGEC        :: (ViewGEC d v) → AGEC d | gGEC{|*|} 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. *AGEC*s allow us to define arbitrarily many editors $gec_{\mathtt{i}}$ :: $AGEC_{\mathtt{d}}$ that have a private implementation of type $GEC_{\mathtt{v_i}}$. Because *AGEC* is abstract, code that has been written for editors that manipulates some type containing $AGEC_{\mathtt{d}}$, does not change when the value of type $AGEC_{\mathtt{d}}$ is exchanged for another $AGEC_{\mathtt{d}}$. 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:

```
vertlistAGEC :: [a] → AGEC [a] | gGEC{|★|} a // all elements displayed in a column
counterAGEC  ::  a  → AGEC  a  | gGEC{|★|}, IncDec a // a special number editor
hidAGEC      ::  a  → AGEC  a                 // identity, no editor
displayAGEC  ::  a  → AGEC  a  | gGEC{|★|} a // identity, non-editable editor
```

The *counter* editor below is a typical member of this library.

```
counterAGEC :: a → AGEC a | gGEC{|★|}, IncDec a
counterAGEC j = mkAGEC { d_val = j, d_oldv_to_v = λi _ → (i,Neutral)
                       , update_v = updateCounter, v_to_d = fst      }
where updateCounter (n,UpPressed)   = (n+one,Neutral)
      updateCounter (n,DownPressed) = (n-one,Neutral)
      updateCounter (n,Neutral)     = (  n  ,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 `updateCounter` function is used to synchronize then 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 *AGEC*s is used to build type-directed editors for higher-order types, which check the type of the entered expressions dynamically. These statically typed higher-order editors are created using the function `dynamicAGEC`. The full definition of this function is specified and explained below.

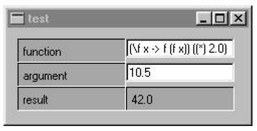
```
dynamicAGEC :: d → AGEC d | TC d
dynamicAGEC x = mkAGEC { d_val=x           , d_oldv_to_v=toView
                       , update_v=updView x, v_to_d=fromView x  }
where toView newx Nothing  = let dx = mkDynStr newx in (dx,hidAGEC dx)
      toView _ (Just oldx) = oldx

      fromView :: d (DynString,AGEC DynString) → d | TC d
      fromView _ (_,oldx) = case ^^oldx of DynStr (x::d^) _ → x

      updView :: d (DynString,AGEC DynString)
              → (DynString,AGEC DynString) | TC d
      updView _ (newx=:(DynStr (x::d^) _),_) = (newx,hidAGEC newx)
      updView _ (_,oldx)                     = (^^oldx,oldx)
```

The abstract `Dynamic` editor, which is the result of the function `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 and the previous value is restored and shown. To do this, the abstract editor has to remember in its internal state also 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:

```
:: 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))}
```



test

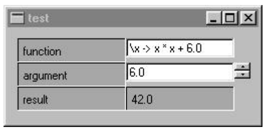
function	(\f x -> f (f x)) ((*) 2.0)
argument	10.5
result	42.0

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`:

```
   initval f v = { function = dynamicAGEC f
                 , argument = counterAGEC v
                 , result   = displayAGEC (f v) }
```



test

function	\x -> x * x + 6.0
argument	6.0
result	42.0

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 type τ, even if it contains polymorphic functions. It is required that all higher-order parts of τ are abstracted, by wrapping them with an *AGEC* type. Basically, this means that each part of τ of the form `a → b` must be changed into `AGEC (a → b)`. For the resulting type τ' 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. 3.2polymorphic types can be handled with dynamic type checking.

5 Applications of higher-order *GEC*s

The ability to generate editors for higher-order types 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 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, 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. Therefore, 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 types. We can create dynamically typed higher-order editors, which have the advantages that we can deal with polymorphic higher order types 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 types 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 obtained 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 [20], 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 `gGEC`. The only plumbing needed in our system is for combining the *GEC*-editors themselves. Any application will only have a very limited number of *GEC*-editors. Furthermore, the Fudget system does not provide support for editing function values or expressions.

Because a $GEC_{\mathtt{t}}$ 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 *mixin*s [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 types, *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 (the `selfGEC` 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. The method could 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 types. We have presented two methods, each with its own advantage and disadvantage.

We can create a editor for higher-order data types using the 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 runtime. We can create a editor for higher-order data types 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.

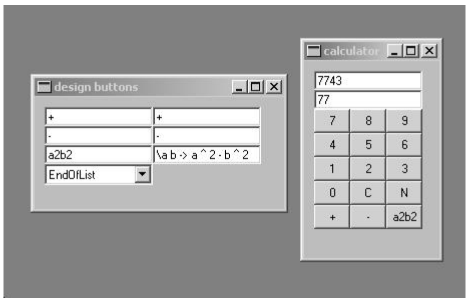
As a result, applications constructed with this toolkit can manipulate the same set of data types as modern functional programming languages can. The system is type-directed and type safe, as well as the GUI applications that are constructed with it.

References

1. M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, and D. Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.
2. P. Achten. *Interactive Functional Programs - models, methods, and implementations*. PhD thesis, University of Nijmegen, The Netherlands, 1996.
3. P. Achten and S. Peyton Jones. Porting the Clean Object I/O library to Haskell. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on the Implementation of Functional Languages, IFL'00, Selected Papers*, volume 2011 of *LNCS*, pages 194–213. Aachen, Germany, Springer, Sept. 2001.
4. P. Achten and R. Plasmeijer. Interactive Functional Objects in Clean. In C. Clack, K. Hammond, and T. Davie, editors, *Proc. of 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, Sept. 1998.
5. Achten, Peter and van Eekelen, Marko and Plasmeijer, Rinus and van Weelden, Arjen. Arrows for Generic Graphical Editor Components. 2004. Under Submission; available as Nijmegen Technical Report NIII-R0416, http://www.niii.kun.nl/research/reports/full/NIII-R0416.pdf.
6. Achten, Peter, van Eekelen, Marko and Plasmeijer, Rinus. Generic 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: draft version available via ftp://ftp.cs.kun.nl/pub/Clean/papers/2004/achp2004-GenericGUI.pdf.
7. Achten, Peter, van Eekelen, Marko and Plasmeijer, Rinus. Compositional Model-Views with Generic Graphical User Interfaces. In *Practical Aspects of Declarative Programming, PADL04*, LNCS. Springer, 2004. To appear; draft version available as Nijmegen Technical Report NIII-R0408, http://www.niii.kun.nl/research/reports/full/NIII-R0408.pdf.
8. 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.
9. E. Barendsen and S. Smeters. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. World scientific, 1999.
10. M. v. d. Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC'01)*, pages 365–370. Springer-Verlag, 2001.
11. 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.
12. 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.
13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98)*, pages 171–183, San Diego, California, 1998. ACM, New York, NY.

14. K. Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
15. R. Hinze. 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.
16. R. Hinze and S. Peyton Jones. Derivable Type Classes. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *ENTCS*. Montreal, Canada, Elsevier Science, 2001.
17. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
18. M. Pil. Dynamic types and type dependent functions. In D. Hammond and Clack, editors, *Implementation of Functional Languages (IFL '98)*, LNCS, pages 169–185. Springer Verlag, 1999.
19. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/∼clean/contents/contents.html.
20. M. Schrage. *Proxima, a presentation-oriented XML editor*. PhD thesis, University of Utrecht, 2004 (to appear).
21. 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; draft version available via ftp://ftp.cs.kun.nl/pub/Clean/papers/2004/vWeA2004-Esther.pdf.
22. 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.

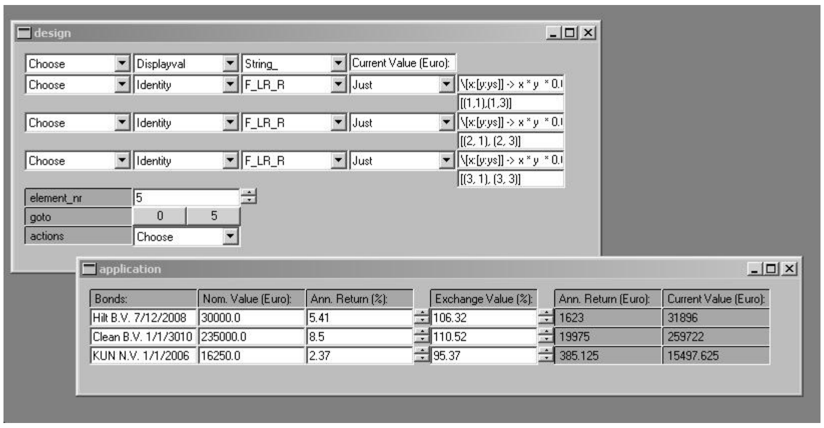
A Screen Shots of Example Applications



+	+
-	-
a2b2	\a b -> a ^ 2 - b ^ 2
EndOfList	

7743		
77		
7	8	9
4	5	6
1	2	3
0	C	N
+	-	a2b2

Fig. 1. A screen shot of the adaptable calculator. Left the editor for defining button names with the corresponding function definitions. Right the resulting calculator editor.



Choose	Displayval	String_	Current Value (Euro):	
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(1,1),(1,3)]
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(2, 1), (2, 3)]
Choose	Identity	F_LR_R	Just	\[x:[y:ys]] -> x * y * 0.(
				[(3, 1), (3, 3)]

element_nr	5	
goto	0	5
actions	Choose	

Bonds:	Nom. Value (Euro):	Ann. Return (%):	Exchange Value (%):	Ann. Return (Euro):	Current Value (Euro):
Hilt B.V. 7/12/2008	30000.0	5.41	106.32	1623	31896
Clean B.V. 1/1/3010	235000.0	8.5	110.52	19975	259722
KUN N.V. 1/1/2006	16250.0	2.37	95.37	385.125	15497.625

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.