GEC: a toolkit for Generic Rapid Prototyping of Type Safe Interactive Applications

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. GUI programming is notoriously tedious. By using generic functions that create Graphical Editor Components (GECs), it becomes possible to define user interfaces without any knowledge of low level I/O handling. A GEC editor can automatically be derived for values of any (user-defined) monomorphic (first order) type. With an editor the application user can create new values of the type the editor is created for.

In this way one obtains an editor for free for almost any type. Such a free editor may not look fancy, but one can change the default look by defining specialized versions of the generic function for certain types, e.g. representing buttons, pull-down menus and the like. Furthermore, with GECs the programmer can create an abstraction level separating the view type and the domain type completely. As a result, the programmer can easily make a library of suited graphical representations and freely choose which representation to use. Consequently, working with GECs a programmer can focus on the data type representing the user interaction instead of on the nasty graphical details.

Editors can be easily combined to ensure that a change in one editor has effects on the contents of others. One can combine editors by hand or use an arrow library of editor combinators.

It is even possible to create editors for higher order types which enables the creation of user interfaces in which functions can be typed in by the user or read from disk at run-time. In the latter case, functions are actually compiled functions that are dynamically linked into the running application.

GECs are suited for rapid prototyping of real world applications, for teaching and for debugging. This paper focuses on the use of the GEC toolkit for functional programmers, only briefly explaining its inner workings and underlying principles.

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 [22]) 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 $GEC_{\mathtt{t}}$ 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 ($\rightarrow$), 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.

The first order *GEC* toolkit is extended in two ways, such that it can construct higher-order value editors. The first extension uses run-time *dynamic typing* [1, 21], 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].

This paper is structured as follows. A brief introduction to generic programming is given in Sect. 2. Section 3 contains an overview of the first-order *GEC* toolkit. In Sect. 4 it is presented how *GEC*s can be composed to form larger applications. An abstraction for model-view programming is introduced in Sect. 5. Extensions for working with higher order types, dynamically and statically are given in Sect. 6. Related work is discussed in Sect. 7 and conclusions are summarized in Sect. 8.

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 [20] 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 Generic Programming

Recently, *generic functions* have been added to Clean [15, 8]. A generic function is an ultimate reusable function that allows reflection on the structure of any data in a type safe way. It is not a single function, but actually a special kind of overloaded function. One might be tempted by the idea to design some suitable universal type, and define generic functions on values of this type, and convert the changed value back to the actual type. However, it is a fundamental property of the language that (without some kind of reflection) there cannot exist a single universal type with which values of any type can be represented.

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

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

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

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

With a collection of generic types, values of any user-defined type can be represented, e.g. [1]::[`Int`] is represented by:

Once defined by the programmer, a generic function can be applied to values of any concrete (user defined) type. The compiler automatically adds conversion functions (*bimaps*) that lazily transform the concrete type to the corresponding

[1] Type definitions are preceded by `::`.

```
TypeCons listT
  (Left (DataCons consD
    (Pair
      1
      (TypeCons listT
        (Right (DataCons nilD Unit))
      )
    )
  ))
with
  listT = {name="_List"²(,arity=1}³
  consD = {name="_Cons",arity=2}
  nilD  = {name="_Nil", arity=0}
```

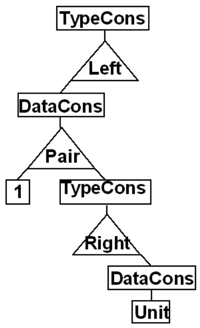


Fig. 1. The generic representation of [1].

combination of generic types (see [16]. Furthermore, the generic types returned by the generic function are converted back again to the actual type demanded.

Generic functions are very useful for defining work of a general nature. Because generic functions can be specialized for any specific concrete type as well, they can also be customized easily. So far, the technique has been successfully used to define functions like equality, map, foldr, as well as for the construction of various parsers and pretty printers. Also, generic programming techniques play an important role in the implementation of automatic test systems [18]. The *GEC* toolkit project is to our knowledge the first project in which generic programming techniques are used for the creation of GUI applications.

3 The basic *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

[2] `_List`, `_Cons`, `_Nil`, are the internal representations of the list-type and its constructors.

[3] $\{f_0 = v_0, \ldots, f_n = v_n\}$ denotes a record with field names f_i and values v_i.

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. 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 standard appearance of a *GEC* is illustrated by the following example:

```
module Editor
import StdEnv, StdIO, StdGEC

Start :: *World → *World[4]
Start world = startIO MDI Void[5] myEditor world

myEditor = generateEditor ("List",[1])

generateEditor :: (String, a) (PSt ps) → PSt ps[6] |[7] gGEC{|*|} a
generateEditor (windowName,initialValue) env = newenv
where
    (gecInterface, newenv) = gGEC{|*|} (windowName, initialValue, const id) env
```

The corresponding GUI will be:

Fig. 2. Generated editor for the standard list type, initially with value [1]

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.

```
generic[8] gGEC t :: GECFunction t (PSt ps)

:: GECFunction t env :==[9] (GECDef t env) env → (GECInterface t env,env)
```

[4] This function is equivalent with Haskell `main::IO ()`.
[5] `Void` is equivalent with Haskell `()`.
[6] Clean separates function arguments by whitespace, instead of `->`.
[7] In a function type, `|` introduces all overloading class restrictions.
[8] **generic** $f\ t :: T(t)$ introduces a generic function f with type scheme $T(t)$.
[9] `:==` introduces a synonym type.

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 by the application user.

```
:: GECDef t env :== (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 $GEC_{\texttt{t}}$.

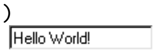
```
:: GECInterface t env = { gecGetValue :: env → (t,env)
                        , gecSetValue :: Update t env → env }
:: Update              = YesUpdate | NoUpdate
```

The `gecGetValue` method returns the current value, and `gecSetValue` sets the current value of the associated $GEC_{\texttt{t}}$ object. The `gecSetValue` method has an argument of type `Update` indicating whether or not the call-back function has to be called propagating the change of the value through the system.

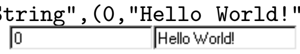
Alternative definition of `myEditor`: **Corresponding GUI:**

```
myEditor2
= generateEditor ("Integer",0)
```


```
myEditor3
= generateEditor ("String","Hello World!")
```




```
myEditor4
= generateEditor ("Tuple of Integer and String",(0,"Hello World!"))
```




```
myEditor5
 = generateEditor ("Tree of Integers",Node Leaf 1 Leaf)

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

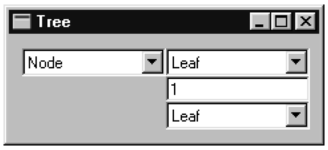


Fig. 3. Automatically derived editors for standard types.

In Fig. 3 the basic use of the function `gGEC` is illustrated by showing the corresponding GUI for several alternative definitions of `myEditor` (as in the example above). This generate an editor for the argument data type. All you have to specify is the name of the window and an initial value. On the right the editor is shown (with only for the `Tree` type also the complete window in which it resides).

For standard types a version of `gGEC` is derived automatically. For user-defined types it is required that explicitly a version of `gGEC` is derived for the given type. For the type `Tree` this is explicitly done in the example. In the rest of this paper these **derive**s are not shown.

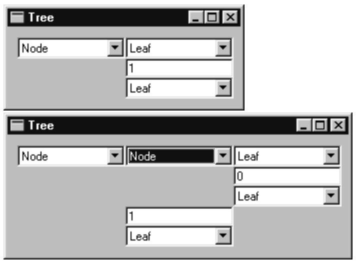
Programs can be constructed combining editors by tying together the various `gecSetValue`s and `gecGetValue`s. In Sect. 4.2 it is shown how an arrow combinator library [5] can be used for the necessary plumbing. For combining editors "by hand" the following function can be very useful (its implementation is discussed in Section 4.1):

`selfGEC :: String (t → t) t (PSt ps) → (PSt ps) | gGEC{|*|}`[10] `t & bimap{|*|}`[11] `ps`

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`.

An example of the use of `selfGEC` is given by the following program that creates an editor for a *self-balancing* binary tree:

```
myEditor = selfGEC  "Tree"
                    balanceTree
                    (Node Leaf 1 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 `balanceTree` is applied. `balanceTree` 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.

[10] Use the generic instance of kind $\star$ of `gGEC`.

[11] The generic `gGEC` function requires an instantiation of the predefined generic function `bimap`.

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.

Customizing Types Clean allows generic functions to be overruled by custom definitions for arbitrary types. `gGEC` is no exception to this rule. The left screenshot in Fig. 4 shows the default interface of the definition below for the ubiquitous *counter* example, when created by:

```
myEditor = selfGEC "Counter" updCntr (0,Neutral)

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

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

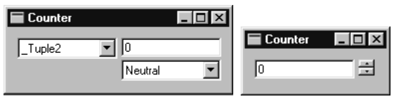


Fig. 4. The default editor (left) and the customized editor (right) of the counter example.

Although the definition of the counter is a sensible one, its visual interface clearly is not. In [6] we show how to change the representation of all values of type `Counter` to the screenshot shown at the right in Fig. 4. Because it has been explained in detail in [6], we will not repeat the code, but point out the important points:

- In this particular example, only the definitions of (,) (hide the constructor and place its arguments next to each other) and `UpDown` (display  instead of ) need to be changed.
- Normally `gGEC` creates the required logical (value passing) and visual infrastructure (GUI components). The programmer, when customizing `gGEC`, only needs to define the visual infrastructure. The programmer must be knowledgeable about Object I/O programming.
- The overruled instance works not only at the top-level. Every nested occurrence of the `Counter` type is now represented as shown right in Fig. 4.

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). In a way similar to the one shown above for the spin button, this has also been done by specializing `gGEC` for a number of other types that either represent GUI elements or layout. Below the predefined specialized editors are shown for a number of types. The specialized editor for `Display` creates a non-editable GUI; for `Button` a button is created; for `<|>` and ↔ two editors are created below each other, respectively next to each other; and finally `Hide` creates no GUI at all which is useful for remembering state.

Type of the value given to `myEditor`:	**Corresponding GUI:**
`:: Display a = Display a`	
`:: Button = Button String` `         | Pressed`	
`:: <|> a b = a <|> b`	
`:: <-> a b = a <-> b`	
`:: Hide a = Hide a`	

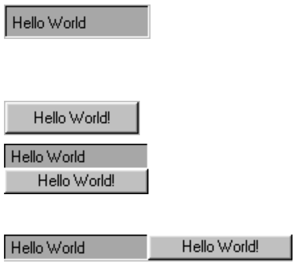


Fig. 5. Effect of some predefined customized editors on "Hello World!".

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

4 Composition of GECs

In this section we present a number of examples to show how *GEC*s can be combined relying on the callback mechanism and method invocation (Sect. 4.1). In Sect. 4.2 we show how these examples can be expressed using arrow combinators.

4.1 Manual composition of GECs

Functionally dependent GECs. The first composition example establishes a functional dependency of type `a → b` between a source editor $GEC_{\mathtt{a}}$ and destination editor $GEC_{\mathtt{Display\ b}}$:

```
applyGECs :: (String,String) (a → b) a (PSt ps) → PSt ps
                                | gGEC{|*|} a & gGEC{|*|} b & bimap{|*|} ps
applyGECs (sa,sb) f va env
   ♯[12] (gec_b, env) = gGEC{|*|} (sb, Display (f va), const id)  env
```

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

```
  ♯    (gec_a, env) = gGEC{|⋆|} (sa, va, set gec_b f) env
  =    env

set :: (GECInterface b (PSt ps)) (a → b) a (PSt ps) → (PSt ps)
set gec f va env = gec.gecSetValue NoUpdate (Display (f va)) env
```

The callback function of $gec_{\mathtt{a}}$ uses the `gecSetValue` interface method of $gec_{\mathtt{b}}$ to update the current `b` value whenever the user modifies the `a` value. As a simple example, one can construct an interactive editor for lists (see Fig. 6) that are mapped to balanced trees by:

```
myEditor = applyGECs ("List","Balanced Tree") list2balancedTree [1,5,2]
```

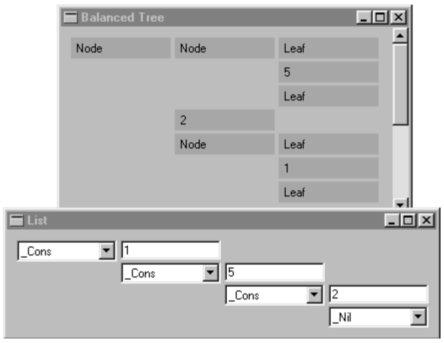


Fig. 6. Turning lists into balanced binary trees.

Of course, the same can be done for binary functions with slightly more effort:

```
apply2GECs :: (String,String,String) (a → b → c) a b (PSt ps) → (PSt ps)
                  | gGEC{|⋆|} a & gGEC{|⋆|} b & gGEC{|⋆|} c & bimap{|⋆|} ps
apply2GECs (sa,sb,sc) f va vb env = env3
where
   (gec_c,env1) = gGEC{|⋆|} (sc,Display (f va vb),const id) env
   (gec_b,env2) = gGEC{|⋆|} (sb,vb,combine gec_a gec_c (flip f)) env1
   (gec_a,env3) = gGEC{|⋆|} (sa,va,combine gec_b gec_c f) env2

combine :: (GECInterface y (PSt ps)) (GECInterface z (PSt ps))
                                             (x → y → z) x (PSt ps) → PSt ps
combine gy gc f x env
   ♯ (y,env) = gy.gecGetValue env
   ♯ env     = gc.gecSetValue NoUpdate (Display (f x y)) env
   = env
```

Notice that, due to the explicit environment passing style, it is trivial in Clean to connect $GEC_{\mathtt{b}}$ with $GEC_{\mathtt{a}}$ and vice versa. In Haskell's monadic I/O one needs to tie the knot with `fixIO`.

As an example, one can construct two interactive list editors, that are merged and put into a balanced tree:

```
myEditor = apply2GECs ("List1","List2","Balanced Tree") makeBalancedTree [1] [1]
where
    makeBalancedTree l1 l2 = list2balancedTree (l1 ++ l2)
```

with `++ :: [a] [a] → [a]` the Clean list concatenation operator. Fig. 7 shows the result.

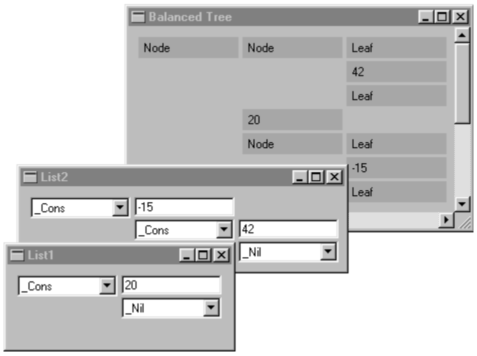


Fig. 7. Merging two lists into a balanced binary tree.

Self-correcting GECs. In this example we give the implementation of the *self-correcting* editor function `gGEC` that was already used in Sect. 3. Self-correcting editors update *themselves* in response to user edit operations. The function definition is concise:

```
selfGEC :: String (a → a) a (PSt ps) → (PSt ps) | gGEC{|*|} a & bimap{|*|} ps
selfGEC s f v env = env1
where (gec,env1) = gGEC{|*|} (s,f v,λx → gec.gecSetValue NoUpdate (f x)) env
```

As an example, one can now construct a *self-sorting* list as follows:

```
myEditor = selfGEC "Self Sorting List" sort [5,1,2]
```

with `sort :: [a] → [a] | Ord a`. This means that it is impossible for a user of this editor to create a stable non-sorted list value.

Mutually Dependent GECs. In a similar way one can define mutually dependent *GEC*s. Take the following definition of `mutualGEC`.

```
mutualGEC :: String a (a → b) (b → a) (PSt ps) → (PSt ps)
                                | gGEC{|*|} a & gGEC{|*|} b & bimap{|*|} ps
mutualGEC gui va a2b b2a env = env2
```

```
where (gec_b,env1) = gGEC{|*|} (gui, a2b va, set gec_a b2a) env
      (gec_a,env2) = gGEC{|*|} (gui,     va, set gec_b a2b) env1
```

This function displays two *GEC*s. It is given an initial value `va` of type `a`, a function `a2b :: a → b`, and a function `b2a :: b → a`. The `gec_a` initially displays `va`, while `gec_b` initially displays `a2b va`. Each time one of the *GEC*s is changed, the other is updated automatically. The order in which changes are made is irrelevant. For example, the application `mutualGEC "Exchange Euros to Pounds"{euros = 3.5} toPounds toEuros` results in an editor that calculates the exchange between pounds and euros (see Fig. 8) and vice versa.

```
exchangerate = 1.4

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

toPounds :: Euros → Pounds
toPounds {euros} = {pounds = euros / exchangerate}

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

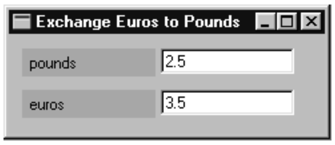


Fig. 8. Mutually dependent $GEC_{\texttt{Pounds}}$ and $GEC_{\texttt{Euros}}$ in one window.

The example of Fig. 8 may look a bit like a tiny spreadsheet, but it is essentially different since standard spreadsheets do not allow mutual dependencies between cells. Notice also the separation of concerns: the way *GEC*s are coupled is defined completely separate from the actual functionality.

4.2 Combinators for GEC composition

The examples in Sect. 4.1 show that *GEC*s can be composed by writing appropriate callback functions that use the `GECInterface` methods `gecGetValue` (get the value of a *GEC*) and `gecSetValue` (set its value). This explicit plumbing can become cumbersome when larger and more complex situations must be specified. What is needed, is a disciplined, and more abstract way of combining components. *Monads* [26] and *arrows* [17] are the main candidates for such a discipline. Monads abstract from computations that produce a value, whereas arrows abstract from computations that, *given certain input*, produce values. Because *GEC*s also have input and produce values, arrows are the best match. In this section we show how arrows can be used successfully for the composition of *GEC*s, resulting in structures that resemble *circuits of GECs* (`GecCircuit a b`).

A `GecCircuit a b` generalizes *GEC*s by accepting input values of type `a` and produces output values of type `b`. Clearly, for every $GEC_{\texttt{a}}$ there exists a `GecCircuit a a`. This relation is expressed concisely with the function `edit`:

```
edit :: String → GecCircuit a a | gGEC{|*|} a
```

The arrow class definition for which we need to provide implementation for our *GEC* arrows of type `GecCircuit` is given below. This class describes the basic combinators ⋙ (serial composition), `arr` (function lifting), and `first` (saving values across computations). The other definitions below can all be derived in the standard way from these basic arrow combinators. They are repeated here because we use them in our examples.

```
class Arrow arr where
    arr   :: (a → b) → arr a b
    (⋙)   :: (arr a b) → (arr b c) → arr a c
    first :: (arr a b) → arr (a,c) (b,c)

/* Combinators for free: */
second :: (arr a b) → arr (c, a) (c, b)
second gec = arr swap ⋙ first gec ⋙ arr swap
where
    swap t = (snd t,fst t)

returnA :: arr a a
returnA = arr id

(⋘) infixr 1 :: (arr b c) (arr a b) → arr a c
(⋘) l r = r ⋙ l

(***) infixr 3 :: (arr a b) (arr c d) → arr (a,c) (b,d)
(***) l r = first l ⋙ second r

(&&&) infixr 3 :: (arr a b) (arr a c) → arr a (b,c)
(&&&) l r = arr (λx → (x,x)) ⋙ (l *** r)
```

Arrow Examples

We use the arrow combinator definitions in the examples below. For each example of Sect. 4.1, we give the definition using arrow combinators, and some of the circuit structures as figures.

Functionally dependent GECs. The first arrow example (of which the external view is given in Fig. 9) shows the arrow combinator version of the `applyGECs` example of Sect. 4.1.

```
myEditor = startCircuit (applyGECs ("List","Balanced Tree")
                                   (Display o[13] list2balancedTree)) [1,5,2]

applyGECs :: (String,String) (a → b) → GecCircuit a b
```

[13] The infix operation `o` is the standard function composition in Clean.

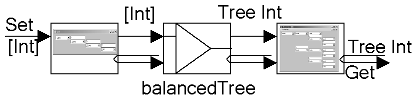


Fig. 9. `applyGECs` using arrows balancing a tree, external view.

```
                                           | gGEC{|*|} a & gGEC{|*|} b
applyGECs (sa, sb) f = edit sa >>> arr f >>> edit sb
```

Again, two visual editors are shown. The first allows the user to edit the (initial) list, and the second shows (and allows the user to edit) the resulting balanced tree. In the hand coded examples, the initial value of a *GEC* was specified at the time of its creation. Using the arrow combinators to construct a `GecCircuit`, we specify the initial values for all *GEC*s when we start the circuit.

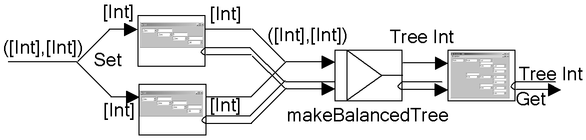


Fig. 10. `apply2GECs` using arrows creating a balanced tree from two lists, external view.

```
myEditor
= startCircuit (apply2GECs ("List1","List2","Balanced Tree") makeBalancedTree) ([1],[2])
where
    makeBalancedTree (l1,l2) = Display (list2balancedTree (l1 ++ l2))

apply2GECs :: (String,String,String) (a → b → c) → GecCircuit a b
                                     | gGEC{|*|} a & gGEC{|*|} b & gGEC{|*|} c
apply2GECs (sa, sb, sc) f = edit sa *** edit sb >>> arr f >>> edit sc
```

The example above (see Fig. 10 for its external view) shows the arrow combinator version of the `apply2GECs` example. The initial values for the input lists are paired, to allow the delayed initialization using `startCircuit`. The example clearly shows that combining *GEC*s using arrow combinators is much more readable than the (often) recursive handwritten functions. The linear flow of information between *GEC*s, using the>>>combinator, corresponds directly with the code. Although splitting points in flow of information, using the *** combinator, is less clear, it is still easier on the eyes than the examples of Sect. 4.1.

Self-correcting GECs. The example below shows the arrow combinator version of the `selfGEC` example (see its external view in Fig. 11).

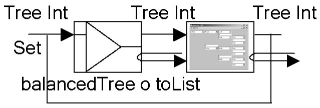


Fig. 11. `selfGEC` using arrows, self balancing a tree, external view.

```
myEditor = startCircuit selfGEC Leaf

selfGEC :: String (a → a) → GecCircuit a a | gGEC{|*|} a
selfGEC s f = feedback (arr f >>> edit s)
```

The way the `feedback` combinator constructs a feedback circuit is by taking the value of the circuit and feeding it back again into the circuit. This is done in such a way that it will not be propagated further when it arrives at a *GEC* editor.

When a feedback circuit contains no editor at all, the meaning of the circuit is undefined since in that case the calculation of the result would depend on itself in a circular way. A feedback circuit in which each path of the circuit contains an editor, is called *well-formed*. It is easy to check syntactically whether feedback circuits are well-formed. Consider the following examples of non well-formed and well-formed feedback circuits.

```
nonWellFormed1 = feedback (arr id >>> arr ((+) 1))
nonWellFormed2 = feedback (arr id &&& edit "Int" >>>
                           arr (λ(x, y) → x + y)    )
wellFormed = feedback (edit "Int" >>> arr ((+) 1))
```

It should be clear that the `selfGEC` function is well-formed. This completes the arrow combinator versions of the examples of Sect. 4.1. The counter example (Sect. 3) below is also conveniently, and concisely, expressed using `arr` and >>>:

```
myEditor = startCircuit (selfGEC "Counter" updCntr) (0,Neutral)
```

As a somewhat larger, and more tantalizing, example we show the basic structure of a *GEC* for *GEC*s below. We use quite complex *GEC*s that allow the user to edit the type and visual appearance of another *GEC*. These editors are not shown because we want to emphasize on *GEC* circuits here, not the internal workings of the editors themselves. The information flow between these editors can, again, nicely be expressed using the arrow combinators.

Both the editor for designing a *GEC*, as well as the editor that displays, and allows the designer to interact with, the designed *GEC* use a well-formed feedback loop. Auxiliary conversion, and state carrying, functions are lifted using the `arr` combinator. Both editors are combined (without feedback) using the ⋙ combinator.

```
myEditor = startCircuit (designEditor   >>>
                         arr convert    >>>
                         applicationEditor) initvalue

designEditor :: GecCircuit DesignEditor DesignEditor
designEditor = feedback (
                  toDesignEditor                    >>>
                  edit "design"                     >>>
                  arr (updateDesign o fromDesignEditor))

applicationEditor :: GecCircuit ApplicationEditor ApplicationEditor
applicationEditor = feedback (
                        arr (toApplicEditor o updateApplication) >>>
                        edit "application"                       >>>
                        arr fromApplicEditor                      )
```

5 Compositional Model-View Programming

When constructing a GUI application, the need arises to incrementally build and modify the GUI under construction. From what is explained in the previous sections, this means that one needs to modify the data structures, and hence also the dependent code, when changing the application. In this section we explain how to obtain a good separation between the *logic* of a GUI application (the *model*) and the way it is presented to the user (the *view*). Put in other words, it is an instance of the *model-view* paradigm [19]. We show that it can be incorporated smoothly within our toolkit by inventing a new way to realize abstraction and composition based on the specialization mechanism that is used by the generic framework of the GEC toolkit. In the following sections we show step by step how this can be accomplished.

First, in Sect. 5.1 we show how to construct self-contained editors that take care of their own update and conversion behavior. In Sect. 5.2 we turn these self-contained editors into *abstract reusable* editors, thus encapsulating all information about their implementation and behaviour. However, *abstract* types seem to be at odds with generic programming. We show in Sect. 5.2 how we have managed to solve this problem. Finally, we show in Sect. 5.3 that these self-contained editors are truly reusable elements themselves.

The technique is illustrated by means of the following running example. The code fragment below shows the *data model*. The data model is a record of type `MyDataModel`. The intention is that whenever the user edits one of the fields `value1` or `value2`, then these new values are summed and displayed in field `sum`. This behavior is defined by `updDataModel`. We want to emphasize that the types and code are ‘carved in stone’: they do not change in the rest of this section.

```
:: MyDataModel
   = { d_value1 :: Int, d_value2 :: Int , d_sum :: Int }

initDataModel :: (Int,Int) → MyDataModel
initDataModel (v1,v2)
   = { d_value1 = v1, d_value2 = v2, d_sum = v1 + v2 }

updDataModel :: MyDataModel → MyDataModel
updDataModel d = { d & d_sum = rec.d_value1 + rec.d_value2 }

myEditor = selfGEC "View on Data"
                   (toMyViewModel o updDataModel o fromMyViewModel)
                   (toMyViewModel (initDataModel (0,0)))
```

The aim of this section is to show that the *view* on this data model can be varied without any modifications to this data model.

5.1 Defining Self-Contained Editors

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

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

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

Next, we define a specialized version of our generic editor `gGEC` for this type. The top-level definition is given below. Notice that in `gGEC{|ViewGEC|}` two additional parameters appear: `gGECd` and `gGECv`. This is caused by the fact that generic functions in Clean are kind-indexed functions. As `ViewGEC d v` is of kind

$\star \to \star \to \star$, the generic function has two additional parameters, one for type `d` and one for type `v`.

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

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

What remains to be defined are the callback function for the view editor (`viewCallback`) and the `GECInterface (ViewGEC d v)` methods (`viewSetValue` and `viewGetValue`). We discuss the most complicated one, the callback function, first. Whenever the application user creates a new `v`-value with this editor, the call-back function of the `v`-editor is called (`viewCallback`) and the `update_v :: v → v` function is applied. This is similar to applying `selfGEC update_v` to the corresponding new value of type `v`. The resulting new `v`-value is shown in the `v`-editor again, and it is converted back to an `d`-value as well, using the function `v_to_d :: v → d`. This new `d`-value is then stored in the `ViewGEC` record in the `d_val` field, and the call-back function for the `ViewGEC` editor is called (`viewGECCallback`).

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

The two interface methods to write (`viewSetValue`) and read (`viewGetValue`) are fairly straightforward. Writing a value of type `ViewGEC d v` amounts to writing a value of type `v` using the current old value of type `v` and the new value of type `d` that is stored in the `ViewGEC` record. Reading a value of type `ViewGEC d v` amounts to reading the current `v` value and wrap it up in the record after converting it to a `d` value.

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

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

```
where
   (current_v,new_env) = vInterface.gecGetValue env
```

The concrete behavior of the generated `ViewGEC` editor now not only depends on the type, but also on the concrete information stored in a value of type `ViewGEC`. A self-contained reusable editor, such as `counterGEC` below, is now quickly constructed. The corresponding editor takes care of the conversions and the update. The `displayGEC` does a trivial update (`id`entity) and also takes care of the required conversions.

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

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

Making use of these new self-contained editors we can attach a *view model* to the *data model* that was presented in the start of this section by giving appropriate definitions of the conversion functions `toMyViewModel` and `fromMyViewModel`. All other definitions remain the same.

```
:: MyViewModel  = { v_value1 :: ViewGEC Int Counter
                  , v_value2 :: ViewGEC Int Counter
                  , v_sum    :: ViewGEC Int (Display Int) }

toMyViewModel :: MyDataModel → MyViewModel
toMyViewModel d = { v_value1 = counterGEC d.d_value1
                  , v_value2 = counterGEC d.d_value2
                  , v_sum    = displayGEC d.d_sum }

fromMyViewModel :: MyViewModel → MyDataModel
fromMyViewModel v = { d_value1 = v.v_value1.d_val
                    , d_value2 = v.v_value2.d_val
                    , d_sum    = v.v_sum.d_val }
```

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

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

5.2 Abstract Self-Contained Editors

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

However, if we manage to hide these types, how can the generic mechanism generate the editor for it? The generic mechanism can only generate an editor for a given concrete type, not for an abstract type of which the content is unknown. The solution is as follows. When the abstraction is being made, we *do* know the contents and its type. Hence, we can store the *generic editor function* (of type `GECFunction`, see Sect.3) in the abstract data structure itself where the abstraction is being made. The stored editor function can be applied later when we really need to construct the editor. Therefore, we define an abstract data structure (`AGEC d`) in which we store the `ViewGEC d v` and its corresponding generic `gGEC` function for `v`. Technically this requires a type system that supports existentially quantified types as well as rank-2 polymorphism.

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

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

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

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

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

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

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

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

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

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

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

Using abstract editors we can refine the *view model* data type and conversion functions:

```
:: MyViewModel    = { v_value1 :: AGEC Int
                    , v_value2 :: AGEC Int
                    , v_sum    :: AGEC Int }

toMyViewModel   :: MyDataModel → MyViewModel
toMyViewModel   d = { v_value1 = counterAGEC d.d_value1
                    , v_value2 = counterAGEC d.d_value2
                    , v_sum    = displayAGEC d.d_sum }

fromMyViewModel :: MyViewModel → MyDataModel
fromMyViewModel v = { d_value1 = ^^ v.v_value1
                    , d_value2 = ^^ v.v_value2
                    , d_sum    = ^^ v.v_sum }
```

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

5.3 Abstract Editors Are Compositional

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

```
sumAGEC :: Int → AGEC Int                          // see counterAGEC (5.2)
sumAGEC i = mkAGEC (sumGEC i)
where sumGEC :: Int → ViewGEC Int MyViewModel  // see counterGEC (5.1)
      sumGEC i = mkViewGEC i toV updV fromV
```

```
where toV   = toMyViewModel o toMyData
      fromV = fromMyData    o fromMyViewModel
      updV  = toMyViewModel o updDataModel o fromMyViewData

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

Now `sumAGEC`, `counterAGEC`, and `displayAGEC` are interchangeable components. If we want to experiment with variants of the running example, we pick the instance of our choice in the `toMyViewModel` function (see Fig. 12).

Alternative definition of `toMyViewModel`: **Corresponding GUI:**

```
toMyViewModel1 d
   = { v_value1 = idAGEC      d.d_value1
     , v_value2 = idAGEC      d.d_value2
     , v_sum    = displayAGEC d.d_sum }

toMyViewModel2 d
   = { v_value1 = idAGEC      d.d_value1
     , v_value2 = counterAGEC d.d_value2
     , v_sum    = displayAGEC d.d_sum }

toMyViewModel3 d
   = { v_value1 = counterAGEC d.d_value1
     , v_value2 = sumAGEC     d.d_value2
     , v_sum    = displayAGEC d.d_sum }
```

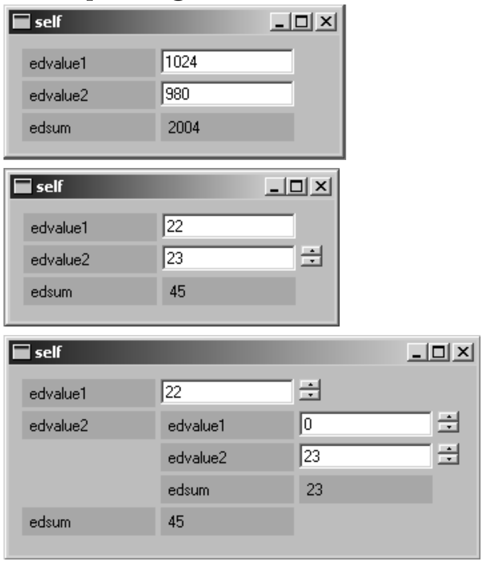


Fig. 12. *Plug-and-play* your favourite abstract editors to experiment with the running example. The only code that changes is the function `toMyViewModel`.

We are setting up a library of abstract components. One of these library functions `idAGEC` (which takes a value of any type and promotes it to an abstract editor component for that type) is used in the example above. With this library it will be possible to rapidly create GUIs in a declarative style. This can be very useful e.g. for prototyping, education, tracing and debugging 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
```

6 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 [24] (Sect. 6.1).

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, 21, 25] of Clean.

The shell uses a text-based interface, and hence it makes sense to create a special *string*-editor (Sect. 6.1), 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. In addition we show that the actual *content* of a dynamic value can be influenced by the very same generic mechanism, using type dependent functions (Sect. 6.2). With this mechanism, dynamics can be used in a type-directed way, but only for monomorphic types in dynamics.

6.1 Dynamically Typed Higher-Order GECs

We first introduce the foundations of the Esther shell, and proceed by showing how to construct an editor for functions.

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. 6.1, 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 [21].

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.

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. 3) 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
   ♯ (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
```

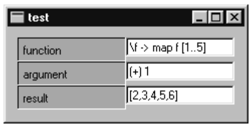
The created $GEC_{\texttt{DynString}}$ displays a box for entering a string by calling the standard generic `gGEC{|⋆|}` function for the value `expr` of type `String`, yielding a `stringGEC`. The `DynString`-editor is completely defined in terms of this `String`-editor. It only has to take care of the 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`.

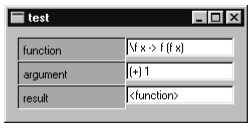
It is convenient to define a constructor function `mkDynStr` that converts any input *expr*, that has value v of type τ, into a value of type `DynString` guaranteeing that if $v :: \tau$ and $[\![expr]\!] = v$, then (`DynStr` $(v{::}\tau)$ *expr*) :: `DynString`.

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

Function Test Example. We construct an interactive editor that can be used to test functions. It can be a newly defined function, say λ`x` → `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=:[14]{ function = DynStr (f::a → b) _
                  , argument = DynStr (v::a)      _ }
                = {r & result = mkDynStr (f v)}
   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. 6.1), 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 λ`f x` → `f (f x)`, and as argument the increment function `((+) 1)`. Then the result is also a function λ`x` → `((+) 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.

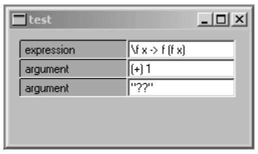
To have a properly higher-order dynamic application example one needs an editor in which the user can type in functions of arbitrary arity, and subsequently enter arguments for this function. The result is then treated such that, if it is a function, editors are added dynamically for the appropriate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

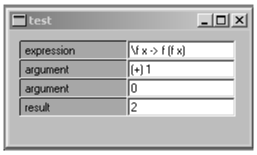
Expression Test Example. We construct a test program that accepts arbitrary expressions and adds the proper number of argument editors, which again can be arbitrary expressions. The number of arguments cannot be statically determined and has to be recalculated each time a new value is provided. Instead

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

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` (Sect. 5.3) and access operator `^^` (Sect. 5.2). 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:args]
       = vertlistAGEC [f:check (fromDynStr f) 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)

    fromDynStr (_,DynStr d _) = d
```





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 requires another argument which is 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.

6.2 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.

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.

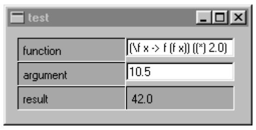
```
dynamicAGEC :: d → AGEC d | TC d
dynamicAGEC x = mkAGEC { d_val=x             , d_oldv_to_v=toView
                       , update_v=updView x, v_to_d=fromView x  }
where toView newx Nothing  = let dx = mkDynStr newx in (dx,hidAGEC dx)
      toView _ (Just oldx) = oldx

      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. 6.1. 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. 6.1). 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. 5.3), which creates an invisible editor, i.e., a store, for any type.

Function Test Example, revisited. Consider the following variation of the function test example on page 25:

```
:: 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))}
```



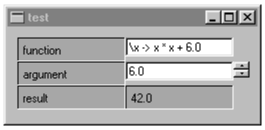
The editor above can be used to test functions of a certain statically determined type. Due to the particular choice of the initial values (`(+) 1.0 :: Real → Real` and `0.0 :: Real`), the editor can only be used to test functions of type

`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.

Function Test Example, revisited. 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) }
```



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 `t`, even if it contains polymorphic functions. It is required that all higher-order parts of `t` are abstracted, by wrapping them with an *AGEC* type. Basically, this means that each part of `t` of the form `a` → `b` must be changed into `AGEC (a` → `b)`. For the resulting type `t'` 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. 6.1 polymorphic types can be handled with dynamic type checking.

Summarizing, we have shown in this section 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.

7 Related Work

We distinguish three areas of related work:

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 [23], 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.

8 Conclusions

We have presented the *GEC* toolkit for rapid prototyping of type safe interactive applications. The toolkit

1. produces type-safe interactive applications composed from *G*raphical *E*ditor *C*omponents;

2. has a high degree of *automation* due to the use of generic generative programming techniques;
3. can be used for first order and *higher order* types;
4. can be *customized* to create any kind of user interface;
5. allows *abstraction* using model-view programming to hide details and allow type-safe view changes;
6. is *compositional* on various levels:
   Types standard composition of types lead to composition of corresponding graphical editor components;
   Expressions the user can enter expressions in which values and functions can be defined/used compositionally; these functions can even be compiled functions (possibly taken from complete applications) that are read from disk, linked in dynamically and applied in a compositional way;
   GECs *GEC*s can be composed in an ad-hoc way by standard functional programming or in a structured way using arrow combinators;
   AGECs AGECs can be composed in a statically type safe way.
7. enables the programmer to *focus on a data type* representing the interaction with the user instead of on the many nasty details of a graphical toolkit.

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*, volume 3057 of *LNCS*, pages 39–55. Springer, 2004.

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. Smetsers. *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. J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.
18. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.
19. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
20. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
21. 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.
22. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/∼clean/contents/contents.html.
23. M. Schrage. *Proxima, a presentation-oriented XML editor*. PhD thesis, University of Utrecht, 2004 (to appear).
24. 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.

25. 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.
26. P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.