Programming Generic Graphical User Interfaces

Peter Achten, Marko v. Eekelen, Rinus Plasmeijer, and Arjen v. Weelden

Institute for Computing and Information Sciences, Radboud University Nijmegen

Keywords: Graphical User Interfaces, Functional Programming, Generic Programming
Submission Category: Full paper
Pages: 12
Corresponding Author: Peter Achten. Toernooiveld 1, 6525ED Nijmegen.
Phone: (+31)(0)24-3652483; Fax: (+31)(0)3652525; E-mail: P.Achten@cs.ru.nl.

Abstract. The GEC Toolkit offers to programmers a high-level, generic style of programming Graphical User Interfaces (GUIs). Programmers are not concerned with low-level widget plumbing. Instead, they use mathematical data models that reflect both the application logic and the visualisation. The data models and the logic are expressed as standard functional style data types and functions over these data types. This significantly brings down the learning effort. In this paper we present an improved programming method of this toolkit and illustrate it by means of a complicated case study: that of a family tree editor. The new programming method brings GUI programming into the reach of every novice functional programmer.

1 Introduction

In this paper we present an improved programming method for the GEC Toolkit [4–7]. The GEC Toolkit is a high level toolkit for the construction of Graphical User Interfaces (GUIs) in terms of mathematical data models and pure functions. Its main features are:

– Automation: for every conceivable data model, a graphical editor component is automatically derived that allows users to edit values of that type.
– Compositional: for free because automation works for all (composite) types.
– Abstract: Programmers do not need to know anything about conventional widget based GUI APIs and their management. Instead, only data models are manipulated with pure functions.

The GEC Toolkit is based on the pure and lazy functional programming language Clean [21, 22]. Functional programming languages such as Clean and Haskell [20], have a sound theoretical foundation: the λ-calculus. One of the main goals of the Clean project has been to demonstrate that the elegance and succinctness of functional programs does not hamper their efficient execution. Contributions of the Clean project in this respect are its strictness analysis,
The key technology on which the GEC Toolkit has been built is generic programming \cite{16,15}. With this technique, the programmer defines a kind-indexed family of functions that have a uniform type scheme. Generic programming has been built in in Generic Haskell \cite{11} and Clean \cite{8}. The main features of this style of generic programming are:

- Only a few function definitions suffice to specify an algorithm for any conceivable custom data type. These function definitions typically correspond with the inductive structural elements of types.
- Besides this minimal number of function definitions, the programmer is allowed to specialize the algorithm for specific types. This feature gives generic programming its flexibility, which we use extensively in this paper.

The GEC Toolkit uses generic programming to automatically create a Graphical Editor Component (GEC) for any conceivable data type \( t \). A GEC is a GUI component that always has a value \( v :: t \), and that can be edited by the user. By editing, we mean any user manipulation of the presented value. This can be keyboard input for strings or numbers, but we also consider button presses to be value-editing actions. Editing is type-safe: the value of a GEC can only be changed in such a way that any new value is of type \( t \).

The generic (kind-indexed family of) function(s) \( \text{gGEC} \) that creates \( \text{GEC} \)'s has type \((\text{GECFunction } t (\text{PSt ps}))\). In Clean, this is declared as follows:

\begin{verbatim}
2 The GEC Toolkit

The key technology on which the GEC Toolkit has been built is generic programming \cite{16,15}. With this technique, the programmer defines a kind-indexed family of functions that have a uniform type scheme. Generic programming has been built in in Generic H\c{e}skell \cite{11} and Clean \cite{8}. The main features of this style of generic programming are:

- Only a few function definitions suffice to specify an algorithm for any conceivable custom data type. These function definitions typically correspond with the inductive structural elements of types.
- Besides this minimal number of function definitions, the programmer is allowed to specialize the algorithm for specific types. This feature gives generic programming its flexibility, which we use extensively in this paper.

The GEC Toolkit uses generic programming to automatically create a Graphical Editor Component (GEC) for any conceivable data type \( t \). A GEC is a GUI component that always has a value \( v :: t \), and that can be edited by the user. By editing, we mean any user manipulation of the presented value. This can be keyboard input for strings or numbers, but we also consider button presses to be value-editing actions. Editing is type-safe: the value of a GEC can only be changed in such a way that any new value is of type \( t \).

The generic (kind-indexed family of) function(s) \( \text{gGEC} \) that creates \( \text{GEC} \)'s has type \((\text{GECFunction } t (\text{PSt ps}))\). In Clean, this is declared as follows:

\end{verbatim}
The type synonym \((\text{GECFunction } t \text{ env})\) is a function that takes two arguments, \(t\) and \(\text{env}\). It creates a GEQ in the environment of type \(\text{env}\). It returns the updated environment, but also the \textit{methods} (of type \((\text{GECMethods } t \text{ env})\)) that a programmer can invoke to obtain access to the GEQ in the environment.

We will not use the GEQ methods in this paper.

\[
:: \text{GECFunction } t \text{ env} := \langle t \text{ env} \rightarrow (\text{GECMethods } t \text{ env}, \text{env}) \rangle
\]

The environment parameter is instantiated with \((\text{PSt } ps)\). This is an Object I/O type that represents the explicit GUI environment that is passed along all GUI callback functions. In pure functional languages, side-effects are modelled by passing environments around, either explicitly as in Clean, or implicitly as for instance \textit{state monads} [19] do in Haskell.

gGEC is a generic function, and hence it can create a GEQ for any conceivable type. Figure 1 shows the GEQs of two values of basic type (Int and String), and two composite types ((Int,String) and \([\text{Int}]^3\)):

\[
\begin{array}{ccc}
42 & :: & \text{Int} \\
"Hello!" & :: & \text{String} \\
(42,"Hello!") & :: & (\text{Int}, \text{String}) \\
[1,2,3] & :: & [\text{Int}] \\
\end{array}
\]

\[
\begin{array}{c}
\text{42} \\
\text{"Hello!"} \\
(42,"Hello!") \\
[1,2,3]
\end{array}
\]

\[
\begin{array}{c}
\text{Hello} \\
\text{Hello} \\
\text{Hello} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Cons} \rightarrow \text{1} \\
\text{Cons} \rightarrow \text{2} \\
\text{Cons} \rightarrow \text{3} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Nil} \\
\end{array}
\]

\textbf{Fig. 1.} Values \(v\) of type \(t\) and their corresponding GEQs.

GUIs typically consist of traditional elements such as buttons, edit fields, radio, and check buttons. These have been provided in the GEC Toolkit using the \textit{specialization} mechanism of generic programming. This means that for these GUI elements new data types have been introduced that \textit{model} these GUI elements. Figure 2 gives the types of some of them and also shows what they look like when applied to gGEC.

Another issue that needs to be addressed with GUIs is the \textit{layout} of elements. The \textit{default layout strategy} of the GEC Toolkit is to arrange data constructor arguments below each other, with the top element right to the data constructor itself. A number of specialized data types have been defined to influence the

---

1 \textbf{generic} \(f\ t :: (T \ t)\) introduces the generic function \(f\) that is generic in type argument \(t\). \((T \ t)\) is the type of \(f\).
2 \(:: \ T_1 := T_2\) introduces the type synonym \(T_1\) for type \(T_2\).
3 \([T]\) is the type \textit{list} of \(T\).
4 \(:: \ T = C_1 | \ldots | C_n\) introduces the \textit{type constructor} \(T\) with data constructors \(C_i\).
layout of elements. Let $v_1 :: t_1$ and $v_2 :: t_2$ be given. Then $(v_1 <\ast|> v_2) :: (t_1 <\ast|> t_2)$ puts $v_2$ below $v_1$, with their left edges aligned. Analogously, the combinators $<\ast|>$ and $<\ast|>$ align the centers and right edges. $(v_1 <\ast|> v_2) :: (t_1 <\ast|> t_2)$ puts $v_2$ right to $v_1$, with their top edges aligned. Analogously, the combinators $<\ast|>$ and $<\ast|>$ align the centers and bottom edges.

The GEC Toolkit is provided with an abstraction mechanism that allows the creation of GECs with the same data model type $d$, but with different view model types $v$ [6]. Such an abstraction is created by converting values of type $d$ to $v$ and vice versa. In many cases this conversion is a bijection of type $(\text{Bimap } d v)$:

$$
\text{Bimap } d v = \{ \text{map_to} :: d \rightarrow v, \text{map_from} :: v \rightarrow d \}
$$

The generic $g\text{GEC}$ function is specialized for the abstract data type $(\text{AGEC } d)$. It is created with the constructor function $\text{mkAGEC}$ given a bijection of type $(\text{Bimap } d v)$ and an initial value of type $d$. The generic function is specialized in such a way that it creates a GEC, that is encapsulated within the $(\text{AGEC } d)$ value, and that works as a $\text{GEC}_d$ in the data domain of which it is part.

$$
\text{mkAGEC} :: (\text{Bimap } d v) \cdot d \rightarrow (\text{AGEC } d) \mid g\text{GEC} \mid v
$$

Given $g :: (\text{AGEC } d)$, then $"\,\!\!\!\!\,g$ is the current value of type $d$, and $(g \leftarrow \text{new})$ is a new value of type $(\text{AGEC } d)$ with current value $\text{new} :: d$. These operations obey the simple law $\text{new} :: (\text{AGEC } d)$.

$$
\text{mkAGEC} :: (\text{Bimap } d v) \cdot d \rightarrow (\text{AGEC } d) \mid ^{\text{gGEC}} \mid v
$$

Abstraction is crucial to obtain easily customizable domain data models. As an example, consider the following $\text{GEC}_{\text{AGEC } \text{Int}}$ that can be used, and freely exchanged, within the very same domain data model: int $\text{AGEC}_{\text{Int}}$ is an integer value editor; dynamic $\text{AGEC}_{\text{Int}}$ is an integer expression editor [7] in which only those Clean expressions can be edited that yield an Int type; counter $\text{AGEC}_{\text{Int}}$ is a spin-button.

We have developed the following programming method to effectively construct GUI applications with the GEC Toolkit:

1. Develop the pure domain data model $d$ without any abstraction.
2. Develop another view data model \( V \) that uses abstraction in the right places.
3. Create \((\text{Bimap} \ D \ V)\) which contains the transformations between \( D \) and \( V \).
4. Create the abstract editor \((\text{AGEC} \ D)\) using the \((\text{Bimap} \ D \ V)\).

We illustrate the programming method by means of the following code fragment:

\[
\begin{align*}
\text{:: } D &= \ldots [\text{Int}] \ldots \\
\text{:: } V &= \ldots (\text{AGEC} [\text{Int}]) \ldots \\
\text{ListV} &= \text{ListV} (\text{Maybe}^\text{a} (\text{Int} <\leftrightarrow \text{ListV})) \\
\text{convertList} &:: ([\text{Int}] \rightarrow (\text{AGEC} [\text{Int}])) \\
\text{convertList} &= \text{mkAGEC} \{ \text{map_to} = \text{toView}, \text{map_from} = \text{toDomain} \} \\
\text{where toView} &:: [\text{Int}] \rightarrow \text{ListV} \\
\text{toView} \ [\ ] &= \text{ListV} \text{Nothing} \\
\text{toView} \ [x : \text{x}s] &= \text{ListV} (\text{Just} \ (x <\leftrightarrow \text{toView} \text{x}s)) \\
\text{toDomain} &:: \text{ListV} \rightarrow [\text{Int}] \\
\text{toDomain} (\text{ListV} \text{Nothing}) &= [\ ] \\
\text{toDomain} (\text{ListV} (\text{Just} \ (x <\leftrightarrow \text{x}s))) &= [x : \text{toDomain} \text{x}s] \\
\end{align*}
\]

The domain data model \( D \) has an integer list component which elements need to be rendered horizontally. Therefore, the view data model \( V \) uses abstraction over the integer list. The conversions between \( D \) and \( V \) need to transform \([\text{Int}]\) values to \((\text{AGEC} [\text{Int}])\) values, and vice versa. This is defined by \text{convertList}, which implements the view of the abstract element as \( \text{ListV} \). \( \text{ListV} \) must be a new type because list is a recursive data type. This is also reflected in the recursive structure of the conversion functions \text{toView} and \text{toDomain}.

3. The Improved \( \text{GEC Toolkit} \) Programming Method

In the previous section we have introduced the \( \text{GEC Toolkit} \) and its programming method. The programming method relies on the abstraction mechanism of the \( \text{GEC Toolkit} \). We identify the following issues with this mechanism:

1. The upside of abstraction is that the programmer does not need to change her code for those (sub)types \( v \) that have been abstracted to \((\text{AGEC} v)\) when switching between abstract components. The downside is that she does have to change her code for those (sub)types that she decides about afterwards to become either abstract or concrete. This is a normal consequence of using abstraction.
2. Recursive data domain (sub)types can only be made abstract by introducing new types and recursive conversion functions.

It should be noted that these issues do not decrease the expressive power essentially, but only stylistically.

The improvement that we propose is the following. Instead of handling the complete transformation from \( D \) values to \( V \) values and vice versa in one go, we

\[^{a}:: \text{Maybe} \ a = \text{Just} \ a | \text{Nothing}. \] This type is useful for handling optional values.
should identify those (sub)types $D_i$ of $D$ for which we want to apply abstraction, so replace with $(\text{AGEC } D_i)$. This leads to a family of functions $f_j :: D_i \rightarrow (\text{AGEC } D_i)$. Now we can specialize each member of this family as follows:

$$g(\text{GECE})[D_j] \ldots dv \text{ env} = \text{specialize } f_i dv \text{ env}$$

and we are done! The technical breakthrough to this apparently simple procedure has been accomplished with the new and complex GEC Toolkit function specialize :: $(d \rightarrow (\text{AGEC } d)) \rightarrow (\text{GECFunction } d (\text{PSt ps}))$. Its task is to create the GECs, that is encapsulated inside the $(d \rightarrow (\text{AGEC } d))$ function in such a way that it can be addressed with the GEC methods for a GEC. Its implementation is beyond the scope of this paper. Instead, we focus on the consequences for the programming method. The new programming method is as follows:

1. Develop the pure data domain model $D$ without any abstraction.
2. Develop $f_j :: D_i \rightarrow (\text{AGEC } D_i)$ for those (sub)types of $D$ that need to be specialized.
3. Specialize each $D_j$ as described above with the function specialize.

This improves the old method in the following ways: (i) It is modular: instead of one $(\text{Bimap } D V)$ the programmer writes several conversions $D_i \rightarrow (\text{AGEC } D_i)$. These functions are easier to understand and can be reused in arbitrary many data domain models $D$. (ii) The view data model $V$ has been eliminated. This implies that the programmer does not have to change her code when switching (sub)types of the pure domain data model to become abstract or not. (iii) The new way of handling abstraction merges the abstraction mechanism with the generic programming scheme. Because the generic programming scheme is inherently recursive, this eliminates the issue of programming recursive conversion functions. (iv) An early experiment with a large application suggests that the new method reduces the number of lines of code with 30%.

Before we move to the case study, we illustrate the new programming method with the list example at the end of Section 2. The essential code fragment is:

```haskell
:: ListV := Maybe (Int <*> [Int])
g(\text{GECE})[[Int]] t pSt = \text{specialize } \text{horlistAGEC } t pSt
where \text{horlistAGEC} = \text{mkAGEC } \{\text{map_to} = \text{toView}, \text{map_from} = \text{toDomain}\}
toView :: [Int] \rightarrow \text{ListV}
toView [] = \text{Nothing}
toView [x : xs] = \text{Just } (x <*> xs)
toDomain :: \text{ListV} \rightarrow [Int]
toDomain Nothing = []
toDomain (\text{Just } (x <*> xs)) = [x : xs]
```

The important differences to observe are: (i) ListV is not a new type anymore, but a type synonym. We have eliminated the need for a new type. (ii) The conversion functions toView and toDomain are not recursive. (iii) Already this very small example shows that the specification becomes shorter and clearer.
4 Case Study: a Family Tree Editor

In this section we demonstrate how to program a GUI using the GEC Toolkit. The case study that we consider is that of a family tree editor. This case study is interesting because of the following reasons:

- It has dynamic behaviour: when edited, (sub) family trees may expand or decrease in size. This causes recalculation of the layout of the remaining (sub) family trees.
- This program can not be created with a visual editor because it has dynamic behavior. Instead, it must be programmed.
- It has logical behaviour: in this case study we want to impose the restrictions that marriage occurs only between two persons of opposite gender and only married couples have children.
- Family trees are usually rendered from top to bottom, which contrasts the default layout strategy of the GEC Toolkit. This is a good test case how well customization of layout works.

We follow the steps of the programming method of Section 3.

Step 1. Develop the Pure Data Domain Model. In the first step we develop the pure data domain model $D$ of the family tree editor. In this case, $D$ is the recursive tree-like data type Family. Its nodes contain information about a person (gender and name), civil status (married or single). Its subtrees are the person’s offspring. Because a person might not be married, the spouse and children are encoded with a Maybe type. The corresponding data types should not be surprising for people familiar with functional programming:

$$
:: \text{Family} = \text{Family} \times \text{Person} \times \text{CivilStatus} \times (\text{Maybe} (\text{Person} \times \text{Kids})) \\
:: \text{Person} = \text{Person} \times \text{Gender} \times \text{String} \\
:: \text{Gender} = \text{Male} | \text{Female} \\
:: \text{CivilStatus} = \text{Married} | \text{Single} \\
:: \text{Kids} = \text{Kids} \times [\text{Family}]
$$

Although this type definition is rather compact, its automatically derived GEC Family is not. The background window in Fig. 3 gives the screenshot of a small family constructed with the editor, that consists of parents Peter and Mirjam and their boys Tijmen and Arjen. It should be clear that this editor is uninformative even to an informed programmer. It also does not implement the logic behaviour requirements. In contrast, the editor in the foreground window is much more compact, uses a more appealing layout scheme, displays redundant information such as number of children, and implements the behaviour requirements.

Step 2. Design the Abstract Types. The next step is to decide what (sub)types to specialize. If we compare the two GUIs in Fig. 3 we conclude that Person, Kids, and Family have to be specialized.
A Person has to be displayed as Peter instead of Male. Expressed as a function: `toView (Person gender name) = name <*|> gender`. This puts the gender information below the name and right-aligned. The inverse function is trivial: `toDomain (name <*|> gender) = Person gender name`. The full specialization is defined by:

```haskell
personAGEC :: (Person \rightarrow\ AEGC Person)
personAGEC = mkAGEC {map_to = toView, map_from = toDomain}
```

The next type to specialize is Kids. Because Kids are defined with a list, the default rendering uses the default list rendering (see also Fig. 1) which is inadequate for our purposes. Instead, we want to display the children next to each other. We use the library function `hor2listAGEC :: a \rightarrow\ AEGC [a]`:

- `hor2listAGEC a [a1..an]` creates an interactive horizontal list with initial elements `[a1..an] (n ≥ 0)`. New list elements have default value `a`. Above this list, we want to display the number of children. This is expressed as:

```haskell
:: KidsView := Display String <|*|> AEGC [Family]

toView :: Kids \rightarrow\ KidsView
toView (Kids ks) = nrOfKids (length ks) <|*|> hor2listAGEC default ks
where nrOfKids n = Display (toString n ++ " Child" ++ if (n==1) " " "ren ")
      default = Family (Person Male "") Single Nothing
```
Converting edited values back to the domain model type is straightforward:

\[
\text{toDomain} :: \text{KidsView} \rightarrow \text{Kids}
\]

\[
\text{toDomain} (_ <|*|> \text{list}) = \text{case } \text{list of } \text{ks} \rightarrow \text{Kids ks}
\]

Putting everything together proceeds as Person:

\[
\text{kidsAGEC} :: (\text{Kids} \rightarrow \text{AGEC Kids})
\]

\[
\text{kidsAGEC} = \text{mkAGEC} \{ \text{map}_\text{to} = \text{toView}, \text{map}_\text{from} = \text{toDomain} \}
\]

The Family specialization requires more attention because it needs to implement both a pleasant visualization and the logic behaviour requirements. The visualization is as follows: the partners in a couple are placed next to each other (<*>); below them and to the left (<*|>) the civil status is shown; and below that and centered (<|*|>) the children are shown. We use the Maybe type in the view model to display nothing at all in case of Nothing values, and (gGEC x) in case of (Just x) values. Therefore, the view data domain has type:

:: \text{FamilyView} := \text{Person <*> Maybe Person <*> CivilStatus <*> Maybe Kids}

Mapping data domain model values to view domain model values and vice versa is done with toView and toDomain. These functions implement the visualization and logic behaviour. Their definitions are:

\[
\text{toView} :: \text{Family} \rightarrow \text{FamilyView}
\]

\[
\begin{align*}
\text{toView} (\text{Family} \ p1 \ \text{Single} \ _) & = p1 <*> \text{Nothing <*> Single <*> Nothing} \\
\text{toView} (\text{Family} \ p1 \ cs \ (\text{Just} \ (p2,\text{kids}))) & = p1 <*> \text{Just p2 <*> cs <*> Just kids} \\
\text{toView} (\text{Family} \ p1 \ cs \ \text{Nothing}) & = p1 <*> \text{Just (other p1) <*> cs <*> Just []}
\end{align*}
\]

where

\[
\text{other} :: \text{Person} \rightarrow \text{Person}
\]

\[
\begin{align*}
\text{other} (\text{Person} \ \text{Female} \ _) & = \text{Person} \ \text{Male} \ "" \\
\text{other} (\text{Person} \ \text{Male} \ _) & = \text{Person} \ \text{Female} \ ""
\end{align*}
\]

\[
\text{toDomain} :: \text{FamilyView} \rightarrow \text{Family}
\]

\[
\begin{align*}
\text{toDomain} (p1 <*> \text{Nothing <*> cs <*> _}) & = \text{Family} \ p1 \ cs \ \text{Nothing} \\
\text{toDomain} (p1 <*> \text{Just p2 <*> cs <*> (Just kids)}) & = \text{Family} \ p1 \ cs \ (\text{Just} \ (p2,\text{kids}))
\end{align*}
\]

The logic behaviour requirement that singles have no children is imposed by the first alternative of toView and toDomain. The requirement that marriage is between persons of opposite gender is imposed by the last alternative of toView, using the local function other :: Person \rightarrow Person.

Again, the specialization is assembled analogously to Person and Kids.

\[
\text{familyAGEC} :: (\text{Family} \rightarrow \text{AGEC Family})
\]

\[
\text{familyAGEC} = \text{mkAGEC} \{ \text{map}_\text{to} = \text{toView}, \text{map}_\text{from} = \text{toDomain} \}
\]

**Step 3. Specialize Abstract Types.** As said earlier in Section 3, this is a trivial step, and we show only its code without further comment:
This concludes the case study. It demonstrates the following points. (i) It shows that the types of the data model are not complex. They belong to any introductory course in functional programming. (ii) A default visualization is always present, but it might not be adequate. However, it can be used for initial testing and verification of the data model. (iii) Improving the visualization of the data model amounts to identification of (sub)types $D_i$ for which specialization functions $(D_i \rightarrow \text{AGEC } D_i)$ need to be developed. These are bijections between $D_i$ and $V_i$, and they can be defined with pure functions on pure data domains.

5 Related Work

The GEC Toolkit is a refined version of the well-known model-view paradigm [18], introduced by Trygve Reenskaug (then named as the model-view-controller paradigm) in the language Smalltalk. In the GEC Toolkit both models and views are defined by means of data models. The generic programming technology provides automatic and specialized visualization of all data models.

Other model-view approaches based on functional programming use a similar value-based approach [10], or an event-based version [17]. In both cases, the programmer needs to explicitly handle view registration and manipulation.

The Vital system [14] is an interactive graphical environment for direct manipulation of Haskell-like scripts. Shared goals are: direct manipulation of functional expressions, manipulation of custom types, views that depend on the data type (data type styles), guarded data types, and the ability to work with infinite data structures. Differences are that our system is completely implemented in Clean, while the Vital system has been implemented in Java. This implies that our system can handle, by construction, all Clean values. Obviously, they are well-typed. In addition, the purpose of a GEC is to edit values of type $t$, while the purpose of a Vital session is to edit Haskell scripts.

Taking a different perspective on the type-directed nature of our approach, one can argue that it is also possible to obtain editors by starting from a grammar specification. Projects in this flavor are for instance Proxima [23], which relies on XML and its DTD (Document Type Definition language), and the Asf+Sdf Meta-Environment [9] which uses an Asf syntax specification and a 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.

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

We have presented a programming method for the GEC Toolkit, and illustrated it by means of the family tree editor case study. Programming GUIs with the GEC Toolkit requires knowledge of functional data structures, such as algebraic data types, and functions that manipulate them. This is material that is covered in any introductory course in functional programming. This enables novice programmers to program highly dynamic GUI applications.

We are currently working on a Web-enabled back-end for the GEC Toolkit. This expands the application domain of GEC programming from the desktop to the world wide web. We are investigating whether the high level of abstraction facilitates reasoning about interactive applications, perhaps using proof tools such as Sparkle.

Acknowledgements

The interactive family tree case study was suggested by Marie-José van Diem.

References


