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*, *uniqueness type system*, and high quality *code generator*. In the Clean project, there is about 13 years of research and experience with GUI programming, resulting in the Object I/O library [2, 3] which is also available for Haskell [1].

We have constructed large GUI applications with Clean and Object I/O. Two examples are the *Integrated Development Environment* of Clean itself and the *proof tool assistant* Sparkle [12]. Although Object I/O offers a high level of abstraction, there is still a steep learning curve for programmers to become proficient. The GEC Toolkit attempts to tackle this problem by taking a radical point of view: the programmer should exclusively *model* his GUI instead of *realizing* it in a *widget* based style. The model is expressed using standard functional data types, and the behaviour is expressed using functions over these domains. This is standard material in any functional programming course. Hence, the GEC Toolkit brings GUI programming within easy reach of every functional programmer.

In this paper we describe the programming method of GUI programming using the GEC Toolkit. Part of this material has been presented earlier [5–7]. This is covered in Section 2. The contribution of this paper consists of two parts. **(i)** We present a programming method for the GEC Toolkit. This method has been realized by means of an improved abstraction mechanism. This is presented in Section 3. **(ii)** We illustrate the improved programming method by means of a complicated case study: a *family tree editor* in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 The GEC Toolkit

The *key technology* on which the GEC Toolkit has been built is *generic programming* [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∀skell [11] and Clean [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_t) for any conceivable data type t. A GEC_t 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_t can only be changed in such a way that any new value is of type t.

The generic (kind-indexed family of) function(s) `gGEC` that creates GEC_ts has type `(GECFunction t (PSt ps))`. In Clean, this is declared as follows:

generic[1] `gGEC t :: (GECFunction t (PSt ps))`

The type synonym (`GECFunction t env`) is a function that takes two arguments, `t` and `env`. It creates a GEC_t in the environment of type `env`. It returns the updated environment, but also the *methods* (of type (`GECMethods t env`)) that a programmer can invoke to obtain access to the GEC_t in the environment. We will not use the GEC methods in this paper.

`:: GECFunction t env :==`[2] `t env → (GECMethods t env,env)`

The environment parameter is instantiated with (`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 *state monads* [19] do in Haskell.

`gGEC` is a generic function, and hence it can create a GEC for any conceivable type. Figure 1 shows the GECs of two values of basic type (`Int` and `String`), and two composite types ((`Int`,`String`) and `[Int]`[3]):

`42`	`:: Int`	42
`"Hello!"`	`:: String`	Hello!
`(42,"Hello!")`	`:: (Int,String)`	42 Hello!
`[1,2,3]`	`:: [Int]`	_Cons 1 _Cons 2 _Cons 3 _Nil

Fig. 1. Values v of type t and their corresponding GEC_t.

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 *specialization* mechanism of generic programming. This means that for these GUI elements new data types have been introduced that *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 *layout* of elements. The *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] **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 *list of T*.

[4] :: $T = C_1 \mid \ldots \mid C_n$ introduces the *type constructor* T with *data constructors* C_i.

`:: Button =`[4]`Pressed | Button Int String`	`(Button 50 "Press Me!")`	Press Me!
`:: UpDown = UpPressed | DownPressed | Neutral`	`Neutral`	
`:: Display a = Display a`	`(Display "Hello!")`	Hello

Fig. 2. Specialized types t for GUI programming, a value v, and GEC_t.

layout of elements. Let $v_1 :: t_1$ and $v_2 :: t_2$ be given. Then (v_1 `<*|>` v_2) :: (t_1 `<*|>`[5] t_2) puts v_2 below v_1, with their left edges aligned. Analogously, the combinators `<|*|>` and `<|*>` align the centers and right edges. (v_1 `<^*>` v_2) :: (t_1 `<^*>` t_2) puts v_2 right to v_1, with their top edges aligned. Analogously, the combinators `<-*>` and `<.*>` 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 (`Bimap d v`)[6]:

```
:: Bimap d v = { map_to :: d → v, map_from :: v → d }
```

The generic `gGEC` function is specialized for the abstract data type (`AGEC d`). It is created with the constructor function `mkAGEC` given a bijection of type (`Bimap d v`) and an initial value of type `d`. The generic function is specialized in such a way that it creates a GEC_v that is encapsulated within the (`AGEC d`) value, and that works as a GEC_d in the data domain of which it is part.

```
mkAGEC :: (Bimap d v) d → (AGEC d) |⁷ gGEC{|*|} v
```

Given `g :: (AGEC d)`, then `(^^g)` is the current value of type `d`, and (`g ^= new`) is a new value of type (`AGEC d`) with current value `new :: d`. These operations obey the simple law `^^(g ^= new) = new`.

```
(^^)         :: (AGEC d) → d
(^=) infixl  :: (AGEC d) d → (AGEC d)
```

Abstraction is crucial to obtain easily customizable domain data models. As an example, consider the following $\mathsf{GEC}_{(AGEC\ Int)}$s that can be used, and freely exchanged, within the very same domain data model: `intAGEC` (42) is an integer *value* editor; `dynamicAGEC` (7*3*2) is an integer *expression editor* [7] in which only those Clean expressions can be edited that yield an `Int` type; `counterAGEC` (42) 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.

[5] We use infix type constructors here for clarity, although Clean does not allow this.

[6] In fact, we allow a more general conversion relation between domain and view, but that is outside the scope of this paper. Please consult [6] for the more general version.

[7] In a type definition of a function, the used *overloaded* and *generic* functions are listed behind `|`.

2. Develop *another* view data model `V` that uses abstraction in the right places.
3. Create (`Bimap D V`) which contains the transformations between `D` and `V`.
4. Create the abstract editor (`AGEC D`) using the (`Bimap D V`).

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

```
:: D     = ...[Int]...
:: V     = ...(AGEC [Int])...
:: ListV = ListV (Maybe⁸ (Int <^*> ListV))

convertList :: ([Int] → (AGEC [Int]))
convertList = mkAGEC { map_to = toView, map_from = toDomain }
where toView :: [Int] → ListV
      toView []       = ListV Nothing
      toView [x : xs] = ListV (Just (x <^*> toView xs))

      toDomain :: ListV → [Int]
      toDomain (ListV Nothing)           = []
      toDomain (ListV (Just (x <^*> xs))) = [x : toDomain xs]
```

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 `[Int]` values to (`AGEC [Int]`) values, and vice versa. This is defined by `convertList`, which implements the view of the abstract element as `ListV`. `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 `toView` and `toDomain`.

3 The Improved GEC Toolkit Programming Method

In the previous section we have introduced the GEC Toolkit and its programming method. The programming method relies on the abstraction mechanism of the 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 (`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

[8] `:: Maybe a = Just a | Nothing`. This type is useful for handling optional values.

should identify those (sub)types $\mathtt{D}_i$ of D for which we want to apply abstraction, so replace with (AGEC $\mathtt{D}_i$). This leads to a family of functions $\mathtt{f}_i :: \mathtt{D}_i \rightarrow (\mathtt{AGEC}\ \mathtt{D}_i)$. Now we can specialize each member of this family as follows:

```
gGEC{|Di|} ... dv env = specialize fi dv 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 → (AGEC d)) → (GECFunction d (PSt ps))`. Its task is to create the GEC_v that is encapsulated inside the (d → (AGEC d)) function in such a way that it can be addressed with the GEC methods for a GEC_d. 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 $\mathtt{f}_i :: \mathtt{D}_i \rightarrow (\mathtt{AGEC}\ \mathtt{D}_i)$ for those (sub)types of D that need to be specialized.
3. Specialize each $\mathtt{D}_i$ as described above with the function `specialize`.

This improves the old method in the following ways: **(i)** It is *modular*: instead of one (Bimap D V) the programmer writes several conversions $\mathtt{D}_i \rightarrow (\mathtt{AGEC}\ \mathtt{D}_i)$. 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 the 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:

```
:: ListV :== Maybe (Int <^*> [Int])

gGEC{|[Int]|} t pSt  = specialize horlistAGEC t pSt
where horlistAGEC    = mkAGEC {map_to = toView, map_from = toDomain}
      toView :: [Int] → ListV
      toView []         = Nothing
      toView [x : xs]   = Just (x <^*> xs)

      toDomain :: ListV → [Int]
      toDomain Nothing              = []
      toDomain (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:

```
:: Family      = Family Person CivilStatus (Maybe (Person,Kids))
:: Person      = Person Gender String
:: Gender      = Male    | Female
:: CivilStatus = Married | Single
:: Kids        = Kids [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.

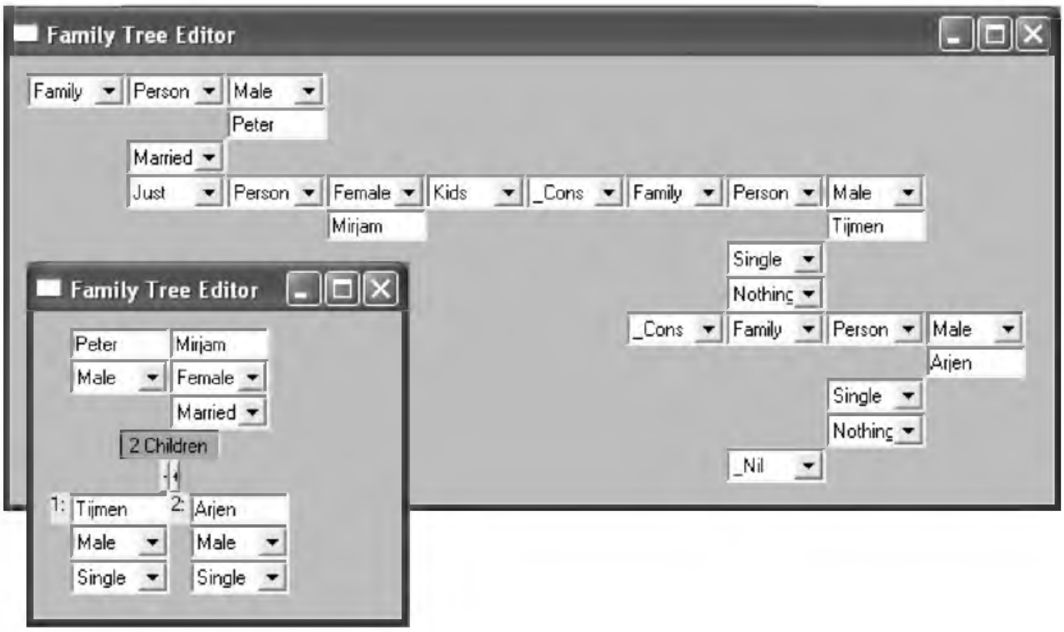

Fig. 3. GEC_{Family}: default (background) and specialized (foreground) rendering.

A Person has to be displayed as 

instead of 

. 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:

```
personAGEC :: (Person → AGEC 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 [a] → AGEC [a]`: `hor2listAGEC a` $[\mathtt{a}_1..\mathtt{a}_n]$ creates an interactive horizontal list with initial elements $[\mathtt{a}_1..\mathtt{a}_n]$ $(n \geq 0)$. New list elements have default value a. Above this list, we want to display the number of children. This is expressed as:

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

toView :: Kids → 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:

```
toDomain :: KidsView → Kids
toDomain (_ <|*|> list) = case ^^list of ks → Kids ks
```

Putting everything together proceeds as `Person`:

```
kidsAGEC :: (Kids → AGEC Kids)
kidsAGEC = mkAGEC {map_to = toView, map_from = 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:

```
:: FamilyView :== 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:

```
toView :: Family → FamilyView
toView (Family p1 Single _)   = p1 <^*> Nothing <*|> Single <|*|> Nothing
toView (Family p1 cs (Just (p2,kids)))
                              = p1 <^*> Just p2 <*|> cs     <|*|> Just kids
toView (Family p1 cs Nothing) = p1 <^*> Just (other p1)
                                                  <*|> cs     <|*|> Just []
where other :: Person → Person
      other (Person Female _) = Person Male   ""
      other (Person Male   _) = Person Female ""

toDomain :: FamilyView → Family
toDomain (p1 <^*> Nothing <*|> cs <|*|> _) = Family p1 cs Nothing
toDomain (p1 <^*> Just p2 <*|> cs <|*|> (Just kids))
                                           = Family p1 cs (Just (p2,kids))
```

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 → Person`.

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

```
familyAGEC :: (Family → AGEC Family)
familyAGEC = mkAGEC {map_to = toView, map_from = 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:

```
gGEC{|Family|} t pSt = specialize familyAGEC t pSt
gGEC{|Kids|}   t pSt = specialize kidsAGEC   t pSt
gGEC{|Person|} t pSt = specialize personAGEC t pSt
```

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

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

1. 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.
2. P. Achten and R. Plasmeijer. The Ins and Outs of Concurrent CLEAN I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.
3. 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.
4. Achten, Peter and van Eekelen, Marko and Plasmeijer, Rinus and van Weelden, Arjen. Arrows for Generic Graphical Editor Components. Technical report NIII-R0416, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands, 2004. available at http://www.niii.kun.nl/research/reports/full/NIII-R0416.pdf.
5. 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*, volume 3145 of *LNCS*. Edinburgh, UK, Springer, 2003.
6. 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.
7. Achten, Peter, van Eekelen, Marko, Plasmeijer, Rinus and van Weelden, Arjen. Automatic Generation of Editors for Higher-Order Data Structures. In Wei-Ngan Chin, editor, *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 262–279. 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. 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.
10. K. Claessen, T. Vullinghs, and E. Meijer. Structuring Graphical Paradigms in TkGofer. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 251–262, Amsterdam, The Netherlands, 9-11 June 1997. ACM Press.
11. 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.
12. M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.
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. W. Karlsen, Einar and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *Implementation of Functional Languages, Selected Papers*, volume 1467 of *LNCS*, pages 285–303, St.Andrews, Scotland, 1997. Springer.
18. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
19. S. L Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, Charleston, Jan. 1993. ACM.
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. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishing Company, 1993. ISBN 0-201-41663-8.
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 editor for structured documents*. PhD thesis, University of Utrecht, 2004.