A Functional Programming Technique for Forms in Graphical User Interfaces

Sander Evers[1], Peter Achten[1], and Jan Kuper[2]

[1] Radboud University Nijmegen, Department of Software Technology, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.
[2] University of Twente, Department of Computer Science, P.O.Box 217, 7500 AE Enschede, The Netherlands.
{s.evers,p.achten}@cs.ru.nl, jankuper@cs.utwente.nl

Abstract. This paper presents FunctionalForms, a combinator library for constructing fully functioning *forms* in a concise and flexible way. A form is a part of a graphical user interface (GUI) restricted to displaying a value and allowing the user to modify it. The library is built on top of the medium-level GUI library wxHaskell. To obtain complete separation between the structure of a form's layout and that of the edited values, we introduce a novel use of *compositional functional references*.

1 Introduction

In many applications, the graphical user interface (GUI) contains parts which can be considered *forms*: they show a set of values, and allow the user to update them. For example, the omnipresent dialogs labeled *Options*, *Settings* and *Properties* are forms. Also, an address book can be considered a form. (Note that in our sense of the word, a form is not only used for input but also for output.)

Despite their simple functionality, programming these forms is often a time-consuming task. A lot of code is spent on converting values and passing them around; furthermore, creating even the smallest form requires quite some knowledge about the architecture of the GUI library. For larger forms, the code tends to get monolithic, badly readable and inflexible.

In this paper we present the combinator library (or *embedded domain-specific language*) FunctionalForms, built on top of the GUI library wxHaskell[1] (while our earlier work[2] shows that the ideas are general enough to build it on top of another library, Object I/O[3]). It is dedicated for building forms in a concise and compositional way, and abstracts over low-level implementation details. A form built with FunctionalForms can be used as an action on initial data; it returns the modified data in the IO monad.

We take special care to preserve the expressivity of wxHaskell's layout combinators, and to separate the look of a form (what are its constituent forms and what is their relative layout) from the structure of the edited value. It is especially this part of FunctionalForms that is the most important contribution

C. Grelck et al. (Eds.): IFL 2004, LNCS 3474, pp. 35–51, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of our framework: we present a technique which uses *compositional functional references* in a novel way to completely separate the two structures.

To indicate the need for a combinator library for forms, we start with a small form programming example in wxHaskell (Sect. 2). Next, FunctionalForms is developed in two stages. In Sect. 3, we define the form abstraction and construct a naïve combinator library for it; in Sect. 4, we transform this library using compositional functional references in order to obtain the desired layout freedom. An elaborate example of programming with FunctionalForms is presented in Sect. 5. Related work is discussed in Sect. 6 and we conclude in Sect. 7.

2 Form Programming with wxHaskell

A recent GUI library for Haskell is wxHaskell[1], an interface to the extensive cross-platform C++ library wxWidgets[4]. Since wxHaskell (intentionally) does not introduce a complete new programming model, programming follows an object oriented style. We show what this means by giving an example of form programming in wxHaskell.[1] It illustrates the problems of programming forms at a too low level (see Sect. 2.2) and serves as running example throughout the paper.

2.1 Example: A Door Information Form

The form we define shows and alters information about a certain door: the name of the person who works behind it and whether s/he is available. This information is exchanged with the rest of the system using a pair of type (*String*, *Bool*). The GUI (see Fig. 1) consists of a small dialog window with four *controls*: a text entry control to show and alter the name, a drop-down choice control showing either 'come on in' or 'do not disturb' and two buttons to close the dialog: *OK* to confirm the changes we made and *Cancel* to reject them.

Figure 2 shows the code producing this dialog. We give a short overview:

- The program starts by creating an empty dialog[2] and the four controls to populate it. For every object, a pointer (*pdialog*, *pentry*, ...) is returned. Controls have dynamic attributes which can be manipulated by the user and/or the program during their lifetime. In particular, the *text* and *selection* attributes (on the *entry* and *choice* control, resp.) are set[3] to the form's initial values (contained in *initDoor*). We have to convert the *Bool* value into an *Int* first.
- Next, the dialog's layout is specified. The function *widget* creates layout information from a control pointer; the combinators *margin*, *column*, *row*

[1] The version of wxHaskell used throughout this paper is 0.8.

[2] Although the terms *dialog*, *window* and *frame* have slightly different technical meanings, we will use them interchangeably.

[3] The 'assignment operator' := looks like a language construct, but is actually just an infix data constructor defined in the wxHaskell library.

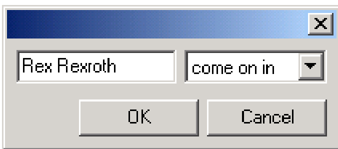

Fig. 1. Door information form

```
doorForm parentWindow initDoor =
  do let (initName, initAvail) = initDoor

       -- create dialog and controls
     pdialog ← dialog parentWindow []
     pentry  ← entry pdialog [text := initName]
     pchoice ← choice pdialog
                 [ items     := ["come on in", "do not disturb"]
                 , selection := bool2int initAvail
                 ]
     pok     ← button pdialog [text := "OK"]
     pcancel ← button pdialog [text := "Cancel"]
       -- set layout
     let mylayout =
           margin 6 $ column 10
             [ row 5 [widget pentry, widget pchoice]
             , alignRight $ row 5 [widget pok, widget pcancel]
             ]
     set pdialog [layout := mylayout]

       -- define event handlers
     let getFinalDoor =
           do finalName ← get pentry text
              finalAvail ← liftM int2bool $ get pchoice selection
              return (finalName, finalAvail)
     let setclose close =
           do set pok [on command :=
                         do finalDoor ← getFinalDoor; close $ Just finalDoor]
              set pcancel [on command := close Nothing]
       -- run dialog
     maybeDoor ← showModal pdialog setclose
     return $ case maybeDoor of
                Just finalDoor → finalDoor
                Nothing        → initDoor
  where bool2int b = if b then 0 else 1
        int2bool i = (i == 0)
```

Fig. 2. wxHaskell code for door information form

and *alignRight* join and transform this information. All layout information is of type *Layout* (we will encounter this type again in Sect. 3.1). Note that the integers 6, 10 and 5 only specify margin widths between controls; actual coordinates are determined by wxHaskell's layout system, which also takes care of resizing controls.
– Both buttons are assigned an *event handler*: a call-back function (IO action) invoked when the user presses the button. It can access the dynamic properties of another control by calling a *get* or *set* function with the corresponding control pointer and property. In the *OK* button's event handler, we obtain the current *String* and *Int* values from the *pentry* and *pchoice* controls, convert the latter back into a *Bool* and join them into a tuple again.
– The last few lines run the dialog modally[4] and determine the function's result: the new values from the controls if the dialog was closed using the *OK* button, and the initial value *initDoor* otherwise.

This *doorForm* function can be used as an IO action in a wxHaskell program.

2.2 Programming Problems Identified

The first thing one may notice about the above example is that, considering the minimal functionality that our dialog provides, 39 lines of code is rather sizable. In the light of defining a form, the only original decisions we express are:

1. We are editing a (*String*, *Bool*) pair; its components are associated with a text entry control and a choice control, respectively.
2. Regarding the latter, the value *True* is associated with the first item, labeled 'come on in', and *False* with the second item, labeled 'do not disturb'.
3. The choice control is placed to the right of the text entry control.

These decisions are encompassed within a lot of procedural code. Moreover, we see that the first two are encoded twice:

1. (i) During control creation, the *text* attribute of control *pentry* is set to the pair's first element; the *selection* attribute of *pchoice* is set to the second.
   (ii) In the button event handler, the values of the same two attributes are retrieved, and a pair is constructed in the same way.
2. (i) During control creation, the *Bool* is converted to *Int*.
   (ii) In the button event handler, the *Int* is converted to *Bool*.

This reduces the modularity and flexibility of our program: if we want to change, say, the choice control into a check box control, we need to make consistent adaptations at two different places.

A third problem, pointed out by Leijen[1], is the possibility to create incorrect layout specifications: forgetting or duplicating a control causes run-time errors.

All three symptoms are evidence that the programming level is too low for forms. In the next section, we design a combinator library to abstract over this level.

---

[4] i.e. the dialog blocks interaction with the rest of the application until it is closed.

3 A Naïve Combinator Library for Forms

In this section, we develop the first stage of FunctionalForms, which focuses on abstracting over low-level form programming details. Structured as a typical combinator library, it revolves around a central data type (*FForm*) that represents both the smallest (atomic) and largest parts of the constructed program; combinators combine and transform these parts.

A value of this type is a *form*: a part of the GUI that is only able to display and alter a certain value. A form lives within a surrounding dialog with *OK* and *Cancel* buttons. When this dialog appears, the form has an *initial value* which is provided by its environment; subsequently, the user can read and alter this value; at the end, the user closes the dialog with one of the buttons, and the form passes the *final value* to the environment. The type of this value is called the *subject type* of the form. It appears as type parameter t in the *FForm* type:

$$\textbf{type}\ \mathit{FForm}\ t\ w\ =\ \mathit{Window}\ w\ \rightarrow\ t\ \rightarrow\ \mathit{IO}\ (\mathit{Layout},\ \mathit{IO}\ t)$$

The top-level IO action, provided with a pointer to a parent window and an initial value, creates the controls which make up the form. It returns a *Layout* value for this form and another IO action. This action is used when the dialog is closed with the *OK* button; it retrieves the form's current value at that moment.

3.1 Components of the Library

Atomic forms correspond to single wxHaskell controls which contain an editable value, such as *entry*. This value is held in some attribute of the control, in this case *text*. The definition of the corresponding form $\underline{\mathit{entry}'}$ simply joins the creation, layout, and attribute-reading functions for this control:

$$\begin{array}{l}\underline{\mathit{entry}'}\ ::\ \mathit{FForm}\ \mathit{String}\ w\\ \underline{\mathit{entry}'}\ =\ \lambda w\ \mathit{init}\rightarrow\\ \quad\textbf{do}\ \mathit{pentry}\ \leftarrow\ \mathit{entry}\ w\ [\mathit{text} := \mathit{init}]\\ \qquad\mathit{return}\ (\mathit{widget}\ \mathit{pentry},\ \mathit{get}\ \mathit{pentry}\ \mathit{text})\end{array}$$

In Fig. 3, some other atomic forms, their subject types, and the corresponding wxHaskell attributes are shown. They are defined analogously. We follow the convention that all exported library functions are underlined.

Forms can be combined into larger forms: taken together, an $\underline{\mathit{entry}'}$ and a $\underline{\mathit{checkBox}'}$ edit a composite value (containing a *String* and a *Bool*). Naïvely, a combinator for joining forms therefore joins their subject types as well as their *Layout* values. However, this will turn out to be a source of trouble for the library (see Sect. 3.2). We demonstrate this with the combinator $\circledast$, which conveniently suits our *doorForm* example:

$$\begin{array}{l}(\circledast)\ ::\ \mathit{FForm}\ t_1\ w\ \rightarrow\ \mathit{FForm}\ t_2\ w\ \rightarrow\ \mathit{FForm}\ (t_1, t_2)\ w\\ \mathit{form}_1\ \circledast\ \mathit{form}_2\ =\ \lambda w\ (\mathit{init}_1, \mathit{init}_2)\rightarrow\\ \textbf{do}\ (\mathit{lay}_1, \mathit{getfin}_1)\ \leftarrow\ \mathit{form}_1\ w\ \mathit{init}_1\\ \quad(\mathit{lay}_2, \mathit{getfin}_2)\ \leftarrow\ \mathit{form}_2\ w\ \mathit{init}_2\\ \quad\mathit{return}\ (\ \mathit{row}\ 5\ [\mathit{lay}_1, \mathit{lay}_2],\ \mathit{liftM2}\ (,)\ \mathit{getfin}_1\ \mathit{getfin}_2\ )\end{array}$$

Name	Appearance	Subject type	wxHaskell attribute
$\underline{entry'}$	foo	*String*	*text*
$\underline{choice'}$	first first second	*Int*	*selection*
$\underline{radioBox'}$	first second	*Int*	*selection*
$\underline{checkBox'}$	☑	*Bool*	*checked*
$\underline{spinCtrl'}$	5	*Int*	*selection*

Fig. 3. Some atomic forms and their subject types

As the type signature shows, the composite form's subject type is a pair of its components' subject types. Its initial value $(init_1, init_2)$ is split up and fed to the two component forms; likewise, the two final values are joined back into a pair (we lift the pair constructor into the IO monad). Regarding layout, both components are put next to each other with a five-pixel gap in between.

Using only $\boxast$ and atomic forms, we can already concisely define a fully functioning form for any combination of simple types expressed in nested pairs. For example, a form for $(String, (Bool, Bool))$ can be defined like:

$$composite \ = \ \underline{entry'} \ \boxast \ (\underline{checkBox'} \ \boxast \ \underline{checkBox'})$$

To actually use this form in a wxHaskell program, we would provide it with an initial value *init* of type $(String, (Bool, Bool))$ and run it:

```
do ...
   final ← runInDialog parentWindow composite init
   ...
```

The function $\underline{runInDialog}$, when given a parent window, a form and an initial value of the form's subject type, yields an IO action producing a modal dialog which contains the form, an *OK* button and a *Cancel* button. This is accomplished by:

1. Setting up the dialog with the buttons.
2. Executing the form's IO action, which creates the controls in the dialog.
3. Augmenting the layout returned by (2) with the layout of the buttons, and attaching it to the dialog.
4. Using the IO action returned by (2) in the *OK* button event handler to retrieve the form's final value.

The result of $\underline{runInDialog}$'s IO action equals the form's final value if the *OK* button is used, and the initial value otherwise. We omit the implementation; it is very similar to the corresponding fragments in Fig. 2.

As the last addition to the combinator library, we define the function $\underline{convert}$[5] and its specialization $\underline{convertL}$. They transform a form's subject type into an 'isomorphic' type, given the corresponding bijection.

$$\underline{convert} :: (t_1 \rightarrow t_2,\ t_2 \rightarrow t_1) \rightarrow FForm\ t_2\ w \rightarrow FForm\ t_1\ w$$

Often, a concept from the data domain, like *week day* or *eye color*, can be captured with a simple enumerated type. To convert between such a type and the zero-based *Int* index used in some atomic forms, we don't need to write out a full bijection; it suffices to enumerate the values in a list. The function $\underline{convertL}$ then maps the first value to 0, the second to 1, etc.:

$$\begin{aligned}
&\underline{convertL} :: Eq\ t \Rightarrow [t] \rightarrow FForm\ Int\ w \rightarrow FForm\ t\ w\\
&\underline{convertL}\ items\ = \underline{convert}\,(f,\ f_{inv})\\
&\quad \textbf{where}\ f\ a\ \ = fromJust\ \$\ elemIndex\ a\ items\\
&\qquad\quad f_{inv}\ i = items!!i
\end{aligned}$$

We are now ready to define the form example from Sect. 2.1 in only three lines:

$$\begin{aligned}
doorForm &= \underline{entry'} \circledast availForm\\
availForm &= \underline{convertL}\,[True, False]\ \$\\
&\qquad \underline{choice'}\ [items := [\text{"come on in"}, \text{"do not disturb"}]]
\end{aligned}$$

3.2 Evaluation of the Combinator Library

An important thing to notice is that the combinator library we defined solves all the problems mentioned in Sect. 2.2. Along with providing a very concise way of specifying the relevant decisions, it also rules out the possibility of forgetting or duplicating controls in the layout specification: an atomic form associates a control with exactly one layout specification, and the combinators maintain this invariant.[6] However, the library has a disadvantage: $\circledast$ is a bad template for form combinators, because it introduces a dependency between the subject type structure and the layout structure of a form. This manifests itself in two ways:

Incompatible types: To increase the layout possibilities for composite forms, the obvious solution would be to introduce combinators which mimic wxHaskell's layout combinators. When we follow the $\circledast$ template, these combinators also have to construct a subject type, but this often causes trouble:
- For one-argument combinators (which transform a single *Layout*) such as *margin*, it is indeed no problem to 'lift' them into the *FForm* domain: we just let them alter the form's layout and leave the subject type alone.
- Lifting a zero-argument combinator such as *label*, which *produces* a *Layout* by itself, is a little more problematic: the lifted combinator should produce a form with a certain subject type and final value. In principle, these can be the unit type and value (). However, every label used in a composite form will then clutter its subject type with another ().

[5] The implementation of $\underline{convert}$ can be found in Fig. 4.

[6] In fact, a similar technique is briefly mentioned in [1] (section *Safety*).

– Combinators of the form $[\mathit{Layout}] \to \mathit{Layout}$, such as *row*, cause even more problems: providing the lifted combinator with a list of forms would force them to have the same subject type. In principle, we could solve this problem by extending the *FForm* type to also accommodate *lists* of *Layout*s, and introducing combinators *nilF* and *consF* to produce such forms, but then our *doorForm* example would turn into

$$\mathit{doorForm} \ = \ \mathit{row}'\ 5\ \$\ \underline{\mathit{entry}'}\ \text{`}\mathit{consF}\text{`}\ (\mathit{availForm}\ \text{`}\mathit{consF}\text{`}\ \mathit{nilF})$$

with subject type $(\mathit{String}, (\mathit{Bool}, ()))$. In practice, this is rather awkward.

Dependency between layout and values: Say we want to swap the two controls in the *doorForm* layout. If we just swap the two operands of ⊛, we also unintentionally change *doorForm*'s subject type from $(\mathit{String}, \mathit{Bool})$ to $(\mathit{Bool}, \mathit{String})$. One way to hack around this would be to $\underline{\mathit{convert}}$ the new form's subject type back:

$$\begin{aligned} &\mathit{doorForm} \ = \ \underline{\mathit{convert}}\,(\mathit{mirror}, \mathit{mirror})\ (\mathit{availForm}\ \circledast\ \underline{\mathit{entry}'}) \\ &\quad \textbf{where}\ \mathit{mirror}\ (a, b) \ = \ (b, a) \end{aligned}$$

... but this is no real solution: with larger forms—say we want to permute eight controls instead of two—the programmer is heavily burdened by these kind of 'plumbing' bijections. Not only is this much work, but it also has an impact on the flexibility of the program: if later we decide to alter the layout structure, we also need to alter the bijection functions again.

The cause of both problems is that we cram too much functionality into the combinators, thereby creating dependencies between two structures which are, in essence, largely unrelated. In the next section, we show how to factor the ⊛ combinator into a layout combinator and a subject type combinator.

4 Separating Subject Type and Layout Combinators Using Compositional Functional References

This section presents the second stage of FunctionalForms. It allows the user to explicitly manage the subject type of a form, separate from its layout, using two types of combinators: subject type combinators, like $\underline{\mathit{declare2}}$ for a pair, and layout combinators, like $\underline{\mathit{row}'}$ and $\underline{\mathit{margin}'}$ (derived from their wxHaskell counterparts). This enables the definition of forms such as

$$\begin{aligned} &\underline{\mathit{declare2}}\ \$\ \lambda(\mathit{name}, \mathit{avail}) \to \\ &\quad \underline{\mathit{row}'}\ 5\ [\mathit{availForm}\ \mathit{avail},\ \underline{\mathit{entry}'}\ \mathit{name}] \end{aligned}$$

to specify a door information form with the name at the first position in the subject type, and at the last position in the layout structure. The connection between the two structures is formed by special values (*name* and *avail* in the example) which we call *compositional functional references*.

4.1 Introducing Compositional Functional References

Reference values are members of an algebraic data type containing two functions:

$$
\begin{aligned}
\textbf{data}\ \mathit{Ref}\ cx\ t \ = \ \mathit{Ref}\{ \ & \mathit{val} \ :: \ cx \to t \\
, \ & \mathit{app} :: \ (t \to t) \to cx \to cx \\
\} &
\end{aligned}
$$

Type variable cx denotes the type of the *context*, a structure of values, which contains some sub-structure of type t. The first function retrieves a t value from a cx structure, while the second updates a cx structure by applying a $t \to t$ update function to the t value at the right spot. An example reference value is *reffst*, a reference to the first element of a pair:

$$
\begin{aligned}
& \mathit{reffst} :: \mathit{Ref}\ (t_1, t_2)\ t_1 \\
& \mathit{reffst} = \mathit{Ref}\ \mathit{fst}\ \mathit{appfst} \\
& \quad \textbf{where}\ \mathit{appfst}\ f\ (x, y) \ = \ (f\ x,\ y)
\end{aligned}
$$

Using *reffst* and *refsnd*, which is defined analogously, we can retrieve or update the values in $\mathit{initcx} \ = \ (39, \text{"foo"})$:

$$
\begin{aligned}
& (\mathit{val}\ \mathit{reffst})\ \mathit{initcx} && \Rightarrow \ 39 \\
& (\mathit{app}\ \mathit{reffst})\ (+3)\ \mathit{initcx} && \Rightarrow \ (42, \text{"foo"}) \\
& (\mathit{app}\ \mathit{refsnd})\ (\mathit{const}\ \text{"bar"})\ \mathit{initcx} && \Rightarrow \ (39, \text{"bar"})
\end{aligned}
$$

Note that when we partially apply the *app* functions by removing *initcx* in the last two examples, we obtain functions of type $cx \to cx$: a context update. We include such a function in the new *FForm* type, which we now present.

4.2 Forms with References

In the transformed library, shown in the right-hand side of Fig. 4, every form has access to the same context, whose type equals the subject type of the topmost form composition. The new *FForm* type clearly shows this: a form no longer depends on an initial *value* for itself, but rather on an initial *context*; and instead of producing a final *value*, it produces a final *context update*. In the *OK* button event handler, this update will be applied to the initial context, yielding a final context.

As the new definition of $\underline{\mathit{entry}\prime}$ shows, an atomic form is now provided by the programmer with a reference value. This determines which part of the context it edits: the *val* function retrieves an initial value from this part and the *app* function writes the final value to this part. The *Ref* type contains the form's subject type, in this case *String*. How the programmer obtains such a reference value is explained in Sect. 4.3.

The combinator $\boxast$ is replaced by $\boxbar$. The resulting composite form distributes the initial context among its components unaltered, instead of splitting it. Conversely, instead of pairing two final component values, it constructs a joint context update by sequencing both component updates (this time, the function composition operator is lifted into the IO monad).

FunctionalForms stage 1	FunctionalForms stage 2

type FForm t w =
  Window w → t →
  IO (Layout, IO t)
	type FForm cx w =
	  Window w → cx →
	  IO (Layout, IO (cx → cx))

entry′ :: FForm String w
entry′ = λw init →
  do pentry ← entry w [text := init]
    return (widget pentry
           , get pentry text
)
	entry′ :: Ref cx String → FForm cx w
	entry′ (Ref val app) = λw init_cx →
	  do pentry ← entry w [text := val init_cx]
	    return (widget pentry
	           , **do** t ← get pentry text;
	                return $ app $ const t
	           )

(⧆) :: FForm t1 w → FForm t2 w
        → FForm (t1, t2) w
form1 ⧆ form2 = λw (init1, init2) →
  do (lay1, getfin1) ← form1 w init1
    (lay2, getfin2) ← form2 w init2
    return (row 5 [lay1, lay2]
           , liftM2 (,) getfin1 getfin2)
	(◫) :: FForm cx w → FForm cx w
	        → FForm cx w
	form1 ◫ form2 = λw init_cx →
	  do (lay1, getupd1) ← form1 w init_cx
	    (lay2, getupd2) ← form2 w init_cx
	    return (row 5 [lay1, lay2]
	           , liftM2 (.) getupd1 getupd2)
	  — actually a template for deriving:
	row′ ::
	  Int → [FForm cx w] → FForm cx w
	margin′ ::
	  Int → FForm cx w → FForm cx w
	label′ :: String → FForm cx w
	⋮

	declare2 :: ((Ref cx t1, Ref cx t2) → z)
	  → Ref cx (t1, t2) → z
	
	declareL :: ([Ref cx t] → z)
	  → Ref cx [t] → z
	⋮ — implementation: see running text

runInDialog :: Window w →
  FForm t (CPanel ()) →
  t → IO t
	runInDialog :: Window w →
	  (Ref cx cx → FForm cx (CPanel ())) →
	  cx → IO cx

convert :: (t1 → t2, t2 → t1) →
  FForm t2 w → FForm t1 w
convert (f, f_inv) form = λw init →
  do (lay, getfin) ← form w $ f init
    return (lay, liftM f_inv getfin)
	convert :: (t1 → t2, t2 → t1) →
	  (Ref cx t2 → z) → (Ref cx t1 → z)
	convert (f, f_inv) refToForm ref =
	  refToForm (refiso • ref)
	  where refiso = Ref f (λg → f_inv . g . f)

Fig. 4. Transforming the combinator library

Since the arguments of ◫ are of the same type, the first problem in Sect. 3.2 is solved: ◫ can easily be generalized to take a list of forms instead of two (and a margin width value), thereby implementing the lifted version $\underline{row}'$ of wxHaskell's layout combinator *row*. As the context update for base case [], we return *id*, the unit value for function composition. This is also the solution for lifting zero-argument layout combinators like *label*.

The second problem is also solved: the two operands of ◫ (and for $\underline{row}'$: all the forms in the list) can be freely swapped without any effect on the initial value for the components or the final value for the composite form.[7] We can conclude that this combinator has no influence on the functionality of a form anymore; indeed it is merely a lifted layout combinator.

In fact, using ◫ as a template, we have lifted *all* of wxHaskell's layout combinators into the *FForm* domain.[8] However, for simplicity's sake, we will still restrict our use of layout combinators to ◫ in the rest of this section.

4.3 Constructing the Subject Type with References

Since the new layout combinators do not construct the subject type, it has to be done in another way: using reference values. For now, we are mainly concerned with subject types consisting of nested pairs. We can derive the reference values to their elements using the reference values to the elements of a simple pair, *reffst* and *refsnd*. This is done by 'normally' composing their *val* functions (*fst* and *snd*), while composing their *app* functions (*appfst* and *appsnd*) in the reverse order. For example, a reference to the c value in $(a, (b, (c, d)))$ is constructed with:

$$Ref\ (fst\ .\ snd\ .\ snd)\ (appsnd\ .\ appsnd\ .\ appfst)$$

This pattern of constructing new reference values can be captured with the operator • for composition of references:

$$
\begin{aligned}
(\bullet) &:: Ref\ b\ c \rightarrow Ref\ a\ b \rightarrow Ref\ a\ c \\
w \bullet v &= Ref\ (val\ w\ .\ val\ v)\ (app\ v\ .\ app\ w)
\end{aligned}
$$

The reference value above can now be written $reffst \bullet refsnd \bullet refsnd$. With the • operator, we can also construct new *forms* in a compositional way. We illustrate this by means of the *doorForm* example, which is *not* compositional when defined in a naïve way:

$$
\begin{aligned}
doorFormNC &:: FForm\ (String, Bool)\ w \\
doorFormNC &= \underline{entry}'\ reffst\ ◫\ availForm\ refsnd
\end{aligned}
$$

[7] Provided that some conditions hold, e.g. that none of the atomic forms is supplied with the same reference value. A formal proof of this can be found in [2].

[8] Alternatively, the *FForm* domain can be structured as a monad. The monadic lifting functions can then be used for this purpose.

This form can only be used as a top-level form; it cannot be usefully joined with another form, because *doorFormNC* ⊞ *otherForm* would force the context type of *otherForm* to be (*String*, *Bool*) as well. Compare this with the compositional way of defining *doorForm*:

$$\begin{array}{l} \mathit{doorForm} :: \mathit{Ref}\ \mathit{cx}\ (\mathit{String}, \mathit{Bool}) \rightarrow \mathit{FForm}\ \mathit{cx}\ w \\ \mathit{doorForm}\ \mathit{ref} = \underline{\mathit{entry}}\ (\mathit{reffst} \bullet \mathit{ref}) \boxplus \mathit{availForm}\ (\mathit{refsnd} \bullet \mathit{ref}) \end{array}$$

This form *can* be used as a component of a larger form. Just like the atomic forms, it should be supplied with a reference value pointing to its subject type (*String*, *Bool*) in a larger context *cx*. It uses this to derive reference values to a *String* and a *Bool* for its sub-forms.

To enforce this pattern of form construction, the library does not export reference creation functions, but only the subject type combinator *declare2*:

$$\begin{array}{l} \underline{\mathit{declare2}} :: ((\mathit{Ref}\ \mathit{cx}\ t_1,\ \mathit{Ref}\ \mathit{cx}\ t_2) \rightarrow z) \rightarrow \mathit{Ref}\ \mathit{cx}\ (t_1, t_2) \rightarrow z \\ \underline{\mathit{declare2}}\ \mathit{refsToForm}\ \mathit{ref} = \mathit{refsToForm}\ (\mathit{reffst} \bullet \mathit{ref},\ \mathit{refsnd} \bullet \mathit{ref}) \end{array}$$

Using this combinator, the same *doorForm* definition can be written as:

$$\begin{array}{l} \mathit{doorForm} = \underline{\mathit{declare2}}\ \$\ \lambda(\mathit{name}, \mathit{avail}) \rightarrow \\ \quad \underline{\mathit{entry}}\ \mathit{name} \boxplus \mathit{availForm}\ \mathit{avail} \end{array}$$

To enable the use of a compositional form like *doorForm* (i.e. parameterized by a reference value) at the top level, the new *runInDialog* is defined to take just this kind of form as its argument. It applies it to $\mathit{refid} = \mathit{Ref}\ \mathit{id}\ \mathit{id}$, the unit element for • (turning *doorForm* back into *doorFormNC*). This is what equates the context type of every form to the subject type of this topmost form.

The new *convert* function also transforms compositional forms. It does this by transforming the reference value that gets passed to a form. Interestingly, this transformation can be performed by composing it with the appropriate *isomorphism reference*; see Fig. 4 for details. Although the type of *convertL* changes due to the type change of *convert*, its textual definition remains the same. The same holds for the user-defined *availForm* in the *doorForm* example.

4.4 Reference Values for Other Subject Types

Up to this point, we have restricted the composite subject types to pairs. Of course, we can easily extend the approach to tuples of higher arity by defining *declare3* et cetera.[9] Using the same scheme as before, it is also possible to define references to the head and tail of a list:

$$\begin{array}{l} \mathit{refhead} :: \mathit{Ref}\ [t]\ t \\ \mathit{refhead} = \mathit{Ref}\ \mathit{head}\ \mathit{apphead}\ \mathbf{where}\ \mathit{apphead}\ f\ (x : \mathit{xs}) = f\ x\ :\ \mathit{xs} \end{array}$$

[9] Using Template Haskell[5], these definitions can be generated automatically. Furthermore, Haskell's type classes can be used to unite the *declare* functions.

$reftail\ ::\ Ref\ [t]\ [t]$
$reftail\ =\ Ref\ tail\ \ apptail\ \ \mathbf{where}\ apptail\ \ f\ (x : xs) =\ x\ :\ f\ xs$

Subsequently, we can define the list of references to all possible list elements, and a subject type combinator for a list (note how $\underline{declareL}$ resembles $\underline{declare2}$):

$refslist\ ::\ [Ref\ [t]\ t]$
$refslist\ =\ refhead\ :\ map\ (\bullet\ reftail)\ refslist$

$\underline{declareL}\ ::\ ([Ref\ cx\ t] \rightarrow z)\ \rightarrow\ Ref\ cx\ [t]\ \rightarrow\ z$
$\underline{declareL}\ refsToForm\ ref\ =\ refsToForm\ \$\ map\ (\bullet\ ref)\ refslist$

The following example illustrates the use of the functions defined in this section.

5 Elaborate Example

To give an impression of the concise declarative style of form programming with FunctionalForms, we present a more elaborate example. While we have thus far kept the atomic form $\underline{entry'}$ as simple as possible for clarity, we use a more flexible version here, with a small adaptation: every atomic form is extended with a property list, which it passes on to its corresponding control.

The form we define is shown in Fig. 6; it edits a list of three alarms. Every alarm consists of three components: a value indicating whether the alarm is enabled, a time setting and a message. This information is encoded in a value of type $(Bool, Int, String)$, where the integer represents the number of minutes elapsed since midnight.

The corresponding code can be found in Fig. 5. In *alarmListForm*, an infinite list of references is generated by *declareL* and bound to *refs*. Then, *makeBox* assigns each reference to an *alarmForm* and puts a box around it. Finally, the first three boxes are taken from the list and put in a column.

An *alarmForm* splits its reference into three parts, which it distributes over a $\underline{checkBox'}$, a *timeForm* and an $\underline{entry'}$. The last two are arranged in a grid, together with two labels (which are aligned middle-left in their cell). The check box is placed left of the grid.

A *timeForm* converts the total number of minutes into a value for hours and a value for minutes using *div* and *mod*, and assigns the corresponding two references to a pair of spin controls. For these controls, minimum and maximum values are set, as well as a custom size.

6 Related Work

The notion of *compositional references* was introduced by Kagawa[6] as a means to compose mutable (i.e. destructively updatable) data structures, such as arrays, in a functional language. Although it was proposed as a primitive data type,

```
alarmListForm :: Ref cx [(Bool, Int, String)] → FForm cx w
alarmListForm = declareL $ λrefs →
  column′ 10 $ take 3 $ zipWith makeBox [1..] refs
  where
    makeBox nr ref = boxed′ ("Alarm " ++ show nr) (alarmForm ref)

alarmForm :: Ref cx (Bool, Int, String) → FForm cx w
alarmForm = declare3 $ λ(enab, time, msg) →
  margin′ 3 $ row′ 8 [ checkBox′ [] enab
                     , grid′ 5 5
                         [ [floatLeft′ $ label′ "time :",     timeForm time]
                         , [floatLeft′ $ label′ "message :", entry′ [] msg]
                         ]
                     ]
timeForm :: Ref cx Int → FForm cx w
timeForm = convert (splittime, jointime) $ declare2 $ λ(hrs, mins) →
  row′ 2
    [ spinCtrl′ 0 23 [outerSize := sz 40 20] hrs
    , spinCtrl′ 0 59 [outerSize := sz 40 20] mins
    ]
  where splittime total = (total ‘div‘ 60, total ‘mod‘ 60)
        jointime (hours, minutes) = 60 ∗ hours + minutes
```

Fig. 5. Definition code for alarm list form

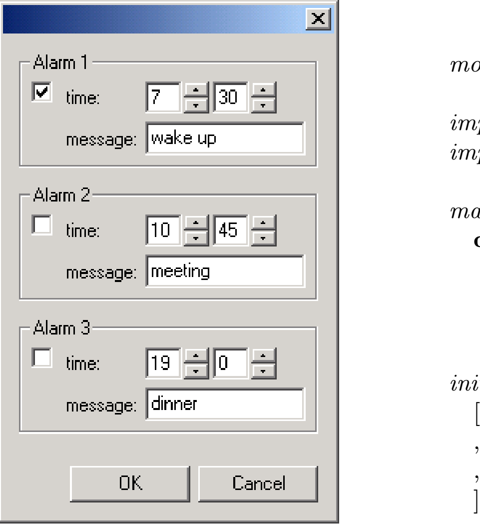

Fig. 6. Appearance of alarm list form

```
module Alarms(main) where

import Graphics.UI.WX
import FForms

main = start $
  do f ← frame []
     final ← runInDialog f
               alarmListForm init
     print final
     close f
init =
  [ (True, 450,  "wake up")
  , (False, 645,  "meeting")
  , (False, 1140, "dinner")
  ]
```

Fig. 7. Startup code for alarm list form

Kagawa also gives a functional account of the reference type. Our references resemble this (except that we use an *apply* function instead of a *write* function to facilitate composition) so we use the name *compositional functional references*.

Closely related are *lenses*[7], which are also pairs of accessor and modificator functions. Several operators, including composition, are used to combine lenses into a large lens which *is* the program; this program specifies a bidirectional transformation between model and view.

Although we have chosen for an underlying GUI library with an object oriented style (which is more widely accepted), declarative form programming is probably achieved most easily on top of a declarative GUI library like FranTK[8] or Fudgets[9]. The latter even defines a form combinator $>\cdot<$ which closely resembles ⧆ (see the corresponding PhD thesis[10], chapter 29). To obtain a layout flexibility similar to ours, a unique name can be assigned to each sub-fudget; these names are used in a *name layout combinator* which is applied to the composite fudget. They play the same role as our references, but:

- Fudget names refer to parts of the layout. We believe that from a top-down design perspective, it is more natural to name the parts of a data structure, because this is designed first and less susceptible to change.
- Fudget names are identifier values. Generating these (unique) values is an extra responsibility for the programmer that our approach does not have.

In functional GUI libraries which are more or less 'object oriented', GUI parts are related using pointers to the controls themselves, instead of to the data structures they edit (our approach) or their layout (the Fudgets approach). Like we have shown in Sect. 2, the wxHaskell control creation functions return these pointers as values in the IO monad. In Clean Object I/O[3], they are generated by a shared environment at user request; in a GUI library for the Curry language[11] (which has a more declarative flavour), these pointers are implemented using free logic variables.

There are several functional libraries for Web form programming. We mention WASH/CGI[12] here; this article provides an overview of the others. With WASH/CGI, the programmer can refer to the (typed) value in a Web form using *input handles*; like wxHaskell's control pointers, these are returned as monadic values by creation functions.

XForms[13], the recent W3C standard for declarative Web forms, also takes the approach of naming parts (XML elements) of the data structure. This is done in the first part of an XForms definition, the *XForms Model*. It also provides every element with an initial value and possibly type or value constraints. In a separate second part, the *XForms User Interface*, GUI controls are bound to these elements by referring to their names.

Generic *Graphical Editor Components* (GECs)[14] use their 'subject type' to convey layout information. A generic function[15] automatically derives the GUI for any given subject type; to create a different GUI for a certain type, one can specialize this function. In order to release this rigid coupling between subject type and layout, *abstract GECs*[16] differentiate between a *domain type* and a *view type*. The GUI is derived from the view type; mapping functions relate domain values to view values, quite like in our *convert* function. Like Fudgets, GECs differ from forms in their ability to react to user events during their whole lifetime and to dynamically create new GECs for editing new values.

7 Conclusions and Future Work

We have introduced FunctionalForms, a combinator library which facilitates the programming of forms in a functional language. (Alternatively, it can be seen as an *embedded domain-specific language* for forms.) First we showed how to build a combinator library capturing the form abstraction on top of an underlying GUI library with an object oriented programming style. This solved the problems of low-level programming like verbosity, but had a drawback: it coupled subject type structure and layout combinator structure together. Then we used compositional functional references in a novel way to release this dependency; this also allowed us to exploit the full power of the layout combinators from the underlying library wxHaskell.

Forms have limited functionality: value editing only affects the rest of the system after the lifetime of a form, and forms can only edit a static, finite, product-like structure of values. While we have already investigated the use of sum-like structures[2] and synchronizing forms briefly, these are yet to be integrated into one framework. However, our results are already of practical use.[10]

A major advantage of our technique is that it does not depend on a special GUI library or language construct. Our earlier work[2], in which we applied the technique to the Clean Object I/O library[3], supports this statement. In fact, the key characteristic of our use of compositional functional references is very general: it allows two different structures to be built from the same set of elements. Therefore, we believe that it can be applied in other areas of functional programming as well.

Acknowledgements

The authors would like to thank Marko van Eekelen, Rinus Plasmeijer and the anonymous referees for their comments on this paper, and Maarten Fokkinga for co-supervising the Master's thesis from which it partially resulted.

References

1. Leijen, D.: wxHaskell – a portable and concise GUI library for Haskell. In: ACM SIGPLAN Haskell Workshop (HW'04), ACM Press (2004)
2. Evers, S.: Form follows function: Editor GUIs in a functional style. Master's thesis, University of Twente (2004) Permanently available at `http://doc.utwente.nl/fid/2101`.
3. Achten, P., Plasmeijer, R.: Interactive Functional Objects in Clean. In Clack, C., Hammond, K., Davie, T., eds.: Proc. of 9th International Workshop on Implementation of Functional Languages, IFL'97. Number 1467 in LNCS, Springer-Verlag, Berlin (1998) 304–321

[10] The library can be downloaded at `http://www.sandr.dds.nl/FunctionalForms`.

4. The wxWidgets home page can be found at `http://www.wxwidgets.org`.
5. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M.M.T., ed.: ACM SIGPLAN Haskell Workshop 02, ACM Press (2002) 1–16
6. Kagawa, K.: Compositional references for stateful functional programming. In: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming (ICFP'97). Volume 32(8) of SIGPLAN Notices., ACM Press (1997) 217–226
7. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. Technical Report MS-CIS-04-15, University of Pennsylvania (2004) An earlier version appeared in the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004, under the title “A Language for Bi-Directional Tree Transformations”.
8. Sage, M.: FranTk - a declarative GUI language for Haskell. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional programming, ACM Press (2000) 106–117
9. Carlsson, M., Hallgren, T.: FUDGETS - a graphical user interface in a lazy functional language. In: Proceedings of the ACM Conference on Functional Programming and Computer Architecture, Copenhagen, DK, FPCA '93, New York, NY, ACM (1993)
10. Carlsson, M., Hallgren, T.: Fudgets – Purely Functional Processes with applications to Graphical User Interfaces. PhD thesis, Chalmers University of Technology (1998) `http://www.cs.chalmers.se/~hallgren/Thesis/`.
11. Hanus, M.: A functional logic programming approach to graphical user interfaces. In: Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00). Volume 1753 of LNCS., Springer-Verlag (2000) 47–62
12. Thiemann, P.: WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages. Volume 2257 of LNCS., Springer-Verlag (2002) 192–208
13. The XForms home page can be found at `http://www.w3.org/MarkUp/Forms/`.
14. Achten, Peter, van Eekelen, Marko and Plasmeijer, Rinus: Generic Graphical User Interfaces. In Greg Michaelson, Phil Trinder, eds.: Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03. Volume 3145 of LNCS., Edinburgh, UK, Springer (2003)
15. Alimarine, A., Plasmeijer, R.: A Generic Programming Extension for Clean. In Arts, T., Mohnen, M., eds.: The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers. Volume 2312 of LNCS., Älvsjö, Sweden, Springer (2002) 168–186
16. 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., Springer (2004) 39–55