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}@science.ru.nl, jankuper@cs.utwente.nl`

Abstract. This paper presents FunctionalForms, a new 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 a medium-level GUI toolkit. In order to lift this toolkit's advanced layout combinators into our library, and at the same time obtain full separation between a form's layout and the data structure it edits, we introduce a technique called *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 toolkit. For larger forms, the code tends to get monolithic, badly readable and inflexible.

In this paper we present the combinator library FunctionalForms, built on top of the GUI toolkit 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 a function 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 of our framework: we have developed a general technique called *compositional functional references* 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 toolkit for Haskell is wxHaskell[1], an interface to the extensive cross-platform C++ toolkit 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.[3] It illustrates the problems of programming 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[4] and the four controls to populate it (lines 4–11). For every object, a pointer (`pdialog`, `pentry`, . . .) is returned. Controls have dynamic properties (*state*, as you wish), which can be manipulated by the user and/or the program during their lifetime. In particular, the dynamic properties `text` and `selection` (on the `entry` and `choice` control, resp.) are set[5] to the form's initial values (contained in `i_door`). We have to convert the `Bool` value into an `Int` first.
- In lines 13–19, the dialog's layout is specified. The function `widget` creates layout information from a control pointer; the combinators `margin`, `column`, `row` 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;

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

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

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

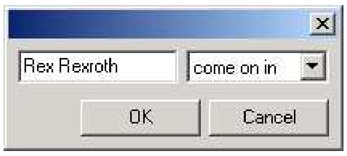


Fig. 1. Door information form

```
doorForm parent_window i_door =
   do let (i_name,i_avail) = i_door

      -- create dialog and controls
      pdialog <- dialog parent_window []
      pentry  <- entry pdialog [text := i_name]
      pchoice <- choice pdialog
            [ items := ["come on in","do not disturb"]
            , selection := bool2int i_avail ]
      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 get_f_door =
         do f_name <- get pentry text
            f_avail <- liftM int2bool $ get pchoice selection
            return (f_name, f_avail)
      let setclose close =
         do set pok [on command :=
               do f_door <- get_f_door; close $ Just f_door]
            set pcancel [on command := close Nothing]

      -- run dialog
      m_door <- showModal pdialog setclose
      return $ case m_door of
         Just f_door -> f_door
         Nothing     -> i_door

   where
   bool2int b = if b then 0 else 1
   int2bool i = (i==0)
```

Fig. 2. wxHaskell code for door information form actual coordinates are determined by wxHaskell's layout system, which also takes care of resizing controls.

– Next, 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[6] 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 `i_door` 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, 40 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` property of control `pentry` is set to the pair's first element, and the `selection` property of `pchoice` is set to the second.
   (ii) In the button event handler, the values of the same two properties 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 we are programming at a level which is too low. In the next section, we design a combinator library to abstract over these low-level details.

[6] 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 describes both the smallest (atomic) and largest parts of the constructed system; combinators combine and transform these parts.

The abstraction captured by this data type can be informally described as follows: we consider *any part of the GUI that is only able to display and alter a certain value* as a form. A form has a certain lifetime: at the beginning, it is provided by its environment with an *initial value*; subsequently, the user can read and alter this value; at the end, the form passes its *final value* back to the environment. We call the type of the edited value the *subject type* of the form.

In Fig. 3, some *atomic forms* and their subject types are shown. There is a rough correspondence between an atomic form and a *control* in wxHaskell; the form looks like the control, and its subject type corresponds to one of the control's dynamic properties.

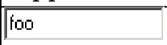
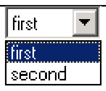
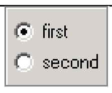
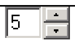
Name	**Appearance**	**Subject type**	**wxHaskell property**
`entry'`		`String`	`text`
`choice'`		`Int`	`selection`
`radioBox'`		`Int`	`selection`
`checkBox'`		`Bool`	`checked`
`spinCtrl'`		`Int`	`selection`

Fig. 3. Some atomic forms and their subject types

However, while interacting with a control in general does not have a fixed semantics (depending on the event handler, a single mouse move could send an email, play a movie or format your hard disk), interacting with a form does: you are merely editing a value. The only way for a form to influence its system environment is through its final value; conversely, the only influence of the environment on the form is through its initial value. This deliberate restriction—which still allows for many realistic forms—greatly simplifies the `FForm` data type and combinators.

Forms can be combined into larger forms: taken together, an `entry'` and a `checkBox'` edit a composite value (containing a `String` and a `Bool`). The *OK* and *Cancel* button in a form dialog are not considered forms but rather, together with the dialog itself, as part of the standard environment to *run* a form in. Everything else in the dialog can then be regarded as one composite form.

3.1 Defining the Data Type and Combinators

To formally capture the abstraction we introduce a type synonym for forms:

```
type FForm t w =
   Window w -> t -> IO (Layout, IO t)
```

The top-level IO action depends on a pointer to a parent window, in which the form is created, and on some initial value of its subject type `t`. It creates the control(s), then returns a `Layout` value for the form and an IO action that, at the moment it is run, retrieves the form's current value. We use this when the dialog is closed with the OK button, so we refer to it as the *final* value. As an example (atomic) form, we show the definition of `entry'`:

```
entry' :: FForm String w
entry' = \w init ->
   do pentry <- entry w [text := init]
      return (widget pentry, get pentry text)
```

Like in Sect. 2.1, wxHaskell's `entry` creates the control with its `text` property set to `init`, and returns the pointer `pentry`; `widget` creates a basic layout for the control and `get` returns the final value of the control's `text` property.

Other atomic forms, like `choice'` and `checkBox'`, are defined analogously. The remainder of our library consists of the functions `*-`, `*|`, `run_in_dialog`, `convert` and `convertL`.

The combinator `*-` joins two forms into one. This entails joining the subject types as well as the `Layout` values. In Sect. 3.2, we will identify this as the major cause of trouble for our library, but for now, we define:

```
(*-) :: FForm t1 w -> FForm t2 w -> FForm (t1,t2) w
form1 *- form2 = \w (init1,init2) ->
   do (lay1,getfin1) <- form1 w init1
      (lay2,getfin2) <- form2 w init2
      return (row 5 [lay1,lay2], liftM2 (,) getfin1 getfin2)
```

As the type signature shows, the composite form's subject type is a pair of its components' subject types. Its initial value `(init1,init2)` is split up and fed to the two component forms; likewise, the two final values are joined back into a tuple (we lift the tuple constructor into the IO monad).

Regarding layout, both components are put next to each other with a five-pixel gap in between. This choice is arbitrary, but it suffices for demonstration purposes. As a vertical alternative to `*-`, we define `*|`. It is equivalent, except that it uses `column` instead of `row`.

Now, using only `*-`, `*|`, 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 = entry' *| (checkBox' *- checkBox')
```

To use this form in a program, we would provide it with an initial value `init_val` of type `(String,(Bool,Bool))` and run it:

```
do ...
   final_val <- run_in_dialog parent_window composite init_val
   ...
```

The function `run_in_dialog`, 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 `run_in_dialog`'s IO action equals the form's final value if the *OK* button is used, and the initial value otherwise. We omit the code; it is very similar to the corresponding fragments in Fig. 2.

As the last addition to the combinator library, we define the function `convert` and its specialization `convertL`. They transform a form's subject type into an 'isomorphic' type, given the corresponding bijection. For example, this makes it possible to use a `choice'` form with two choices (which originally has subject type `Int` with domain $\{0, 1\}$) to edit a `Bool`, just like in Sect. 2.1.

```
type Bijection a b = (a->b, b->a)

convert :: Bijection t1 t2 -> FForm t2 w -> FForm t1 w
convert (forth,back) form = \w init ->
   do (lay,getfin) <- form w $ forth init
      return (lay, liftM back getfin)
```

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 `convertL` then maps the first value to 0, the second to 1, etc.:

```
convertL :: Eq t => [t] -> FForm Int w -> FForm t w
convertL items = convert (forth,back)
   where
   forth a = fromJust $ elemIndex a items
   back i = items!!i
```

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

```
doorForm = entry' *- availForm
   where
   availForm = convertL [True,False] $
      choice' [items := ["come on in","do not disturb"]]
```

3.2 Evaluation of the Combinator Library

The combinator library we defined solves 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. In fact, a similar technique is briefly mentioned in [1] (section *Safety*). However, the library has two disadvantages:

Crude layout combinators: Compared to wxHaskell's sophisticated layout system, `*-` and `*|` provide only very crude control over layout. The obvious solution is to introduce more combinators, which mimic wxHaskell's layout combinators. Unfortunately, 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 `()`.
- Combinators which join a list of `Layouts`, 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. We could solve this problem by extending the `FForm` type to also accommodate *lists* of `Layouts`, and introducing the combinators `nilF` and `consF` to produce such forms, but then our `doorForm` example would turn into

```
doorForm = row' 5 $ entry' `consF` (availForm `consF` nilF)
```

  and have subject type `(String,(Bool,()))`. This is rather cumbersome.

Dependency between layout and subject type: 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 `(String,Bool)` to `(Bool,String)`. One way to hack around this would be to `convert` the new form's subject type back:

```
convert (mirror,mirror) (availForm *- entry')
   where mirror (a,b) = (b,a)
```

. . . 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: they construct both the layout structure and the subject type structure, thereby creating dependencies between two structures which are, in essence, largely unrelated. In the next section, we show how to free the combinators from constructing the subject type.

4 Using Compositional Functional References

This section presents the second stage of FunctionalForms, in which the *point-free* combinator library is transformed into a *point-ful* one: using a concept we call *compositional functional references*, we shift the task of (de)composing the subject type from the combinators to the programmer. As a result, the subject type disappears from the `FForm` type; all the forms in a combinator expression work on a shared *context type*. However, every form edits only its own piece of this type: a *reference value* defines which piece this is. This allows us to write

```
declare2 $ \(ref_name,ref_avail) ->
   row' 5 [availForm ref_avail, entry' ref_name]
```

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. In this example, `ref_name` and `ref_avail` are reference values, generated by the function `declare2`.

4.1 Introducing Compositional Functional References

The key to our technique is the following type for references:

```
data Ref cx t =
   Ref (cx->t) ((t->t)->(cx->cx))
```

In this definition, type variable `cx` denotes the type of the *context*, a larger structure of values; `t` denotes the type of the component which is referenced.

A reference value consists of two functions; we will often bind these to identifiers such as `val` (for *value*) and `app` (for *apply*). The former retrieves the value from the context, whereas the latter updates the value in the context by applying its argument function (of type `t->t`) to the current value. An example will clarify the usage of these functions.

Suppose that the context is a tuple of type `(a,b)`. Now we can define two references: `ref1` refers to the first element, and `ref2` to the second.

```
ref1 :: Ref (a,b) a
ref1 = Ref fst appfst

ref2 :: Ref (a,b) b
ref2 = Ref snd appsnd

appfst f (x,y) = (f x, y)
appsnd f (x,y) = (x, f y)
```

Using `(Ref val2 app2) = ref2`, we can retrieve or update the integer in the initial context `initcx = ("foo",39)`:

`val2 initcx`	reduces to	`39`
`app2 (+3) initcx`	reduces to	`("foo",42)`
`app2 (const 1) initcx`	reduces to	`("foo",1)`

Note that when we partially apply `app2` by removing `initcx` in the last two examples, we obtain a function of type `cx->cx`: a context update. We include such a function in the new `FForm` type, which we now present.

Point-free forms	**Forms with references**
`type FForm t w =`	`type FForm cx w =`
`  Window w -> t ->`	`  Window w -> cx ->`
`    IO (Layout, IO t)`	`    IO (Layout, IO (cx->cx))`
`entry' :: FForm String w`	`entry':: Ref cx String-> FForm cx w`
`entry' = \w init ->`	`entry' (Ref val app) = \w initcx ->`
` do pentry <- entry w`	` do pentry <- entry w`
`              [text := init]`	`              [text := val initcx]`
`    return`	`    return`
`      ( widget pentry`	`      ( widget pentry`
`      , get pentry text )`	`      , do t <- get pentry text;`
	`           return $ app $ const t )`
`(*-) :: FForm t1 w -> FForm t2 w ->`	`(.-) :: FForm cx w -> FForm cx w ->`
`         FForm (t1,t2) w`	`         FForm cx w`
`fm1 *- fm2 = \w (init1,init2) ->`	`fm1 .- fm2 = \w initcx ->`
` do (lay1,getfin1) <- fm1 w init1`	` do (lay1,getupd1) <- fm1 w initcx`
`    (lay2,getfin2) <- fm2 w init2`	`    (lay2,getupd2) <- fm2 w initcx`
`    return`	`    return`
`      ( row 5 [lay1,lay2]`	`      ( row 5 [lay1,lay2]`
`      , liftM2 (,) getfin1 getfin2 )`	`      , liftM2 (.) getupd1 getupd2 )`
`convert :: Bijection t1 t2 ->`	`convert :: Bijection t1 t2 ->`
`   FForm t2 w -> FForm t1 w`	`  (Ref cx t2->fm) -> (Ref cx t1->fm)`

Fig. 4. Transforming the combinator library

4.2 Forms with References

We transform the library so that every form has access to the same context, whose type equals the subject type of the topmost form composition. An overview of the changes is shown in Fig. 4. The new `FForm` type clearly shows that a form no longer depends on an initial *value* for itself, but rather on an initial *context*; and that instead of producing a final *value*, it produces a final *context update*. In the *OK* button event handler this update is applied to the initial context.

As the new definition of `entry'` shows, the programmer now provides an atomic form with a reference value. This determines which part of the context it edits: the `val` component retrieves an initial value from this part and the `app` component 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 `*-` is replaced by `.-` . 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).

This effectively solves the first problem from Sect. 3.2. Since the arguments of `.-` are of the same type now, it can be easily generalized to take a list of forms instead of two (and a margin width value), thereby providing the lifted version `row'` of the 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 combinators like `label`.

The second problem is also solved: the operands of `.-` (and for `row'`: 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, following the example of `row'`, we were able to lift all of wxHaskell's layout combinators into the `FForm` domain.[8] However, for simplicity's sake, we will still restrict our use of combinators to `.-` and `row'` in the rest of this section.

4.3 Constructing the Subject Type with References

Since the combinators do not construct the subject type anymore, we have to do it ourselves, using references. First, we show how we can derive any reference value for a context type consisting of nested pairs. Suppose that in the context type `(a,b)` which we introduced before, `a` itself is a pair type `(a1,a2)`. As the reference `ref12` to the value of type `a2`—in other words, the second element of the first element of the whole context—we define:

```
ref12 :: Ref ((a1,a2),b) a2
ref12 = Ref (snd . fst) (appfst . appsnd)
```

It is easily seen how this leads to a general scheme for manually defining an arbitrary reference value: for the *value* part, we compose a sequence of `fst` and `snd` functions, and for the *apply* part, we compose the reverse sequence of `appfst` and `appsnd` functions. This can be captured in an inductive solution, where we derive two new references from a reference to a pair:

```
splitref2 :: Ref cx (t1,t2) -> (Ref cx t1, Ref cx t2)
splitref2 (Ref val app) = (ref1, ref2)
   where
   ref1 = Ref (fst . val) (app . appfst)
   ref2 = Ref (snd . val) (app . appsnd)
```

With this function, we can derive the reference value `ref12` like this:

```
idref = Ref id id
(ref1,ref2) = splitref2 idref
(ref11,ref12) = splitref2 ref1
```

The advantage of using `splitref2` is the ability to create references relative to other references. This can be used to combine atomic forms into a larger form which is also parameterized by a single reference. We illustrate this by means of our `doorForm` example, rewritten for the transformed library:

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

[8] When we structure the `FForm` domain as a monad (which we avoided in this paper) we can even use the monadic lifting functions for this purpose.

```
doorForm :: Ref cx (String,Bool) -> FForm cx w
doorForm ref_door =
   entry' ref_name .- availForm ref_avail
   where
   (ref_name,ref_avail) = splitref2 ref_door
```

To shorten this definition, we introduce the function `declare2`:

```
declare2 refs_to_form = refs_to_form . splitref2
```

. . . so that we can rewrite the example into:

```
doorForm = declare2 $ \(ref_name,ref_avail) ->
   entry' ref_name .- availForm ref_avail
```

Note that we have omitted the definition of `availForm`. In fact, it is defined exactly the same as in Sect. 3. However, just like the atomic forms and `doorForm`, it is now also parameterized by a reference; in the transformed library, `convert` and `convertL` operate on values of the type `Ref ... -> FForm ...` (they leave the form itself alone and convert only the reference value).

The same goes for the new version of `run_in_dialog`. It applies a parameterized form to the root reference `idref :: Ref a a`. This is what equates the context type of every form to the subject type of this topmost form.

4.4 Reference Values for Other Subject Types

Up to this point, we have restricted the subject types to (nested) pairs. We can easily extend them to tuples of higher arity n by defining `splitref`n and `declare`n, following the same scheme we used for `splitref2` and `declare2`.[9] Perhaps more suprisingly, we can also use a similar scheme to define a function which splits a reference to a list into an infinite list of references to its elements:

```
splitrefL :: Ref cx [t] -> [Ref cx t]
splitrefL (Ref val app) = refhead:(splitrefL reftail)
   where
   refhead = Ref (head . val) (app . apphead)
   reftail = Ref (tail . val) (app . apptail)
   apphead f (x:xs) = (f x):xs
   apptail f (x:xs) = x:(f xs)

declareL refs_to_form = refs_to_form . splitrefL
```

Furthermore, we generalize `declare2` to the operator `.*.` :

```
(f1 .*. f2) refs_to_form ref =
   f2 (f1 (curry refs_to_form) ref1) ref2
   where (ref1,ref2) = splitref2 ref
```

[9] It is tempting to think that these functions can be defined using *generic programming*[5, 6], but our attempts have not been fruitful thus far.

While `id.*.id` equals `declare2`, repeated application of the operator splits the reference value more than once, without the need to name intermediate values. For example, a reference to a type `((a,(b,c)),d)` can be split into four references:

```
((id.*.(id.*.id)).*.id) $ \((ref1,(ref2,ref3)),ref4) -> ...
```

Note that `declareL` can also be used as an operand to `.*.` . The use of the functions defined in this section is illustrated in the following example.

5 Elaborate Example

The example of form programming with FunctionalForms that we now present shows its declarative style and power. While we have thus far kept the library as simple as possible for clarity, we use a more realistic version here. First, to make use of the full potential of wxHaskell's control creation function, every atomic form is extended with a property list, which it passes on. Second, we have lifted some of wxHaskell's layout combinators—namely `row`, `column`, `grid`, `margin`, `floatLeft` and `label`—into the `FForm` domain (with an apostrophe appended to their names), as explained in section 4.2. Both adaptations were fairly trivial.

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. 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 `checkBox'`, a `timeForm` and an `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

FunctionalForms adds a declarative abstraction layer (albeit restricted to forms) to wxHaskell, which can be regarded as a medium-level, procedural GUI toolkit. Many other functional GUI toolkits exist (see [1] for an overview), some of which are also more declarative. A well-known example of these is Fudgets[7].

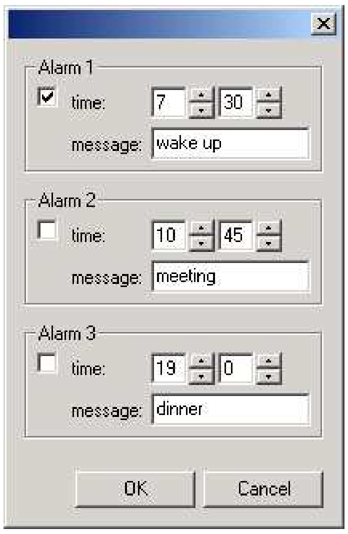
Fudgets are compositional stream based elements that transform input values to output values using a GUI. Stream composition is done via combinators, which

```
alarmListForm :: Ref cx [(Bool,(Int,String))] -> FForm cx w
alarmListForm = declareL $ \refs ->
   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 = (id.*.(id.*.id)) $ \(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 $ \(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



Alarm 1
☑ time: 7 30
message: wake up

Alarm 2
☐ time: 10 45
message: meeting

Alarm 3
☐ time: 19 0
message: dinner

Fig. 6. Appearance of alarm list form

```
module Alarms(main) where

import Graphics.UI.WX
import FForms

main = start $
   do f <- frame []
      f_alarms <- run_in_dialog
         f alarmListForm i_alarms
      print f_alarms
      close f

i_alarms =
   [ (True, (450, "wake up"))
   , (False,(645, "meeting"))
   , (False,(1140,"dinner"))
   ]
```

Fig. 7. Startup code for alarm list form

also take care of the relative layout of their arguments. It is worthwhile to point out that 'raw' fudgets suffer from a bad separation between the look of a program and its combinatorial structure, just like the first stage of FunctionalForms. Two techniques have been added[8] to solve these issues:

- The programmer can apply a transformation (such as *reverse*) to the composite fudget's layout. This transformation is the converse of the `convert` 'hack' we showed in Sect. 3.2: there, the combinator structure follows the layout structure, and the composite subject type is transformed.
- A more flexible approach is that of associating an identifier value with the sub-fudgets. These values can be used in a *name layout* expression which is applied to the composite fudget. In a sense, this is similar to the references in the second stage of FunctionalForms. However, while the former name sub-layouts, the latter 'name' data sub-structures. We believe that from a top-down design perspective, it is more natural to name the data structures, because they are designed first and are less susceptible to change.

Naming data structures is also the approach taken by XForms[9], a recent W3C standard to define Web forms. A definition has two parts: the *XForms Model* defines the structure of the edited data and provides every element with a name, an initial value and possibly type or value constraints. In the *XForms User Interface*, GUI controls are bound to these elements by referring to their names.

Generic *Graphical Editor Components* (GECs)[10] use their 'subject type' to convey layout information. A generic function[6] 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*[11] 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 library which facilitates the programming of forms in a functional language. First we built a combinator library capturing the form abstraction. This solved the problems of low-level programming, such as verbosity and inflexibility, but had a drawback: it coupled subject type structure and layout combinator structure together. Then we introduced compositional functional references to release this dependency; this allowed us to exploit the full power of the layout combinators from the underlying GUI toolkit 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.

A major advantage of our technique is that it does not depend on a special GUI toolkit 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 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.

8 Acknowledgements

The authors would like to thank Marko van Eekelen and Rinus Plasmeijer 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) 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
4. The wxWidgets home page can be found at `http://www.wxwidgets.org`.
5. Clarke, D., Löh, A.: Generic Haskell, Specifically. In Gibbons, J., Jeuring, J., eds.: Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, Kluwer Academic Publishers (2003) 21–48 ISBN 1-4020-7374-7.
6. 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
7. 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)
8. 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/`.
9. The XForms home page can be found at `http://www.w3.org/MarkUp/Forms/`.
10. 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. LNCS, Edinburgh, UK, Springer (2003) To appear: draft version available via `ftp://ftp.cs.kun.nl/pub/Clean/papers/2004/achp2004-GenericGUI.pdf`.
11. 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