Chapter 1

Disjoint Forms in Graphical User Interfaces

Sander Evers, Peter Achten, Rinus Plasmeijer[1]

Abstract: Forms are parts of a graphical user interface (GUI) that show a (structured) value and allow the user to update it. Some forms express a choice between two or more (structured) values using radio buttons or check boxes. We show that explicitly modelling such a choice leads to a cleaner separation of logic and layout. This is done by extending the combinator library FunctionalForms with *disjoint form combinators*. To implement these, we have generalized the technique of compositional functional references which underlies the library.

1.1 INTRODUCTION

Forms are parts of a graphical user interface (GUI) that show a (structured) value and allow the user to update it. For example, the omnipresent dialogs labeled *Options*, *Settings* and *Preferences* are forms. An address book can also be considered a form. In our previous work, we have developed the combinator library FunctionalForms[2] for building forms in a concise, compositional way.

Many real-life forms allow a choice between two or more alternatives, some of which require extra information. For example, the form in Fig. 1.1 indicates whether the user wishes to receive a certain newsletter; if s/he does, the text entry field next to this option should contain his/her email adress. If s/he does not, this text field is irrelevant (some GUIs provide a visual clue for this: the control is dimmed).

Usually, the information in such a form is processed as a product-like data structure containing the choice (e.g. as a boolean) and the extra information (e.g. as a string). However, most functional languages allow data types which are more suited to this task, namely *disjoint union types*. In Haskell[4], we would define

[1]Radboud University Nijmegen, Department of Software Technology, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. {s.evers,p.achten,rinus}@cs.ru.nl

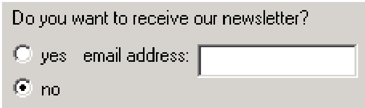

FIGURE 1.1. A 'disjoint' form

$$\textbf{data}\ \mathit{NewsLetter} = \mathit{NewsYes}\ \mathit{String} \mid \mathit{NewsNo}$$

for the type of information in the example form.

While the combinators in FunctionalForms previously only supported forms with product-like data structures, in this paper we extend them to enable the explicit definition of such a *disjoint form*. Rather than as a 'yes/no' form *and* an 'email' form, it can now be composed as a 'yes, email' form *or* a 'no' form. We demonstrate that this technique leads to a better separation of logic and layout in disjoint forms. For its implementation, we have generalized the *compositional functional references* which underlie the library.

This paper is organized as follows: it first gives a summary of the library's basic use, which has not changed (Sect. 1.2). Then, the use and merits of the extension are demonstrated (Sect. 1.3), after which its implementation is discussed in Sect. 1.4. Next, we show that the gained flexibility leads to some safety issues (Sect. 1.5). We finish with related work (Sect. 1.6) and conclusions (Sect. 1.7).

1.2 FUNCTIONALFORMS SUMMARY

FunctionalForms[2] is a combinator library built on the GUI library wxHaskell[7] (itself built on the cross-platform C++ library wxWidgets[11]). It can be seen as an embedded domain-specific language for forms: it consists of *atomic forms* and ways to combine them into larger forms in a declarative style. In this section, we give a brief summary of its basic use, which is the same as described in [2], although the types have changed a little.

A *form* is a GUI part, residing somewhere within a dialog with *OK* and *Cancel* buttons, which is only able to display and alter a certain value. When the 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* back to the environment. The type of this value is called the *subject type* of the form.

Atomic forms correspond to a single control containing an editable value. Examples are a text entry field, containing a *String*, and a spin control, containing an *Int*:

$$\begin{array}{l}\underline{\mathit{entry}'} :: \mathit{Monad}\ m \Rightarrow \\ \quad [\mathit{Prop}\ (\mathit{TextCtrl}\ ())] \rightarrow \mathit{Ref}\ m\ \mathit{String} \rightarrow \mathit{FForm}\ \mathit{win}\ m\ \mathit{Layout} \\ \underline{\mathit{spinCtrl}'} :: \mathit{Monad}\ m \Rightarrow \\ \quad \mathit{Int} \rightarrow \mathit{Int} \rightarrow [\mathit{Prop}\ (\mathit{SpinCtrl}\ ())] \rightarrow \mathit{Ref}\ m\ \mathit{Int} \rightarrow \mathit{FForm}\ \mathit{win}\ m\ \mathit{Layout}\end{array}$$

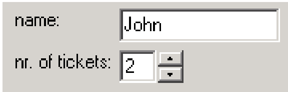

FIGURE 1.2. ***ticketsForm***

We follow the convention that library functions are underlined, and an atomic form is named after the corresponding wxHaskell function which creates its control, but with an additional prime symbol. Every atomic form is parameterized with a list of optional properties used for customizing the control, e.g. its size and font (leaving this list empty produces reasonable defaults). Some atomic forms, like *spinCtrl′*, require additional parameters: its first two arguments indicate a minimum and maximum value.

All these parameters actually have little to do with FunctionalForms: they are directly passed to the wxHaskell control creation function. In contrast, the last parameter of both forms is specific to FunctionalForms; it is a *reference value* which relates the atomic form's subject type (*String* and *Int*, resp.) to the subject type of the top-level form. A more detailed description of the *Ref* and *FForm* types is postponed to Sect. 1.4.

To combine atomic forms into larger forms, two aspects have to be composed: layout and subject type. The former is performed by *layout combinators* like *grid′*, *margin′* and *floatLeft′*. These are based on the wxHaskell layout combinators after which they are named, but operate directly on forms.[1] For example, the two atomic forms can be put in a grid layout with some labels (see Fig. 1.2):

```
grid′ 5 5 [ [ label′ "name :",            entry′ [] name      ]
          , [ label′ "nr. of tickets :",  spinCtrl′ 1 6 [] nr ]
          ]
```

Note that the two reference values $(name :: Ref\ m\ String)$ and $(nr :: Ref\ m\ Int)$ are free variables in this expression. Also, this form's subject type is not yet established. To complete the form composition, *name* and *nr* are bound in a lambda expression, onto which a *subject type combinator*, namely *declare2*, is applied:

```
ticketsForm :: Monad m ⇒ Ref m (String, Int) → FForm win m Layout
ticketsForm = declare2 $ λ(name, nr) →
  grid′ 5 5 [ [ label′ "name :",            entry′ [] name      ]
            , [ label′ "nr. of tickets :",  spinCtrl′ 1 6 [] nr ]
            ]
```

This 'declares' *ticketsForm*'s subject type to be $(String, Int)$, as witnessed by its actual type declaration (which can be omitted). Just like the atomic forms, *ticketsForm* can now be used as a component of a larger form. Note how this two-stage process of form construction separates the definition of layout and subject type structures, providing a great deal of freedom to the library user (see also [2]).

[1] instead of on *Layout* values of widgets—for those familiar with wxHaskell

FIGURE 1.3. ***contactForm***$_1$

Besides *declare2*, which declares a pair, the library also provides subject type combinators for tuples of higher arity and for lists.

$$
\begin{aligned}
&\underline{declare2} :: Monad\ m \Rightarrow \\
&\quad ((Ref\ m\ t_1,\ Ref\ m\ t_2) \to z) \to Ref\ m\ (t_1, t_2) \to z \\
&\underline{declare3} :: Monad\ m \Rightarrow \\
&\quad ((Ref\ m\ t_1,\ Ref\ m\ t_2,\ Ref\ m\ t_3) \to z) \to Ref\ m\ (t_1, t_2, t_3) \to z \\
&\ldots \\
&\underline{declareL} :: Monad\ m \Rightarrow \\
&\quad ([Ref\ m\ t] \to z) \to Ref\ m\ [t] \to z
\end{aligned}
$$

The *declareL* combinator only composes forms for the list *elements* and cannot alter the spine; it produces a form for lists of a fixed length.

To run a form in a wxHaskell program, the library function *runInDialog* is used. For example, this runs the above defined *ticketsForm* with *John* and *2* in the atomic forms:

$$
\begin{aligned}
&\mathbf{do}\ \ldots \\
&\quad (newname, newnr) \leftarrow \underline{runInDialog}\ parentWindow\ ticketsForm\ (\text{"John"}, 2) \\
&\quad \ldots
\end{aligned}
$$

The function takes as its arguments a pointer to a parent window, the form itself, and an initial value of the form's subject type. It returns an IO action, which produces a modal dialog containing the form and *OK/Cancel* buttons. When the user presses *OK*, the return value is bound to altered value in the form; if *Cancel* is pressed, the initial value is returned instead. After this, the IO thread continues.

1.3 COMBINATORS FOR DISJOINT FORMS

This section describes, from a library user's point of view, the additions for defining disjoint forms. As an example, we will define a form for contact information, depicted in Fig. 1.3. It has subject type

$$\mathbf{data}\ Contact = ByPhone\ Phone \mid ByEmail\ String \mid NotAtAll$$

and expresses a choice between a phone number, an email address and no information at all. Before we can start defining the form itself, we need to define three custom subject type combinators for this type's data constructors. This is done using a Template Haskell [8] macro named *declare*, which is included in the library.

$$\begin{aligned}
declareByPhone &= \$(\underline{declare}\ [|ByPhone|]\ 1)\\
declareByEmail &= \$(\underline{declare}\ [|ByEmail|]\ 1)\\
declareNotAtAll &= \$(\underline{declare}\ [|NotAtAll|]\ 0)
\end{aligned}$$

For each of the constructors, we provide the macro with its name and arity. The delimiters $\$(\ldots)$ and $[|\ldots|]$ are Template Haskell syntax, which the library user does not need to worry about.[2]

The three fresh subject type combinators are used to turn forms with subject types (resp.) *Phone*, *String* and no subject type at all into forms with subject type *Contact*. Their type signatures are:

$$\begin{aligned}
declareByPhone &:: Monad\ m \Rightarrow\\
&\quad (Ref\ m\ Phone \rightarrow z) \rightarrow Ref\ m\ Contact \rightarrow z\\
declareByEmail &:: Monad\ m \Rightarrow\\
&\quad (Ref\ m\ String \rightarrow z) \rightarrow Ref\ m\ Contact \rightarrow z\\
declareNotAtAll &:: Monad\ m \Rightarrow\\
&\quad FForm\ win\ m\ l \rightarrow Ref\ m\ Contact \rightarrow FForm\ win\ m\ l
\end{aligned}$$

In the last type signature, the type *FForm win m l* plays the same role as *z* in the above signatures. The reason why it is more constrained is that *declareNotAtAll* appends its argument form with an invisible form for handling the *NotAtAll* value.

Using these subject type combinators, $contactForm_1$ can be defined as follows; we assume $(phoneForm :: Ref\ m\ Phone \rightarrow FForm\ win\ m\ Layout)$ is defined somewhere else:

$$\begin{aligned}
contactForm_1 &= \underline{radioGrid}\ [byPhone, byEmail, byNothing]\\
byPhone &= declareByPhone\ \$\ \lambda phone \rightarrow\\
&\quad \underline{row'}\ 5\ [\underline{label'}\ \text{"by phone :"}, phoneForm\ phone]\\
byEmail &= declareByEmail\ \$\ \lambda email \rightarrow\\
&\quad \underline{row'}\ 5\ [\underline{label'}\ \text{"by email :"}, \underline{entry'}\ []\ email]\\
byNothing &= declareNotAtAll\ \$\\
&\quad \underline{label'}\ \text{"do not contact me"}
\end{aligned}$$

The new disjoint form combinator *radioGrid* arranges its list of subforms into a grid layout, with radio buttons to the left of them. Due to their subject type combinators, the three subforms have the same subject type as the composite form (*Contact*), but each only 'works' for a particular data constructor. For example, the *byEmail* form only handles *ByEmail* values. This means that when $contactForm_1$ is run with an initial *ByEmail* value, the middle radio button is selected, and only the text field next to it receives an initial value. The other text

[2] Template Haskell is a GHC compiler extension for meta-programming, i.e. programmatically manipulating a program at the syntactic level. The delimiters turn a meta-language expression into an object-language expression and vice versa. Both object language and meta-language are Haskell.

FIGURE 1.4. ***contactForm***$_2$

FIGURE 1.5. ***contactForm***$_3$

field is left empty (or contains a default value, if the programmer has specified this in *phoneForm*). When the form is closed, every subform contains a final value with its particular *Contact* data constructor, but only one of them is promoted to $contactForm_1$'s final value; this choice is determined by the radio button selected at that time.

What is the advantage of using the disjoint form combinator $\underline{radioGrid}$, apart from stylistic arguments? Consider the alternative case, in which the form in Fig. 1.3 is defined as a *conjunction* of a $\underline{radioBox'}$ (with an *Int* for three possible choices), a *phoneForm* and an $\underline{entry'}$; its subject type would be $(Int, Phone, String)$. At some later time, the interaction design department decides the form should rather look like Fig. 1.4 or like Fig. 1.5. Note that these forms still express exactly the same choice. However, when the form code is changed accordingly, its subject type would be $(Int, Int, Phone, String)$ or $(Bool, Int, Phone, String)$, and the code which handles the form data should also be altered.

If we use disjoint forms instead, the disjoint subject type can remain the same. In the code, we only need to add an extra $\underline{radioGrid}$ for the first case:

$$
\begin{aligned}
contactForm_2 &= \underline{radioGrid} \\
&\qquad [noContact, \lambda yes \rightarrow \underline{row'}\ 5\ [\underline{label'}\ \text{"yes"}, yesContact\ yes]] \\
noContact &= declareNotAtAll\ \$\ \underline{label'}\ \text{"no"} \\
yesContact &= \underline{radioGrid}\ [byPhone, byEmail]
\end{aligned}
$$

For the second case, we use another disjoint form combinator, namely $\underline{checkRow}$:

$$
\begin{aligned}
&contactForm_3 = \underline{checkRow} \\
&\quad (\lambda yes \rightarrow \underline{column'}\ 5\ [\underline{label'}\ \text{"}Please\ contact\ me\text{"}, yesContact\ yes]) \\
&\quad (declareNotAtAll\ \underline{noLayout})
\end{aligned}
$$

The functionality of these forms is still the same: they display a value of their subject type *Contact*, and allow the user to change it into another value of that type.

1.4 IMPLEMENTATION

Although the user of FunctionalForms does not notice a difference, apart from the new combinators and slightly altered *Ref* and *FForm* types, the implementation of the library has undergone substantial changes since its first version in [2]. These allow for generalized forms, which may *fail* to consume an initial value (or produce a final value), and which can be joined with the disjoint form combinators <u>*radioGrid*</u> and <u>*checkRow*</u>. To construct these forms, the *compositional functional references* have also been generalized. Furthermore, the 'heart' of a form, which determines the communication with its environment, has been made explicit in a type *RefLink*. In order to deal with the new *FForm* type in Sect. 1.4.3, we will first discuss these *Ref* and *RefLink* types.

1.4.1 The *Ref* type

A reference value consists of two functions which are used to 'refer to' a *t* part of a—usually stateful—monad *m*:

$$\begin{array}{l}\textbf{data}\ \mathit{Ref}\ m\ t\ =\ \mathit{Ref}\ \{\ \mathit{val}\ ::\ m\ t\\ \qquad\qquad\qquad\qquad\quad ,\ \mathit{app} ::\ (t \to m\ t) \to m\ ()\\ \qquad\qquad\qquad\qquad\quad \}\end{array}$$

The *val* function retrieves the value of this particular part of the monadic state, whereas the *app* function updates it. For example, for a state of type (t_1,t_2), the value referring to the t_1 element would be:

$$\begin{array}{l}\mathit{reffst} :: \mathit{MonadState}\ (t_1,t_2)\ m\ \Rightarrow\ \mathit{Ref}\ m\ t_1\\ \mathit{reffst} = \mathit{Ref}\ \{\ \mathit{val}\ = \textbf{do}\ \{\ (x,y)\ \leftarrow\ \mathit{get};\ \mathit{return}\ x\ \}\\ \qquad\qquad\quad ,\ \mathit{app} = \lambda f \to\ \textbf{do}\ \{\ \tilde{}(x,y)\ \leftarrow\ \mathit{get};\ x'\ \leftarrow\ f\ x;\ \mathit{put}\ (x',y)\ \}\\ \qquad\qquad\quad \}\end{array}$$

Note the lazy pattern match in the *app* function; it is useful when constructing a new state from scratch (i.e. the previous state contains $\perp$).

A reference to the value in a *Just* data constructor (from the well-known *Maybe* type) can be defined in a very similar way:

$$\begin{array}{l}\mathit{reffromJust} ::\ \mathit{MonadState}\ (\mathit{Maybe}\ t)\ m\ \Rightarrow\ \mathit{Ref}\ m\ t\\ \mathit{reffromJust} = \mathit{Ref}\\ \quad \{\ \mathit{val}\ = \textbf{do}\ \{\ \mathit{Just}\ x\ \leftarrow\ \mathit{get};\ \mathit{return}\ x\ \}\\ \quad ,\ \mathit{app} = \lambda f \to\ \textbf{do}\ \{\ \tilde{}(\mathit{Just}\ x)\ \leftarrow\ \mathit{get};\ x'\ \leftarrow\ f\ x;\ \mathit{put}\ (\mathit{Just}\ x')\ \}\\ \quad \}\end{array}$$

This reference value may seem ill-defined because it can 'dangle': when the monadic state contains *Nothing*, it does not refer to anything. However, this situation can be detected using monadic error-handling, and the control flow can be adapted. We will show how this is done in Sect. 1.4.2, when we join two *RefLink*s.

The operator • composes two reference values, taking the referred part of the second value's state as the state of the first value. For example, the following value refers to the first element of a pair within a *Just* value:

$$\mathit{reffst}\ \bullet\ \mathit{reffromJust}\ ::\ \mathit{MonadState}\ (\mathit{Maybe}\ (t_1,t_2))\ m\ \Rightarrow\ \mathit{Ref}\ m\ t_1$$

The composition is performed by applying a monad transformer to the monad of the second reference value. This 'adds state' to this monad, on which the first reference value can act. Meanwhile, properties of the original monad such as IO ability or error handling are preserved.

$$
\begin{aligned}
&\bullet :: \mathit{Monad}\ m \Rightarrow \mathit{Ref}\ (\mathit{StateT}\ \mathit{cx}\ m)\ t \rightarrow \mathit{Ref}\ m\ \mathit{cx} \rightarrow \mathit{Ref}\ m\ t \\
&w \bullet v = \mathit{Ref} \\
&\quad \{ \mathit{val} = \mathit{val}\ v \ggg= \mathit{evalStateT}\ (\mathit{val}\ w) \\
&\quad , \mathit{app} = \lambda f \rightarrow (\mathit{app}\ v)\ \$\ \mathit{execStateT}\ \$\ (\mathit{app}\ w)\ \$\ \mathit{lift}\ .\ f \\
&\quad \}
\end{aligned}
$$

This operator is used to define subject type combinators like:

$$
\begin{aligned}
&\underline{\mathit{declare2}} :: \mathit{Monad}\ m \Rightarrow \\
&\quad ((\mathit{Ref}\ m\ t_1, \mathit{Ref}\ m\ t_2) \rightarrow z) \rightarrow \mathit{Ref}\ m\ (t_1, t_2) \rightarrow z \\
&\underline{\mathit{declare2}}\ \mathit{refsToForm}\ \mathit{refP} = \mathit{refsToForm}\ (\mathit{reffst} \bullet \mathit{refP}, \mathit{refsnd} \bullet \mathit{refP}) \\
\\
&\underline{\mathit{declareJust}} :: \mathit{Monad}\ m \Rightarrow (\mathit{Ref}\ m\ a \rightarrow z) \rightarrow \mathit{Ref}\ m\ (\mathit{Maybe}\ a) \rightarrow z \\
&\underline{\mathit{declareJust}}\ \mathit{refToForm}\ \mathit{refMaybe} = \mathit{refToForm}\ (\mathit{reffromJust} \bullet \mathit{refMaybe})
\end{aligned}
$$

These subject type combinators all follow the same pattern. This pattern is captured in the Template Haskell macro *declare*, so definitions like the two above do not have to be handwritten for every data constructor.

1.4.2 The *RefLink* type

The heart of an atomic form consists of a link between two reference values. The first is of type *Ref m t*, and relates the subject type *t* of this form to that of the topmost form (the state type in *m*). This is the reference value that is explicitly provided by the library user, e.g. in the expression $\underline{\mathit{entry}}'\ []\ \mathit{ref}_m$. The second reference value is implicit in every atomic form; it is of type *Ref IO t*, and relates this form's subject type to a part of the IO state. It is constructed from wxHaskell's *get* and *set* functions for the control's main attribute (e.g. *text* on an *entry* control).

In the terminology of the well-known model–view(–controller) paradigm[6], the reference values refer to a part of the topmost form's *model* and a part of its *view*, respectively. Joining them in a *RefLink* means linking those parts to each other: the *val* output from the first reference is used as *app* input for the second, and vice versa. Thus, two operations are obtained which both enforce consistency between model and view:

- The *update* operation copies the value from the form's model to its view. This is used to show the form's initial value.

- The *propagate* operation copies the value from the form's view to its model. This is used to read the form's final value.

In monadic terms, the *update* operation is a read action in the *m* monad producing a write action in the *IO* monad. Conversely, the *propagate* operation is a read action in the *IO* monad producing a write action in the *m* monad. This results in the following type for *RefLink* (where the *n* monad abstracts from *IO*):

$$
\begin{array}{l}
\textbf{data}\ \mathit{RefLink}\ m\ n\ =\ \mathit{RefLink} \\
\quad \{\ \mathit{update}\quad ::\ m\ (n\ ()) \\
\quad ,\ \mathit{propagate} ::\ n\ (m\ ()) \\
\quad \}
\end{array}
$$

The function *refLink* connects the two references to create such a *RefLink*. For the *update* function, first an input *v* is retrieved from the *m* reference. Then, a constant function $const\ (return\ v)$ is applied to the corresponding part in the *n* monad using the *n* reference; this action in the *n* monad is returned in the *m* monad. For the *propagate* function, the roles of *m* and *n* are reversed:

$$
\begin{array}{l}
\mathit{refLink}\ ::\ (\mathit{Monad}\ m,\ \mathit{Monad}\ n)\ \Rightarrow\ \mathit{Ref}\ m\ a\ \rightarrow\ \mathit{Ref}\ n\ a\ \rightarrow\ \mathit{RefLink}\ m\ n \\
\mathit{refLink}\ \mathit{ref}_m\ \mathit{ref}_n\ =\ \mathit{RefLink} \\
\quad \{\ \mathit{update}\quad =\ (\mathit{val}\ \mathit{ref}_m)\ \gg=\ \mathit{return}\ .\ (\mathit{app}\ \mathit{ref}_n)\ .\ \mathit{const}\ .\ \mathit{return} \\
\quad ,\ \mathit{propagate} =\ (\mathit{val}\ \mathit{ref}_n)\ \gg=\ \mathit{return}\ .\ (\mathit{app}\ \mathit{ref}_m)\ .\ \mathit{const}\ .\ \mathit{return} \\
\quad \}
\end{array}
$$

When two forms are joined, their *RefLink*s are combined into a new *RefLink*. Usually, the intention is that the joint *update* performs *both* component *update*s, and likewise for the *propagate*s. We consider this to be the 'default' operator on *RefLink*. In order to meet the *MonadWriter* interface (see Sect. 1.4.3), we encode it using the *Monoid* class:

$$
\begin{array}{l}
\mathit{instance}\ (\mathit{Monad}\ m,\ \mathit{Monad}\ n)\ \Rightarrow\ \mathit{Monoid}\ (\mathit{RefLink}\ m\ n) \\
\quad \textbf{where}\ \mathit{mempty}\ =\ \mathit{RefLink} \\
\qquad\qquad \{\ \mathit{update}\quad =\ \mathit{return}\ \$\ \mathit{return}\ () \\
\qquad\qquad ,\ \mathit{propagate} =\ \mathit{return}\ \$\ \mathit{return}\ () \\
\qquad\qquad \} \\
\qquad\ \ \mathit{rl}_1\ \text{`}\mathit{mappend}\text{`}\ \mathit{rl}_2\ =\ \mathit{RefLink} \\
\qquad\qquad \{\ \mathit{update}\quad =\ \mathit{liftM2}\ (\gg)\ (\mathit{update}\ \mathit{rl}_1)\ (\mathit{update}\ \mathit{rl}_2) \\
\qquad\qquad ,\ \mathit{propagate} =\ \mathit{liftM2}\ (\gg)\ (\mathit{propagate}\ \mathit{rl}_1)\ (\mathit{propagate}\ \mathit{rl}_2) \\
\qquad\qquad \}
\end{array}
$$

For disjoint forms, the two *RefLink*s should be joined in an alternative way. In this situation, they share one part of the model part, which is a disjoint union (e.g. the subject type of both forms is *Either a b*). Meanwhile, they refer to different parts of the view (which contains controls for both *a* and *b*). Hence, the two *RefLink*s connect independent parts of the view state space to 'competing' parts of the model state space. Instead of performing both *update* (*propagate*) operations, only one can (should) be performed.

We obtain this behaviour by using the *mplus* operator of the model monad *m*; therefore, this should be an instance of *MonadPlus*. The joint *update* will then (dynamically) choose between the first component *update* or the second—and likewise for *propagate*. Hence, we define an alternative monoid on the *RefLink* domain. By using a different representation for the *RefLink* type, the *Monoid* class can again be used for this:

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{RefLinkPlusM}\ m\ n = \mathit{RefLinkPlusM}\ \{\mathit{pm} :: \mathit{RefLink}\ m\ n\}\\
&\mathit{instance}\ (\mathit{MonadPlus}\ m, \mathit{Monad}\ n) \Rightarrow \mathit{Monoid}\ (\mathit{RefLinkPlusM}\ m\ n)\ \textbf{where}\\
&\quad \mathit{mempty} = \mathit{RefLinkPlusM}\ \$\ \mathit{RefLink}\\
&\qquad \{\ \mathit{update} = \mathit{mzero}\\
&\qquad ,\ \mathit{propagate} = \mathit{return}\ \mathit{mzero}\\
&\qquad \}\\
&\quad \mathit{rl}_1\ \text{`}\mathit{mappend}\text{`}\ \mathit{rl}_2 = \mathit{RefLinkPlusM}\ \$\ \mathit{RefLink}\\
&\qquad \{\ \mathit{update} = (\mathit{update}\ \$\ \mathit{pm}\ \mathit{rl}_1)\ \text{`}\mathit{mplus}\text{`}\ (\mathit{update}\ \$\ \mathit{pm}\ \mathit{rl}_2)\\
&\qquad ,\ \mathit{propagate} = \mathit{liftM2}\ \mathit{mplus}\ (\mathit{propagate}\ \$\ \mathit{pm}\ \mathit{rl}_1)\ (\mathit{propagate}\ \$\ \mathit{pm}\ \mathit{rl}_2)\\
&\qquad \}
\end{aligned}
$$

The exact semantics of *mplus* depend on the monad *m*. In practice, we use an error-handling state monad *ErrorT e* (*State a*). This means that the first argument of *mplus* is always tried first; if it fails, the second argument is tried. When disjoint forms are used correctly, the alternatives are mutually exclusive, so this order is irrelevant.

1.4.3 The *FForm* type

A form is a value of the following type:

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{FForm}\ \mathit{win}\ m\ a = \mathit{FForm}\\
&\quad \{\ \mathit{runFForm} :: \mathit{Window}\ \mathit{win} \rightarrow \mathit{IO}\ (a, \mathit{RefLink}\ m\ \mathit{IO})\ \}
\end{aligned}
$$

It contains three pieces of information:

1. An IO action which creates the form's controls. This action depends on a pointer to a parent window of type *Window win*, in which they are created.
2. A *RefLink* used to update the values in the controls from—and propagate them to—the form's model.
3. Additional information of type *a*; usually layout information of type *Layout* (defined by the wxHaskell library).

The *FForm* type can be used as a monad, which *binds* the additional (layout) information, *reads* the window pointer, *executes* the control creation functions, and *writes* a *RefLink*.

$$
\begin{aligned}
&\mathit{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{FForm}\ \mathit{win}\ m)\\
&\mathit{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadReader}\ (\mathit{Window}\ \mathit{win})\ (\mathit{FForm}\ \mathit{win}\ m)\\
&\mathit{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadIO}\ (\mathit{FForm}\ \mathit{win}\ m)\\
&\mathit{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadWriter}\ (\mathit{RefLink}\ m\ \mathit{IO})\ (\mathit{FForm}\ \mathit{win}\ m)
\end{aligned}
$$

So, $\mathit{form}_1 \gg\!= f$ means:

- f is applied to the additional information from $form_1$, producing (let's call it) $form_2$.
- The window pointer passed to $form_1 \ggg= f$ is passed to $form_1$ and $form_2$.
- The IO actions in $form_1$ and $form_2$ are sequenced.
- The *RefLink* in $form_2$ is joined to the *RefLink* in $form_1$ using the 'default' *mappend* operator.

Furthermore, functions like *ask* (extract the window pointer), *liftIO* (insert an IO action at form creation time) and *tell* (insert a *RefLink*) are implemented for the *FForm* monad (being an instance of *MonadReader*, *MonadIO* and *MonadWriter*).

We stated in Sect. 1.4.2 that in order to combine two forms in a disjoint way, the *RefLinkPlusM* monoid should be used, which dynamically performs *one* of the *update*/*propagate* operations. Meanwhile, *both* forms should be shown: at form creation time, *both* IO actions should be performed, and *both* layout values are used. Therefore, we have implemented alternative bind and unit operators for forms: $\gg\pm$ and $return^0$. They are similar to $\gg=$ and *return* in every respect, except that they use the *RefLinkPlusM* monoid.

$$\begin{aligned}
&return^0 :: MonadPlus\ m \Rightarrow a \rightarrow FForm\ win\ m\ a \\
&(\gg\pm) :: MonadPlus\ m \Rightarrow \\
&\qquad FForm\ win\ m\ a \rightarrow (a \rightarrow FForm\ win\ m\ b) \rightarrow FForm\ win\ m\ b
\end{aligned}$$

These operators are at the core of the disjoint form combinators <u>*radioGrid*</u> and <u>*checkRow*</u>, whose implementation is discussed in the next section.

1.4.4 Disjoint form combinators

A naïve disjoint form combinator would be

$$\begin{aligned}
&refToForm_1\ \text{`}or\text{`}\ refToForm_2 = \lambda ref \rightarrow \\
&\quad refToForm_1\ ref \gg\pm \lambda lay_1 \rightarrow \\
&\quad refToForm_2\ ref \gg\pm \lambda lay_2 \rightarrow \\
&\quad return^0\ \$\ column\ 5\ [lay_1, lay_2]
\end{aligned}$$

The composite form shows both forms, while the composite *update* function performs only one of the component *update* functions—the first one that succeeds. The same goes for the composite *propagate* operation. However, the form's user has no means whatsoever to discover which *update* has been performed, or to influence which *propagate* to perform![3]

The <u>*radioGrid*</u> combinator does provide these functions: both are fulfilled by the radio buttons. When a subform's *update* is performed, the system selects the radio button in front of it. Conversely, the form's *propagate* is only performed if

[3]Both *propagate* operations will succeed when performed. Due to the *mplus* semantics, the first one will always be selected.

the radio button in front of it *is* indeed selected (the user influences this during the form's lifetime).

The nice thing is that we can express this behaviour quite elegantly in terms of *RefLink* operations. We show this by defining the somewhat simpler disjoint form combinator *alt*, which is a specialisation of <u>*radioGrid*</u>: it joins exactly two forms (denoted $refToForm_1$ and $refToForm_2$). Assume that we can create a two-button radio group, returning a reference value $refRadio :: Ref\ IO\ Int$ to its current selection, which can take values $\{0,1\}$. Now we can define a *RefLink* value:

$$
\begin{array}{l}
rl_1 \ = \ refLink\ ref0\ refRadio \\
\quad \textbf{where}\ ref0 \ = \ Ref \\
\qquad\qquad \{\ val \ = \ return\ 0 \\
\qquad\qquad ,\ app = \lambda f \rightarrow \textbf{do}\ \{\ 0 \leftarrow f\ 0;\ return\ ()\ \} \\
\qquad\qquad \}
\end{array}
$$

In other words: we link *refRadio* to a reference 'to the unchangeable number 0', whose *val* is always 0, and whose *app* function only succeeds when the result of the function application is 0. This has the effect that the *update* operation in rl_1 always selects the topmost radio button (and succeeds), while its *propagate* operation only succeeds when this radio button is selected.[4]

We then lift rl_1 into a form, and join it with the first subform $form_1$ using $\gg$, which utilizes the default (conjunctive) *mappend* operator:

$$tell\ rl_1 \ \gg\ refToForm_1\ ref$$

This form has the desired properties: with an *update*, the radio button is only selected if the value in $form_1$ can be updated, and with a *propagate*, the value in $form_1$ is only propagated if the radio button is selected. We define and use rl_2 in a similar way, producing a second form. We finish the *alt* combinator by joining both forms with $\gg\pm$:

$$
\begin{array}{l}
refToForm_1\ \text{`}alt\text{`}\ refToForm_2 \ = \ \lambda ref \rightarrow \\
\quad liftIO\ \ldots \ \gg\!= \ (laybutton_1, laybutton_2, refRadio) \ \rightarrow \\
\qquad \textbf{let} \\
\qquad\quad rl_1 \ = \ \ldots \qquad \text{— see above} \\
\qquad\quad rl_2 \ = \ \ldots \\
\qquad \textbf{in} \\
\qquad (tell\ rl_1 \ \gg\ refToForm_1\ ref) \ \gg\pm\ \lambda layform_1 \ \rightarrow \\
\qquad (tell\ rl_2 \ \gg\ refToForm_2\ ref) \ \gg\pm\ \lambda layform_2 \ \rightarrow \\
\qquad return^0\ \$\ grid\ 5\ 5\ [[laybutton_1, layform_1], [laybutton_2, layform_2]]
\end{array}
$$

What we have omitted in the second line goes into too much implementation detail; it is an IO action which creates the radio buttons, and returns the layout values $laybutton_1$ and $laybutton_2$, as well as *refRadio*.

[4]Note that this *RefLink* does not use any model state!

The *radioGrid* combinator is a straightforward generalization of *alt* for lists. The *checkRow* combinator is also very much like *alt*, but does not show its second argument form. However, it *does* use its *RefLink*.[5]

1.5 SAFETY

The flexibility provided by compositional functional references has a downside: by omitting reference values, duplicating them, or using them in the wrong places, forms with strange behaviour can be constructed. We give some examples:

1. *declare2* $\lambda(a,b) \rightarrow$ *entry*′ [] a

   This form never shows or changes the second element of its subject type.

2. $\lambda a \rightarrow$ *row*′ 5 [*entry*′ [] a, *entry*′ [] a]

   This form shows its value twice, and only propagates the new value in the control on the right.

3. *declareJust* $ *entry*′ []

   This form is only updated if its model contains a *Just x* value (actually, this is the desired behaviour if the form is part of a disjoint form). If it does not, all forms in the same alternative of the surrounding ‘disjoint clause’ are prevented from being updated (normally, there should be none).

4. *radioGrid* [*entry*′ [], *entry*′ []]

   This form will always put its value in the upper entry control. However, it will propagate values from whichever entry control has its radio button selected.

To prevent the construction of these forms, the programmer can follow some rules such as:

- Every declared reference should be used exactly once.
- Every data constructor of a form’s subject type should be declared exactly once.
- References declared outside a disjoint form must not be used inside it.

Of course, it would be better if these rules would be enforced automatically, e.g. by the type system. Future research should formalize these rules.

1.6 RELATED WORK

As far as we know, the idea of explicitly using a radio button grid to combine forms in a disjoint way is new. The fact that some radio buttons make other elements (ir)relevant is recognized, but existing declarative (web) form languages

[5] Its *RefLink* is joined with a *refLink* between the value *False* and the checkbox value.

have to go out of their way to specify this. In XForms[12], it is accomplished by providing an element's `relevant` property with a Boolean expression that includes an XPath pointer to the radio button choice. In WASH/CGI[9], the programmer builds a *decision tree* (see [10]) to express which data to use when a certain radio button is selected.

A simple disjoint form combinator is already introduced in the thesis[1] from which FunctionalForms originated. However, this combinator always joins exactly two alternative forms. If the subject types of the top form and bottom form are t_1 and t_2, resp., then the subject type of the composite form is always *Either* t_1 t_2. In other words, logic and layout are not separated like they are presently.

Compositional references were introduced by Kagawa[5] as a means to compose mutable data structures such as arrays. In our previous work[1, 2] we used them in a more simple form and with a different goal: to conceptually separate a form's subject type and its layout.

Closely related to compositional references are *lenses*[3], which are also pairs of accessor and modificator functions. While our approach uses a lot of 'little' references throughout the program, lenses are combined into a big lens which *is* the program; this program specifies a bidirectional transformation between model and view.

1.7 CONCLUSIONS AND FUTURE WORK

In this paper, we have identified two patterns for composing forms that edit values of a disjoint union type. The first pattern involves a list of radio buttons, and the second involves a check box. To support these patterns in the FunctionalForms library, we have introduced several new combinators.

These patterns illustrate a novel view, in which a form itself can be seen as 'disjoint'. To demonstrate the fertility of this view, we have shown that these disjoint forms exhibit a cleaner separation between logic and user interface. This makes them more flexible.

However, this flexibility comes at a price: the construction of forms with unwanted behaviour is possible. Methods for preventing this have yet to be researched.

As a further enhancement to FunctionalForms, defining forms for values of a custom Haskell type is made easier, using a Template Haskell macro. This brings the library closer to real-life use. The library version discussed in this paper will shortly be available for download.[6]

We hope to further develop the approach of programming with reference values. We believe that it can be used to construct a far wider range of interfaces in a declarative way.

[6] `http://www.cs.ru.nl/~sandr/FunctionalForms`

REFERENCES

[1] Sander Evers. 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`.

[2] Sander Evers, Peter Achten, and Jan Kuper. A functional programming technique for forms in graphical user interfaces. In *Proceedings of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004), Technical Report 0408*, pages 81–96. University of Kiel, 2004. Selected to appear in LNCS.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. Technical Report MS-CIS-04-15, University of Pennsylvania, August 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".

[4] Simon Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1):0–255, January 2003.

[5] Koji Kagawa. 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*, pages 217–226. ACM Press, 1997.

[6] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[7] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, September 2004.

[8] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[9] Peter Thiemann. 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*, pages 192–208. Springer-Verlag, 2002.

[10] Peter Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. Technical report, Universität Freiburg, May 2003. Available at `http://www.informatik.uni-freiburg.de/~thiemann/papers/`.

[11] The wxWidgets home page can be found at `http://www.wxwidgets.org`.

[12] The XForms home page can be found at `http://www.w3.org/MarkUp/Forms/`.