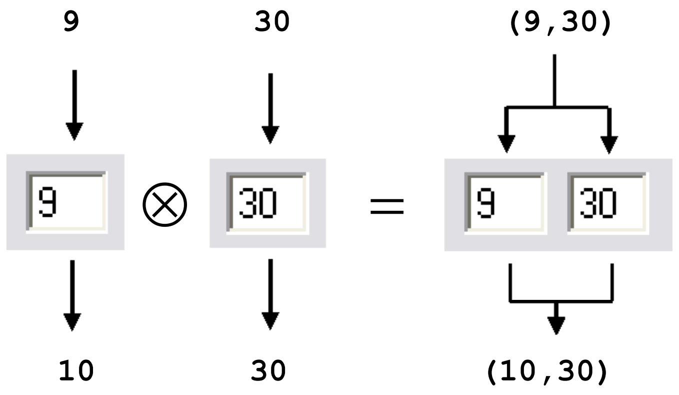
Form follows function
Editor GUIs in a functional style



Master's Thesis

by

Sander Evers

supervised by

dr. ir. J. Kuper
dr. P.M. Achten
dr. M.M. Fokkinga

at the

Language, Knowledge and Interaction group

Faculty of Electrical Engineering,
Mathematics and Computer Science

University of Twente

March 2004

Acknowledgements

I would like to thank my supervisors: Jan for giving me the freedom and trust to pursue my own research ideas, Peter for his helpful ideas and meticulous comments, and Maarten for some valuable process reflection (and useful TEXhelp). I would also like to thank my parents for never questioning my judgement and my flatmates for a nice working environment (with just enough occasional distraction).

Abstract

Programming a graphical user interface (GUI) is often time-consuming and boring, requires quite some knowledge about the GUI library, and is likely to result in monolithic, badly readable and inflexible code — even for small and simple programs like *editors*. These omnipresent GUI parts (for example, all 'Options' and 'Properties' dialogs are editors) allow the user to inspect and update a set of values.

We introduce a small abstract language for describing editors in a modular, flexible, compositional and concise way. In this language, an editor is characterized by its *subject type*, the type of values it can edit. The operators $\otimes$, $\oplus$ and $\mathcal{C}$ are used to construct new editors with new subject types.

We implement this language as a layer of functions upon the Object I/O Library in the purely functional language Clean. Using this functions, it is indeed possible to quickly construct editors in a declarative way, i.e. without defining object identifiers and event handlers.

However, the layout structure of these editors is coupled to the structure of their subject type. We investigate two approaches to decouple these two structures: using a *monadic style* and using *references*.

Samenvatting

Het programmeren van een grafische user interface (GUI) is vaak langdradig, vergt veel kennis van de GUI library en resulteert meestal in monolithische, slecht leesbare en niet-flexibele programmacode — zelfs voor simpele programmaatjes zoals *editors*. Deze alomtegenwoordige GUI-onderdelen (alle windowtjes voor 'Opties' en 'Eigenschappen' zijn voorbeelden van editors) laten de gebruiker een verzameling waarden bekijken en bewerken.

We definiëren een abstract taaltje waarmee editors beschreven kunnen worden op een modulaire, flexibele, compositionele en compacte manier. Hierin wordt een editor gekarakteriseerd door zijn *subject type*, het type van waarden die ermee bewerkt kunnen worden. De operatoren $\otimes$, $\oplus$ en $\mathcal{C}$ construeren nieuwe editors met nieuwe subject types.

We implementeren dit taaltje als een laag functies bovenop de Object I/O Library in de puur functionele taal Clean. Met deze functies kan een programmeur daadwerkelijk snel en declaratief editors specificeren, dus zonder dat hij/zij object identifiers hoeft te gebruiken of event handlers hoeft te schrijven.

Van deze editors is echter wel de layoutstructuur gekoppeld aan het subject type. We onderzoeken twee manieren om deze twee structuren los te koppelen: door een *monadische stijl* en m.b.v. *references*.

Contents

1 Introduction

Programming a graphical user interface (GUI) is often a time-consuming and boring task. Usually, a lot of code is spent on converting values and passing them around; furthermore, creating even the smallest dialog requires quite some knowledge about library functions, parameters and general architecture. These problems become especially significant when the goal is not to produce a professional and highly customized GUI, but rather a quick and simple one which 'gets the job done'.

Another problem is that GUI code tends to get monolithic, badly readable and tangled up with application code. As a consequence, it is difficult to change (parts of) an existing GUI. However, with today's rapidly increasing variety of computing devices, the need for flexible interfaces—and tools for creating them—rises [13].

In this thesis, we investigate an approach for describing GUIs in such a way that they are

1. easy and efficient to program;
2. flexible in changing their form, while keeping their functionality.

In order to do so, we will restrict ourselves to simple GUIs with limited functionality, which we call *editors*. An editor allows the user to inspect and update a set of values. While virtually no application consists solely of editors, they constitute a substantial part of many GUIs. For example, the 'Options' and 'Properties' dialogs found in every desktop application are editors; an address book can be considered an editor; and on small devices such as mobile phones, editors take the form of menu structures.

The advantage of this restriction is that in editors almost every GUI element has a very clear and simple meaning: it contains a value, a piece of information. For example, a list of radio buttons specifies one out of n possibilities, a check box answers a yes/no question and a text field contains a string. The meaning of the whole editor is simply the sum of its parts.

Four desired properties which guide our language design are:

1. *Modularity:* it must be possible to divide the program into different modules. Definitions regarding one particular subject, e.g. editing an address, should be restricted to one particular module as much

as possible. Dependencies between modules should be low, so most changes remain local.

2. *Flexibility* regarding form: changes which only affect the *form* of an editor (and not the functionality, e.g. the values it can produce) should be easy to make.

3. *Compositionality:* it should be possible to 'glue' editors together to form a larger editor.

4. *Conciseness:* an editor definition should be short. It should not contain double work.

In chapter 3, we will introduce a small abstract language for describing editors. For the implementation of this language, we turn to functional programming, which is known for its compact and declarative nature and high-level program combinators. We first implement non-graphical editors in Haskell (chapter 4), and subsequently use the language Clean[14] and its GUI library[3] for graphical editors (chapter 5). After this, we solve a flexibility problem of our abstract language in two different ways: using a monadic style in chapter 6 and using references in chapter 7.

2 Related work

A widely recognized method to manage the complexity of a GUI and to separate GUI and application logic is the model–view–controller (MVC) paradigm[12]. The idea of MVC is to manage application domain data in *model* objects. The graphical representation of this data is managed by one or more *view* objects per model (a progress bar is a typical view object) and the user can change the data using *controller* objects (a button is a typical controller object). In practice, view and controller are often joined in one view/controller object. An example of such an object is a text field: it can both show data and alter it.

A model has no knowledge of how its views present the information; when the model changes, it simply informs the view object of its new state. All communication between different view/controller pairs is routed through the model; this way, different view/controller pairs do not need to be aware of (and dependent on) each other's existence. Furthermore, when different models communicate with each other, they do not have to be aware of each other's views and controllers.

Together, the models form an abstraction layer on top of a layer of user interface objects. It hides the details about *form* to the rest of the program, exposing only application-related *functionality.* We use the same abstraction in our research.

On the World Wide Web, editors take the shape of *forms.* For programming these forms, a W3C standard called XForms[1] has been developed recently. This standard also separates application logic from presentation. For each form, the structure of the edited data is described in the *XForms Model*: every data element gets a name, an initial value, and possibly type or value constraints. In the *XForms User Interface*, each GUI object is bound to one of these data elements using the name of the element as a *reference.* In section 7, we adopt a similar approach with references.

As for GUI abstractions in a functional language, the Fudgets system[7] is worth mentioning. It features high-level combinators to connect different GUI parts. The resulting GUI itself can also be connected to other GUI parts, which is exactly the kind of compositionality we also seek. The connections between these parts consist of information streams. However, these streams have to be managed explicitly, which results in a loss of flexibility.

Closely related to our own approach are Graphical Editor Components[4] (GECs). These are editors in Clean which can be automatically derived from a datatype by generic progamming, but can also be customized. Their most recent extension is called Abstract GECs or AGECs[5] and also incorporates the model–view–controller paradigm to increase flexibility. AGECs distinguish between a *Data Model* and a *View Model*. The correspondence between the two models consists of a bijection in a very similar way that our $\mathcal{C}$-transformation uses a bijection (see section 3.3). With AGECs, highly customizable editors can be built, but they are also more complex to program than our editors, which provide only simple functionality.

3 An abstract view on defining editors

In this chapter, we will provide an abstraction for defining editors. We introduce a small mathematical language which exhibits the desired properties of modularity, flexibility, compositionality and conciseness. This language will guide us in writing combinators in a functional programming language (in chapters 4 and 5).

3.1 Editors

The objects of interest in our language are called editors. An editor can be any process or object in the user interface that is able to display and alter a certain value. Some examples of simple editors are: a text entry field, a drop-down list for selecting a single value out of a few possibilities, a slider to denote a numerical value on some scale, a check box to denote a truth value. Combinations of editors are also regarded as editors themselves; a window with a text entry field and a check box edits a composite value (in this case, a tuple containing a text string and a truth value). We define:

> **An *editor* is a part of the user interface which allows the user to view any value of a certain type and change it into any other value of this type. This type is called the *subject type* of the editor.**

There can be many different editors with the same subject type. Although their form may be different, they all share the same functionality.[1] One can also say that their *user interface* is different, but their *program interface* is not. We capture this shared property with a set definition:

> $[\![A]\!]$ **is the set of all editors with subject type** A**.**

Every editor has exactly one subject type, i.e. if $e \in [\![X]\!]$ and $e \in [\![Y]\!]$ then $X = Y$. We denote editors with lowercase and subject types with uppercase letters.

Subject types can be constructed in the following way:

- *Unit* is a subject type. There is exactly one value of this type.

[1] or, in terms of the model–view–controller paradigm, the same model

- If X and Y are subject types, then $X \times Y$ is a subject type. $X \times Y$ is the Cartesian product of X and Y. Values of this type correspond to ordered pairs.
- If X and Y are subject types, then $X + Y$ is a subject type. $X + Y$ is the disjoint union of X and Y. Values of this type correspond to tagged values of X or Y.

Although these three are theoretically sufficient for most purposes, we also introduce some convenience types:

- *Bool* to represent booleans. There are two values of this type.
- $\mathit{Int}_{n..m}$ ($n \leq m$) to represent integers ranging from n through m (inclusive). There are $m - n + 1$ values of this type.
- *Int* to represent integers when we don't care about the range.
- *String* to represent strings of characters.

3.2 Constructing new editors

Assuming some given atomic editors, we would like to express ways to construct new editors out of less complex editors. To this end, we introduce two operators on editors: $\otimes$ and $\oplus$. Because we are defining an abstract language, these do not go into details about the form of the constructed editor, but regarding its functionality (i.e. its subject type), the following properties hold. Assume $x \in [\![X]\!]$ and $y \in [\![Y]\!]$. Then

$$x \otimes y \in [\![X \times Y]\!]$$
$$x \oplus y \in [\![X + Y]\!]$$

So, the editor $x \otimes y$ allows the user to edit a value of type $X \times Y$. This can be accomplished by deconstructing the ordered pair, editing a value of type X and editing a value of type Y separately, and combining the result in an ordered pair again. For this separate editing, we can make use of the editors x and y, and this is exactly what we are going to do when we implement $\otimes$ in a programming language.

There are many conceivable ways to join the concrete editors x and y into $x \otimes y$. Some examples are:

- first present x; when the user is done with it, present y
- first present y; then x
- present the objects x and y next to each other in a window

- present the questions x and y in a 'wizard'-type dialogue with 'previous' and 'next' buttons

- ask the user which values to edit; as a response present x, y, both, or none

While these are all different ways to construct a user interface for $x \otimes y$, depending on the user interfaces of x and y, it should be noted that the program interface of $x \otimes y$ remains identical.

In fact, these different possible concrete semantics for $\otimes$ do not have to exclude each other. Each one can be regarded as a different *variant* of $\otimes$. When we want to use several different variants of $\otimes$ in the definition of an editor, we denote those variants $\otimes_1$, $\otimes_2$, $\otimes_3$, etc.

Although $x \otimes y$ edits a value of type X *and* one of type Y, $x \oplus y$ does not simply edit a value of type X *or* one of type Y. It should allow the user to change any value of type $X + Y$ into any other value of that type, so it must also be possible to change a (tagged) value of type X into a (tagged) value of type Y, and vice versa.

Considering this, some concrete ways to join x and y into $x \oplus y$ are:

- present the objects x and y next to each other with one of them marked; the user can toggle the mark between them at any time to indicate the final value of the editor

- first show the user the current 'tag' and allow switching (i.e. first edit the tag); then present either x or y

In each case, both x and y have to be provided with a default value, which they can use as their initial value in case the initial value for $x \oplus y$ has the other 'tag'. In section 4.4, we will discuss this in more detail.

3.3 Information equivalence of subject types

Some types can contain exactly the same amount of information; they are said to be *information equivalent*. For example, to store a person's name and age, one can use the type $\mathit{String} \times \mathit{Int}$ as well as $\mathit{Int} \times \mathit{String}$. We denote this equivalence

$$\mathit{String} \times \mathit{Int} \sim \mathit{Int} \times \mathit{String}.$$

Any value of type $\mathit{String} \times \mathit{Int}$ can be mapped to a distinct corresponding value of type $\mathit{Int} \times \mathit{String}$, and vice versa. This is the property we use to formally define information equivalence:

> $A \sim B$ **iff there exists a bijection between the values of A and the values of B.**

Clearly, $\sim$ is an equivalence relation. Also, it is easily verified that the following laws hold:

$$\begin{array}{rcl}
X & \sim & X \\
X \times Y & \sim & Y \times X \\
X + Y & \sim & Y + X \\
(X \times Y) \times Z & \sim & X \times (Y \times Z) \\
(X + Y) + Z & \sim & X + (Y + Z) \\
X \times (Y + Z) & \sim & (X \times Y) + (X \times Z) \\
X \times \mathit{Unit} & \sim & X \\
\mathit{Bool} & \sim & \mathit{Unit} + \mathit{Unit} \\
\mathit{Int}_{n..m} & \sim & \mathit{Unit} + (\mathit{Unit} + (\mathit{Unit} + \ldots)) \\
& & \quad \text{(where } \mathit{Unit} \text{ occurs } m - n + 1 \text{ times)}
\end{array}$$

Editors with information equivalent (but different) subject types almost have the same functionality, but not quite: the interface to the rest of the program represents the same information in a different way. However, it is easy to adapt this interface; we can, for example, adapt an editor with subject type $\mathit{Int} \times \mathit{String}$ so that it behaves like an editor with subject type $\mathit{String} \times \mathit{Int}$ to the rest of the program, while the user interface stays the same. We define the unary operator $\mathcal{C}$ on editors to carry out this transformation.

Assume $b \in [\![B]\!]$ and $A \sim B$, so there exists an bijection $f: A \leftrightarrow B$. Then

> **$\mathcal{C}_f$ applied to b yields an editor which, supplied by its program environment with initial value α of type A, behaves to the user like b supplied with initial value $\beta = f(\alpha)$. When the user changes this value into β', the editor passes the value $\alpha' = f^{-1}(\beta')$ of type A to its program environment.**

Now we can use the editor $\mathcal{C}_f b$ to change any value of type A into any other value of that type. Therefore $\mathcal{C}_f b \in [\![A]\!]$.

3.4 Defining editors

So far, our language consists of the binary operators $\otimes$ and $\oplus$ (with their variants $\otimes_1, \otimes_2, \ldots, \oplus_1, \oplus_2, \ldots$) and the unary operator $\mathcal{C}$. When needed, we can assume that there are some atomic editors. With these constructs, we can denote a large range of editors.

However, we also want to model the *definition* of these editors explicitly, so we add a definition statement to our language. It allows the 'programmer' to give a name to an editor, and use this name in other definitions. The ability to name values occurs in every practical programming language, and is used to:

- avoid doing the same work twice
- clarify the meaning of a sub-program
- create modularity: the definition of a sub-program is separated from its use; this definition can be changed without changing the code which is using it

It is mainly the last point that is of interest to us. We denote an editor definition like this:

$$\mathsf{name} := \mathit{value}$$

To distinguish programmer-defined names from other editor values, we write them in a sans-serif font. We do not trouble ourselves with scope rules or (mutually) recursive definitions.

3.5 Properties of the abstract language

By means of a very simple editor, we will show that our abstract language now exhibits the properties of modularity, flexibility, compositionality and conciseness. The editor we define allows the user to change a date, which is a value of type $Int_{1..31} \times Int_{1..12}$, where the first element represents the day and the second element represents the month.

We assume the existence of the atomic $\mathit{intEditor}_{n..m}$, displaying an integer in the range n–m, which the user can change. Our first definition of dateEditor will be

$$\begin{aligned} \mathsf{dayEditor} &:= \mathit{intEditor}_{1..31} \\ \mathsf{monthEditor} &:= \mathit{intEditor}_{1..12} \\ \mathsf{dateEditor} &:= \mathsf{dayEditor} \otimes_1 \mathsf{monthEditor} \end{aligned}$$

As seen from the last line, we can use a ‘divide–and–conquer’ strategy to define editors with a composite subject type. We just compose the editor in the same way its subject type is composed, only having to choose a variant of $\otimes$ or $\oplus$.

3.5.1 Modularity

Now assume we want to change the month-editing part of our date editor into another atomic editor, with the same functionality but a different form. We change the second line into

$$\mathsf{monthEditor} := \mathit{otherIntEditor}_{1..12}$$

We have only made a local change, without the need to change other parts of the program: a simple example of modularity. This modularity is a triviality in our abstract language; here, it stems from the fact the name $\mathsf{monthEditor}$ can refer to any element of $[\![Int_{1..12}]\!]$; they all have the same interface to the rest of the program and can all be used as an operand to $\otimes$.

3.5.2 Flexibility

How easy is it to change the form of a composite editor without changing its functionality? We take $\mathsf{dateEditor}$ as an example, and distinguish four cases:

1. We want to change an operand into a different editor, but with the same subject type. We have already seen an example of this in subsection 3.5.1.

2. We want to change an operand into a different editor with a different, but information equivalent, subject type. Say we want to replace the $intEditor_{1..12}$ for months with a drop-down list in which we can pick a month, and this drop-down list specifies this choice with the integers 0 through 11. Now we have to write a bijection

   $$\begin{aligned} &f\colon Int_{1..12} \leftrightarrow Int_{0..11} \\ &f(x) = x - 1 \end{aligned}$$

   and replace $intEditor_{1..12}$ with $\mathcal{C}_f\ dropdownlist$.

3. We want to change the operator variant. Say we replace $\otimes_1$, which places the first editor to the left of the second, with $\otimes_2$, which places the first above the second.

4. We want to change the order of the operands to achieve certain effects in the user interface. However, the resulting editor $\mathsf{monthEditor} \otimes \mathsf{dayEditor}$ has subject type $Int_{1..12} \times Int_{1..31}$, which is information equivalent but different. To keep the functionality the same, we must change the subject type back, so we apply a $\mathcal{C}$-transformation:

   $$\begin{aligned} \mathsf{dateEditor} \quad := \quad & \mathcal{C}_f(\mathsf{monthEditor} \otimes_1 \mathsf{dayEditor}) \\ & \text{where } f\{\,(\delta, \mu)\,\} = (\mu, \delta) \end{aligned}$$

In the cases (1) and (3), the changes are easy to make. In case (2), a conversion function has to be specified, but this seems unavoidable (this information has to be specified somewhere) and could be relatively easy, like in the example. The changes in case (4), however, feel awkward. We need to make a similar change in two[2] distinct places. The cause of this is

[2] In an implementation in a functional language, we specify a bijection f with the tuple (f, f^{-1}), containing both the function itself and its inverse; this results in yet another place to make the change.

that the operators $\otimes$ and $\oplus$ are actually too strong: they construct both the user interface and the program interface. We only wanted to change the user interface, so we undo the other change with the $\mathcal{C}$-transformation. In chapters 6 and 7 we investigate ways to decouple both structures.

3.5.3 Compositionality

The property of compositionality is also achieved trivially, because we intended $\otimes$ and $\oplus$ for that purpose: we can use any editor to construct other editors. For example, we could construct

$$\mathsf{dateEditor} \otimes \mathsf{timeEditor}$$

to let the user specify a date and a time for an alarm to go off, and

$$(\mathsf{optionDaily}^3 \oplus \mathsf{dateEditor}) \otimes \mathsf{timeEditor}$$

to include the possibility for the alarm to go off every day.

3.5.4 Conciseness

We can be short about this point: our language is very concise, but of course it is still only an abstract language. We should be careful not lose this conciseness in our implementations.

[3] $\mathsf{optionDaily} \in [\![Unit]\!]$

4 Console-based editors in Haskell

In this chapter, we show that the abstract language can be made executable in a functional language. Before turning to graphical editors in chapter 5, we start with a simple implementation of console-based editors. In a functional language with a built-in I/O monad such as Haskell[10], this proves to be quite straightforward. Only the implementation of $\oplus$ causes some trouble.

4.1 Console-based I/O in Haskell

For now, we restrict ourselves to editors which only use a simple two-way communication channel over which they can transmit and receive characters. The most widely used way of dealing with such an I/O channel in a functional language is representing operations on this channel by monadic values. We use Haskell as our implementation language for these editors, since it provides standard support for monads (in particular the I/O monad).

Haskell's standard library functions

```
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
readLn   :: Read a => IO a
```

take care of console input and output; `putStrLn` is an operation which prints a string on the console (terminated by a newline character), `print` is a similar operation, but able to handle any value which can be converted to a string, and `readLn` is an operation which takes one line of input from the console. Operations are sequenced with the monadic bind operator, which is invisibly applied if we use Haskell's `do` notation.

4.2 Representation of editors

An editor with Haskell subject type `a` corresponds to a monadic function of type `a -> IO a`. It takes as an argument the initial value for the editor, and its result is a monadic computation which uses the I/O channel to produce the changed value.

```
type Editor a = a -> IO a
```

As an example we construct an atomic editor with subject type `Int`:

```
intEditor :: Editor Int
intEditor initval =
   do
      putStrLn "Current value:"
      print initval
      putStrLn "Input new value:"
      readLn
```

This very simple editor prints the initial value to the console and subsequently prompts for a new value. As `readLn` is the last operation in the do-sequence, the monadic value of the whole editor corresponds to the input that the user has given.

At this point, we would like to mention that we do not focus on the usability or attractiveness of our editors here. Rather, we construct editors with only the bare functionality that makes them editors, in order to keep our function definitions as readable as possible. A more attractive design alternative for `intEditor` would have customizable prompts (so we can keep the user aware of what s/he is editing), the ability to increase and decrease the current value with the $+$ and $-$ keys, the option to leave the initial value unchanged with one keypress, etc. These improvements are all possible within our general framework; the type of the editor remains `Editor Int`.

4.3 Implementing $\otimes$ and $\mathcal{C}$

As a variant of $\otimes$, we implement `andthen`, an infix function which takes two editors as its arguments. It runs the first editor, followed by the second. Conforming to the definition of $\otimes$, `editor1 ‘andthen‘ editor2` is an editor itself, with a tuple as its subject type.

```
andthen :: Editor a -> Editor b -> Editor (a,b)
editor1 ‘andthen‘ editor2 =
   \(initval1, initval2) ->
      do
         changedval1 <- editor1 initval1
         changedval2 <- editor2 initval2
         return (changedval1, changedval2)
```

The reason why this definition is so short is that the actual work of sequencing two I/O operations is already done for us in the I/O monad; the only thing we add is the (de)construction of the subject type tuple. Defining `convertF`, the implementation of $\mathcal{C}$, is just as easy. This function takes a bijection as its first argument, which we represent with a tuple containing both the function itself and its inverse. The second argument is the editor we want to adapt:

```
convertF :: (a->b, b->a) -> Editor b -> Editor a
convertF (forth,back) editorB =
   \initvalA ->
      do
         changedvalB <- editorB (forth initvalA)
         return (back changedvalB)
```

Using `intEditor` and `andthen`, we can already write the date editor example from section 3.5.[1] Its definition exactly mimics the abstract language:

```
monthEditor = intEditor
dayEditor   = intEditor
dateEditor  = dayEditor ‘andthen‘ monthEditor
```

4.4 Implementing $\oplus$

Implementing `alt`, a variant of $\oplus$, is more difficult. This function should take an `Editor a` and an `Editor b` as its arguments, and produce an `Editor (Either a b)` as its result. Its naïve definition would be

```
alt :: Editor a -> Editor b -> Editor (Either a b)
editorL ‘alt‘ editorR = editorE
   where
   editorE (Left initval) =
      do
         changedval <- editL initval
         return (Left changedval)
   editorE (Right initval) =
      do
         changedval <- editR initval
         return (Right changedval)
```

but this would only allow the user to make a change within one alternative, e.g. from `Left "foo"` into `Left "bar"` or from `Right 3` into `Right 4`. However, it should also be possible to change `Left "foo"` into `Right 4`. To accomplish this, we will let the user edit the Left/Right tag first. We define the editor `tagEditor`, which shows the current tag using the character `L` or `R`. The user can then choose a new tag by pressing L or R:

[1]The only difference is that this editor does not restrict the integer range. To remedy this, we could define an `intEditor n m`, which checks whether the user input falls between `n` and `m`; if it does not, it could ask the user for a different value, or it could just return the initial value or one of the boundary values.

```
tagEditor :: Editor Char
tagEditor initval =
   do
      putStrLn "Current tag: "
      putStrLn [initval]
      putStrLn "Input new tag: "
      c <- getChar
      putStrLn ""
      return c
```

Now the decision which of the two editors to run can depend on user input, rather than on the initial value. However, this creates a problem. Say we combine the editors

```
stringEditor :: Editor String
intEditor :: Editor Int
```

into

```
stringEditor `alt` intEditor :: Editor (Either String Int)
```

and this editor is run with the initial value `Left "foo"`. When the user chooses to leave the `Left` tag unchanged, we run `stringEditor` and supply it with the initial value `"foo"`. However, when the user changes the tag into `Right`, we need to run `intEditor`, but we have got no initial value to supply it with!

We can solve this problem by requiring the programmer to supply a *default* value for both editors. Using those, we determine the initial values for both editors, given the initial value for the composite editor: one will be this initial value without its tag, the other one will be the default value for that editor. The definition of `alt` becomes:

```
alt :: (a, Editor a) -> (b, Editor b) -> Editor (Either a b)
(defaultL,editorL) `alt` (defaultR,editorR) = editorE
   where
   editorE initvalE =
      do
         changedtag <- tagEditor inittag
         chooseEditor changedtag
      where

      (inittag, initvalL, initvalR) = det_inits initvalE
      det_inits (Left val) = ('L', val, defaultR)
      det_inits (Right val) = ('R', defaultL, val)

      chooseEditor 'L' =
         do
            changedval <- editorL initvalL
            return (Left changedval)
      chooseEditor 'R' =
         do
            changedval <- editorR initvalR
            return (Right changedval)
```

However, this definition of `alt` violates the design of our abstract operator $\oplus$: it does not work on editors anymore, but on tuples of type `(a, Editor a)`. This has serious repercussions for the usability of the editor language:

- We cannot use the same type of operands to $\otimes$ and $\oplus$ anymore, which is confusing. One could say that we are creating two subclasses of editors: with and without a default value.
- Because `(d1,e1) 'alt' (d2,e2)` itself is in the latter class, we cannot simply write `((d1,e1) 'alt' (d2,e2)) 'alt' (d3,e3)`.
- When adding a default value to `(d1,e1) 'alt' (d2,e2)`, we could in principle specify an entirely new value, but in practice it will always be `(Left d1)` or `(Right d2)`. This means double work, and an extra dependency: if we later decide to swap the two operands, we must also change the default value from `Left x` to `Right x` or vice versa.
- The default value for editors with subject type *Unit* will of course always be the unit value, but we still need to specify it.

Therefore a better choice is to include a default value in all our editors, even if it is never used. It requires the following changes to our implementation:

- The type `Editor a :: a -> IO a`
  changes into `Editor a :: (a, a -> IO a)`.
- All atomic editors are provided with reasonable default values.
- The default value of `editor1 'andthen' editor2` is defined to be the tuple containing the default value of `editor1` and the default value of `editor2`.
- The default value of `convertF (forth,back) editorB` is defined to be `back` applied to the default value of `editorB`.
- `alt` is replaced with two variants: `altL` and `altR`, with respective default values `Left d1` and `Right d2` (where `d1` and `d2` are the default values of the left and right operand)

- We define the function `setDefault :: a -> Editor a -> Editor a` to alter the default value of any editor.

Note that most of these changes are invisible to the programmer. The `dateEditor` example even remains exactly identical. The only visible effect is that `altL` and `altR` can now be used in the same way as `then`, just as in the abstract language.

The implementation of `altL`, `altR` and `setDefault`, as well as the changed versions of `intEditor`, `andthen` and `convert`, can be found in appendix A.

5 Graphical editors in Clean

We now take the step from the simple world of console-based user interfaces into the complex world of graphical user interfaces. Not only does the interaction take place through different hardware components (graphical display, pointing device), but also through abstract 'software devices' such as windows, menus, buttons, icons, check boxes, etc. In the 1990s, interfaces using these concepts have become the established standard, at least from a user's perspective.

From a programmer's perspective, there are several 'standards' for building these GUIs; these are known as GUI toolkits. Some more or less widely used toolkits are Microsoft Foundation Classes (MFC) for Windows, Swing (part of Java Foundation Classes) and GTK (cross-platform, mostly used on Unix variants). An important common factor is that they all use an object oriented framework.

In the functional world, a standard toolkit does not exist yet. There are several libraries in existence which form an interface between a functional language and existing object oriented toolkits. The Clean Object I/O Library[3] is one of them. Using this library feels a lot like object oriented programming:

- One can define windows (or *dialogs*) and populate them with *controls*, such as an `EditControl` (for entering text), a `PopUpControl` (to select an option from a drop-down list) and a `ButtonControl`.
- Every control has certain static and dynamic properties, like its caption, select state (whether the control is enabled or disabled), visibility and current selection (for controls like radio buttons and drop-down lists).
- During the execution of the program, the dynamic properties can be read and changed using *get* and *set* functions.
- One can define event handlers which are invoked when the user interacts with a control. These event handlers change the state of the program (and specifically the GUI).

However, all these concepts are expressed in the purely functional language Clean:

- A dialog or control is defined by a value of an elaborate algebraic type (a different type for every kind of control). This value determines the static properties of the object, and the initial value for the dynamic properties. Some of these properties are a compulsory part of the value; others are not, and assume certain default values when left unspecified. For example, the control type for an `EditControl` is:

  ```
  :: EditControl ls pst
   = EditControl String ControlWidth NrLines
       [ControlAttribute *(ls,pst)]
  ```

  In this type, `ls` and `pst` are two free variables which have to do with state management (see below). The data constructor is `EditControl`, which is followed by four arguments: the initial text in the control, the width, the number of lines and a list of optional properties.

- One of these properties is the `ControlFunction`, which acts as an event handler. It is a state transition function, defined by the programmer: it takes a current state (of the whole program) as an argument and its result is a new state.

- This program state, which has type `(ls, PSt ps)`, contains a custom local state (`ls`) and a global process state (`PSt ps`). The latter is a combination of a custom global state (`ps`) and a GUI state (`IOSt ps`). Both custom local and global states can be used to store information (of arbitrary types `ls` and `ps`, respectively) between event handlers. The local state is used to encapsulate (i.e. hide) information which is only relevant for certain controls or dialogs.[1]

- The GUI state has quite a complex type, but this remains hidden from the programmer since it is only accessed using library functions (mainly also state transition functions, restricted to the GUI state). The get and set functions we mentioned are in fact such functions.

- Controls (i.e. the values that define controls) can be joined together using the infix data constructor `:+:`. The resulting value itself is also a control, so multiple controls can be glued together this way. The order in which controls are glued together affects the layout of the composite control.

- Every dialog contains exactly one control, but this can of course be a composite control.[2] The value defining this control is a compulsory part of the value defining the dialog.

[1]An explanation of Object I/O state management can be found in [3].

[2]There is also an 'empty' control available, so it is possible to create an empty dialog.

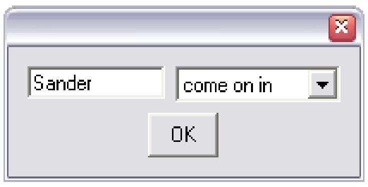


Figure 5.1: Door information editor

- When a dialog definition is *run*, the dialog is shown on the screen in its initial state. When the user interacts with it, the concerning `ControlFunction`s are invoked on the current program state and the dialog is updated accordingly.

The Clean Object I/O Library gives the programmer a medium level of abstraction; e.g. s/he does not need to bother about screen pixel positions, drawing the controls or keeping track of the input focus. The level of abstraction we seek with our editor language lies higher: we do not want the programmer to spend time on naming the objects, writing state transition functions and passing values around using get and set functions.

5.1 An editor example

As an example of programming with the Clean Object I/O Library, we construct a simple composite editor. It edits information about a certain door: the name of the person who works behind it and whether s/he can be disturbed or not. This is done by a small dialog with three controls (see fig. 5.1): an `EditControl` to show and alter the name, a `PopUpControl` showing either 'come on in' or 'do not disturb' and an OK button (a `ButtonControl`) to close the dialog and save the changes. The code that produces this dialog looks like this:

```
mydialog (name,disturb) =
   Dialog "" controls [WindowId idDialog]
   where

   controls =
      EditControl name (PixelWidth 80) 1 [ControlId idEdit]
      :+: PopUpControl labels (bool2int disturb) [ControlId idPopUp]
      :+: ButtonControl "OK"
             [ControlFunction okfun, ControlPos (Center,zero)]

   okfun (ls1,pst1) = (ls2,pst3)
      where
      (Just wstate, pst2) = accPIO (getWindow idDialog) pst1
      (_, Just newtext) = getControlText idEdit wstate
      (_, Just newint) = getPopUpControlSelection idPopUp wstate
      ls2 = (newtext, int2bool newint)
      pst3 = closeActiveWindow pst2

bool2int b = if b 1 2
int2bool i = (i==1)
labels = zip2 ["come on in","do not disturb"] (repeat id)
```

Explained from the top down, this has the following meaning: we are defining the function `mydialog`. When provided with an argument of type `(String,Bool)`, which specifies the initial value for this editor, this function yields a dialog definition. This dialog contains a composite control consisting of an `EditControl`, a `PopUpControl` and a `ButtonControl`.

- The `EditControl` has initial value `name`, is 80 pixels wide, one line high and can be referred to by other controls with the identifier `idEdit`. (This value is defined somewhere in scope; see the next section for some more details about these identifiers.)

- The `PopUpControl` has two labels (defined below by `labels`). The initially selected label is given by `(bool2int disturb)`; i.e. when `disturb` is `True` the first label is initially selected, and when it is `False`, the second label is selected. This control can be referred to with `idPopUp`.

- The `ButtonControl` has the text 'OK' on it, uses event handler `okfun` and is positioned in the center of a new line. (By default, `:+:` positions controls on a line from left to right.) It does not need a control identifier because it is not referred to in any other place.

The event handler `okfun` transforms the current[3] program state `(ls1,pst1)` into a new program state `(ls2,pst3)`. The local state will be used to store the changed value (the reason for this is explained in section 5.5). The process state contains the current GUI state, which is accessed with `accPIO`.

From this GUI state, `okfun` reads the current states of the two controls indicated by the control identifiers `idEdit` and `idPopUp`, using the library functions `getWindow`, `getControlText` and `getPopUpControlSelection`. It combines them into a tuple (with the second element converted back into a `Bool`) and stores this value in the dialog's local state. Finally, it closes the dialog using `closeActiveWindow`.

5.2 Representation of editors

There is a rough correspondence between controls and editors. For example, an `EditControl` would make an editor with subject type *String* and

[3]at the moment the button is pushed

a `PopUpControl` with four items would edit an $Int_{1..4}$. If we glue them together, the composite control's subject type would be $String \times Int_{1..4}$ or $Int_{1..4} \times String$.

However, we cannot directly identify an editor with a control definition. The problem is that some of the behaviour and effect of a control is defined in the event handlers of other controls. In our example, this only happens in the event handler of the OK button: we use `getControlText` and `getPopUpSelection` to extract the updated values from the two controls.

Furthermore, the `ButtonControl` itself cannot be considered an editor. Unlike the `EditControl` and the `PopUpControl`, there is no notion of a current value[4] (which the user can change). Rather than a third independent control, it is more a part of the environment that the other two controls are in; it ends the editing process of those controls.

We will therefore consider our example as some $\mathsf{nameEditor} \otimes \mathsf{disturbEditor}$ which is *run in a dialog with an OK button.* In general, we consider any editor to be run in such an environment. That is why we represent an editor with two main parts:

1. The *opening code*, which describes everything needed when the control is created, i.e. its static properties and initial value. (Note that the word *control* can also mean a composite control!)

2. The *closing code*, which provides the necessary means for the event handler of the OK button to get the updated value from the control.

When joining editors with $\otimes$ or $\oplus$, we combine the opening code of both operands as well as their closing code, but still keep both parts separate. Only at the moment that we construct the environment to actually run an editor in, we combine them. The editor's opening code will then be part of the dialog definition code, together with the definition code of an OK button. The editor's closing code is used in the event handler of this button. (All this is carried out by the function `run_in_dialog`, which we explain in section 5.5.)

This leads to the following schematic type definition of an editor:

```
:: Editor a :== a -> (opening code type, closing code type)
```

Just as with the console editors, the editor type is parameterized with its subject type `a`, and we make a functional abstraction from the initial value (which is of this type). This initial value occurs only in the opening code; however, we include the closing code in the function result for better readability, because we are now going to add another functional abstraction with the same scope.

The reason for this second abstraction is the control identifier mechanism of the Object I/O Library. In the opening code, we give id values to our

[4]although theoretically, this could be a value of type *Unit*

controls, and in the closing code, we refer to the controls using these values. Id values are of type `Id` and are dynamically assigned; at the moment that we create the dialog, we can obtain a list of n ids using `openIds n pst` (the last argument is the current program state).

To take care of these ids, we make two adaptations to the editor type: we add an integer to keep a record of the number of ids needed and make a functional abstraction with argument type `[Id]`. When we construct the dialog, we retrieve the ids with `openIds` and apply the function to that list.

The last addition we make to the `Editor` type is a default value for each editor. It has exactly the same purpose as with console editors (see chapter 4). The type now looks like this:

```
:: Editor a :==
   ( a [Id] ->                                  initial value; list of ids
        (opening code type, closing code type)
   , a                                          default value
   , Int                                        number of ids
   )
```

Note that Clean function type definitions are not in a curried style; if a function has two arguments they are simply separated by a space. In Haskell, we would need an extra `->` between `a` and `[Id]`. To give a concrete example of an editor, we implement a simple `stringEditor`:

```
stringEditor =
   (to_code, "", 1)
```

(its default value is the empty string, and it needs one id value; `to_code` is a function from initial value and list of ids to the opening and closing code, which is defined below:)

```
   where
   to_code initval [cid] =
      ( EditControl initval (PixelWidth 80) 1 [ControlId cid]
      , to_closingval
      )
```

(its `EditControl` has initial value `initval`, is 80 pixels wide and 1 line high, and gets an id value which is used in the closing code `to_closingval`:)

```
      where
      to_closingval wstate = text
         where (_, Just text) = getControlText cid wstate
```

The editor's closing value depends on a value of type `WState`, which describes the state of a window (dialog) and all its controls. When the OK button is pushed, this `WState` is obtained from the GUI state using `getWindow`; the string contents of this particular control are extracted from it using the function `getControlText`. Note the use of the control's id (`cid`).

We can now explain the exact types for the opening and closing code of an editor. The closing code maps a `WState` to the closing value of the particular editor, so its type should be `WState -> a`. The opening code is a control definition; as each kind of control has its own type, we include it in the `Editor` type as a type variable. Therefore, the actual `Editor` type now has two type variables: its subject type `a` and its control type `c`. Its definition is:

```
:: Editor a ct :==
   ( a [Id] ->              initial value; list of ids
        (ct, WState -> a)  control definition; closing value
   , a                      default value
   , Int                    number of ids
   )
```

As an example, the type declaration of our `stringEditor` is

```
stringEditor :: Editor String (EditControl ls pst)
```

where `ls` and `pst` are two free type variables representing custom local state and global process state, respectively.

5.3 Implementing $\otimes$ and $\mathcal{C}$

Implementing the $\otimes$ operator is a little verbose, but actually almost trivial. Its main function is to combine the controls from its two operands, so that they appear next to each other in the dialog. This work is done for us by the data constructor `:+:` from the Object I/O library. The rest is simply administration:

- Just like with the console editors, the initial value (a tuple) is split up into the two initial values for the operands, and their closing values are combined into a tuple again.
- The default values are also combined into a tuple.
- The list of ids is split up (the split position is determined by the id count of the first operand) into two lists.
- The id counts from both operands are added.

For the reader familiar with the concept, this can be seen as a nonterminal production in an *attribute grammar*[11], where

- the two operands are other (non)terminals,
- initial value and list of ids correspond to *inherited attributes*,
- control definition, closing value, default value and id count correspond to *synthesized attributes*.

Johnsson[9] shows how to evaluate an attribute grammar in a functional language, using function arguments to pass inherited attributes and the function result to pass synthesized attributes.

We use an adaptation of this idea (using a different function for every production allows us to work with more than one type). The implementation of `:&:` reads:

```
(:&:) editor1 editor2 = (to_code, (def1,def2), nr1+nr2)
   where
   (to_code1, def1, nr1) = editor1
   (to_code2, def2, nr2) = editor2
   to_code (initval1,initval2) ids =
      (control1 :+: control2, to_closingval)
      where
      (control1,to_closingval1) = to_code1 initval1 ids1
      (control2,to_closingval2) = to_code2 initval2 ids2
      (ids1,ids2) = splitAt nr1 ids
      to_closingval wstate =
         (to_closingval1 wstate, to_closingval2 wstate)
```

The implementation of $\mathcal{C}$ also consists mainly of administration. The essence of the code is that the bijective function `forth` is applied to the initial value, and its inverse `back` to the closing value and default value.

```
convertF (forth,back) editorB = editorA
   where
   (to_codeB, defB, nrB) = editorB
   editorA = (to_codeA, back defB, nrB)
   to_codeA initvalA ids =
      (controlB, back o to_closvalB)
      where
      (controlB, to_closvalB) = to_codeB (forth initvalA) ids
```

5.4 Implementing $\oplus$

Basically, the graphical $\oplus$ implementation is very similar to the console implementation: we add an extra editor to edit the Left/Right tag, and for one of the operand editors we take its default as the initial value. The main

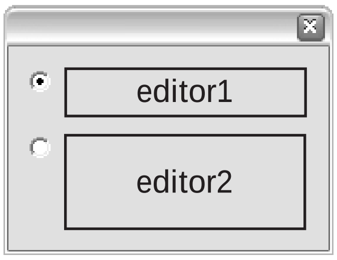


Figure 5.2: Ideal layout for editor1 $\oplus$ editor2

difference is that we now always show both operand editors, along with the tag editor; the user interacts with all three at the same time.

As a tag editor, we use a `RadioControl` with two options. Only one of them can be selected at a time: this way the user makes a choice between the values in the operand editors. We would like to create a layout with the second editor below the first one, and a radio button to the left of each editor, aligned with its top (see fig. 5.2).

However, we run into a cosmetic problem with the Object I/O Library. A group of two radio buttons is considered to be a single item, and we have little control over its internal layout. Although it destroys the usability and attractiveness of our interface, we decide to put the two editors *beside* each other, with the two radio buttons in between. A doubtful advantage of this approach is that the interface now exactly reflects the structure of the expression editor1 $\oplus$ editor2.[5]

The actual implementation is again quite verbose and does not introduce any new concepts. For that reason, it is left out here; it can be found in appendix B. We have named the left- and right-defaulting variants `:.|:` and `:|.:` respectively, with `:|:` as a slightly shorter synonym for the first one. All are defined in term of the more general function `altD`, which takes a boolean argument indicating whether it should default to the left or to the right.

As a small example, we construct a dialog with which we can either choose *yes* or *no*. In the latter case, we must also state a reason why not (see fig. 5.3). Its implementation is:

```
yesnoEditor = yesEditor :|: noEditor
yesEditor = unitLabel "yes"
noEditor = (unitLabel "no, because") :&: stringEditor
```

Note the use of `unitLabel`: it is an editor with subject type *Unit* which appears as a static text label. The subject type of our composite editor is $Unit + (Unit \times String)$; if we want it to be just $Unit + String$ we could apply a $\mathcal{C}$-transformation to `noEditor`:

[5]Unfortunately, it becomes ambiguous when more editors are involved, because we cannot see parentheses in the layout!

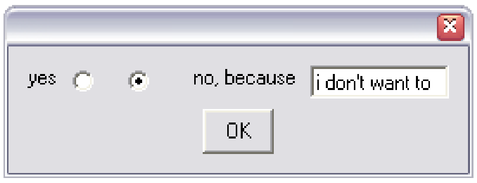


Figure 5.3: `yesnoEditor`

```
noEditor2 = convertF unitTimes noEditor
unitTimes = (forth,back)
   where
   forth x = (UNIT,x)
   back (UNIT,x) = x
```

5.5 Running an editor

Once we have constructed a complete editor using `:&:`, `:|:` and `convertF`, we can use it in a Clean Object I/O program by *running* it with the function `run_in_dialog`. When applied to an editor with subject type `a`, this function yields a state transition function of type `(a, PSt ps) -> (a, PSt ps)`. The first value in these tuples represents the value which is edited, the second is the process state (with an arbitrary global custom state `ps`).

If we apply this state transition function, e.g. in an event handler somewhere in our program, to `(initval, initpst)`, a *modal dialog* is opened, containing the editor, an OK button and a Cancel button. *Modal* means that all user interaction is restricted to that particular dialog, until it is closed. When this happens, the state transition function terminates; a new value and process state are returned in a tuple `(newval, newpst)`.

The editor dialog can be closed in three ways: the user pushes the OK button, the Cancel button or the X button in the upper right corner. In the first case, `newval` contains the altered value from the editor. In the other cases, the user cancels the whole editing operation, so `newval` equals `initval`.

To implement this behaviour, we use the Object I/O library function `openModalDialog`, which opens a modal dialog. Its arguments are:

1. a dialog definition
2. an initial local state for the dialog
3. the initial process state

Its result becomes available when the dialog is closed or when an error occurs while opening the dialog. It consists of a tuple `((err,mls),pst)` where `pst` is the new process state, `err` is an error report and `mls` can be either `Nothing` (when an error occurred) or `Just ls` (otherwise)— `ls` is the dialog's final local state. For simplicity's sake, we assume in our code that no error occurs.

The dialog's local state is used to communicate the edited value to and from the environment: that is why we use `initval` as the second argument to `openModalDialog`, and the pattern `((_, Just newval), pst)` as its result. During its lifetime, this local state is only changed when the OK button is pushed. At that point, the altered value from the editor is copied into it. When the user cancels the operation, the OK button is never pushed, so this copying never happens; the local state is returned unchanged.

The function application `run_in_dialog editor (initval,pst)` performs the following tasks:

- It uses the id count in `editor` to request the necessary number of ids for the editor controls, and one extra for the dialog, with `openIds`. This function returns an id list of the proper length.
- The first element of the `editor` tuple is applied to `initval` and the id list. This yields the editor's control definition and closing code.
- The control definition is joined with two buttons: an OK button and a Cancel button, positioned in the center of a new line.
- The event handler of the OK button transforms the current local and process state tuple `(ls,pst)` into a new one:
  1. It obtains a `WState` value from the current process state using `getWindow` and the dialog id.
  2. It applies the editor's closing code to this value, yielding the closing *value* of the editor. The dialog's local state is replaced by this value.
  3. It closes the dialog with `closeActiveWindow`.

  The event handler of the Cancel button just closes the dialog.
- A dialog is built using the control definition and the dialog id. It has an empty string for a title.
- This dialog is run using `openModalDialog`.
- The result `((_, Just newval), pst)` is converted into `(newval,pst)`.

The implementation of this function can be found in appendix B. In appendix E.2, the function can be seen in action. Using the editor `doorEditor`, we can simply edit `initval` like this:

```
(newval,newpst) = run_in_dialog doorEditor (initval,initpst)
```

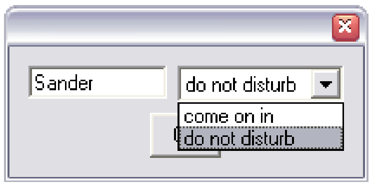


Figure 5.4: Original `doorEditor`

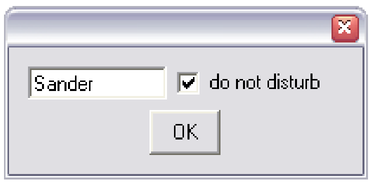


Figure 5.5: Altered `doorEditor`

5.6 The `doorEditor` example revisited

We can now create the `doorEditor` from the beginning of this chapter (see fig. 5.4) using this small program fragment:

```
doorEditor = stringEditor :&: disturbEditor
   where
   disturbEditor =
      convertF (bool2int,int2bool)
         (dropdownEditor ["come on in","do not disturb"])
      where
      bool2int d = if d 1 2
      int2bool i = (i==1)
```

This makes use of the atomic `dropdownEditor`, which takes a list of n strings as an argument and has $Int_{1..n}$ as its subject type. We use $n = 2$ and convert the subject type to *Bool* using `convertF`.

When used in combination with `run_in_dialog`, this new `doorEditor` has the same functionality as the old one; its code is shorter and more understandable. See appendix E for a comparison of the two complete programs including all overhead to import libraries and start up the GUI.

Now let's decide to change the user interface; we want to use a check box for the do–not–disturb option (see fig. 5.5). To produce this check box, we use the atomic `checklistEditor`, which encapsulates a `CheckControl`. This control produces a list of n check boxes, so `checklistEditor`'s subject type is a list of *Bool* (with fixed length n), indicating for all the checkboxes whether they are checked. Like `dropdownEditor`, `checklistEditor` takes a list of n labels as its argument.

We only need one check box, so we use the singleton label list `["do not disturb"]`. The resulting `checklistEditor` has a value `[True]` when the person can *not* be disturbed. Our `disturbEditor` should have value `False` in this case; we make this conversion (singleton to its opposite ordinary value) with `convertF`. We replace the definition of `disturbEditor` with:

```
   disturbEditor =
      convertF (a2b,b2a)
         (checklistEditor ["do not disturb"])
```

```
        where
        a2b a = [not a]
        b2a [b] = not b
```

Compare this to the changes we would have to make to the example program in section 5.1:

- The `PopUpControl` definition is replaced with a `CheckControl` definition, which takes slightly different parameters. For example, together with each label it takes an initial state which can be `Mark` (checked) or `NoMark` (not checked).
- The initial `Bool` value has to be converted to `Mark` / `NoMark`.
- In the button's event handler, we use `getCheckControlSelection` instead of `getPopUpControlSelection`. This function yields a list of integers, indicating which checkboxes are checked.
- This list of integers has to be converted to a `Bool` value: `[1]` maps to `False` and `[]` to `True`.

Clearly, our editor language keeps the changes more local and presents a more consistent manner of dealing with the two different editors. Moreover, this manner also transfers to our composite editor.

6 Decoupling form and functionality

In the previous sections, we have shown that it is possible to create implementations of the abstract editor language for crude textual and graphical editors. These little languages are very concise and 'high-level'; concrete values are passed implicitly to and from the editors in a sort of point-free style. This imposes a limitation, which we already mentioned in section 3.5: only editors with the same user interface structure as their subject type structure can be directly defined. Theoretically, this limitation can be overcome by defining $\mathcal{C}$-transformations on the subject type, but in practice, this is a cumbersome solution. In this chapter, we investigate the possibility of naming the edited values, so the programmer can explicitly couple them to editors in an independently defined user interface structure.

6.1 Explicit value passing in console editors

First, we take a step back to the console editors without default values from section 4.2. We were able to define

```
dateEditor = dayEditor ‘andthen‘ monthEditor
```

with the `andthen` operator distributing the initial values, sequencing the editors and collecting the new values for us. If we want more control, we can simply choose to ignore this operator and do everything ourselves. First we rewrite `dateEditor` (its effect and type remain the same):

```
dateEditor (day,month) =
   do
      day’ <- dayEditor day
      month’ <- monthEditor month
      return (day’, month’)
```

Now we are able to reverse the sequence while maintaining the right subject type:

```
dateEditor (day,month) =
   do
      month’ <- monthEditor month
```

```
        day' <- dayEditor day
        return (day', month')
```

We made a transition from a point-free style to a point-ful style, where we can control the plumbing of the edited values ourselves, regardless of the sequencing of the editors. In fact, this transition consists of two changes which complement each other:

- Changing the type of our building blocks from `a -> IO a` to `IO a` makes function arguments[1] explicit, in this case: the initial day and month values.
- Using the monadic bind operator makes function results[2] explicit, in this case: the changed day and month values.

6.2 Explicit value passing in graphical editors

We now attempt to adopt a similar point-ful approach for graphical editors. The first step, making function arguments explicit in our building blocks, is easy to accomplish. We just leave out a functional abstraction from the `Editor` type, so the control definition includes the initial value again:

```
:: Editor a ct :==
   ( [Id] ->                  list of ids
        (ct, WState -> a)     control definition; closing value
   , a                        default value
   , Int                      number of ids
   )
```

Our atomic editors, however, should *not* contain a built-in initial value; they should be able to handle *any* initial value. Therefore, our new `stringEditor` will not have type `Editor...`, but `String -> Editor...`; its definition stays the same, except that the functional abstraction moves from *inside* of the editor to *outside*:

```
stringEditor :: String -> Editor String (EditControl ls pst)
stringEditor initval =
   (to_code, "", 1)
   where
   to_code [cid] =
      ( EditControl initval (PixelWidth 80) 1 [ControlId cid]
      , to_closingval
      )
      where
```

[1]or: inherited attributes
[2]or: synthesized attributes

```
      to_closingval wstate = text
         where (_, Just text) = getControlText cid wstate
```

Note that while we made the `initval` argument (inherited attribute) explicit by moving the functional abstraction out of the editor, we still keep the `[cid]` argument inside the editor. The reason for this is that we do not want the programmer to pass around lists of ids.

With this new kind of atomic editors, our basic building blocks will no longer be of the form `stringEditor` or `intEditor`, but rather of the form `stringEditor s` and `intEditor i`.

We now want to make a similar change for the components of the function result (i.e. synthesized attributes): the closing values should be made explicit, without exposing the control definition, default value and number of ids. Inspired by the monadic style, we attempt to define a binding function. It should have the type signature

```
(>>&) :: (Editor a _) (a -> Editor b _) -> (Editor b _)
```

with some control definition types filled in at the underscores. The intention is that this function joins an editor with subject type `a` and an editor with subject type `b`. Therefore, the control definitions from the two editors are joined with `:+:` again. This implies that the type variables at the underscores should be `c1 ls pst`, `c2 ls pst` and `:+: c1 c2 ls pst`, respectively, but we will not elaborate on this.[3]

Meanwhile, the closing value of the first editor is made explicit. The idea behind this is that we can use `>>&` several times in a row, at the end of which we construct a closing value for the whole editor. We do this with `returnc`, our equivalent of the monadic return function. As an example, we would like to construct an editor with subject type `(String,Int)` in the following way:

```
myEditor (s,i) =
   intEditor i >>& \new_i ->
      (stringEditor s >>& \new_s ->
         (returnc (new_s, new_i) ))
```

Since `returnc x` must also be an editor, it must contain a control definition which can be joined to the other editors with `:+:`. However, it must not have a representation on the screen; it must be some kind of *empty control*. Fortunately, the Object I/O Library provides such an empty control. It is denoted `NilLS` and is of type `NilLS ls pst` (with the two free type

[3]For the interested reader: `c1` and `c2` are of kind $\star \to \star \to \star$ and are applied to types `ls` (local state) and `pst` (global state). The type constructor `:+:` is of kind $(\star \to \star \to \star) \to (\star \to \star \to \star) \to \star \to \star \to \star$ and ensures that the two controls have the same local and global state types. An explanation of Object I/O state management can be found in [3].

variables signifying that it does not impose any restrictions on the local and global state type). This leads to the following type for `returnc`:

```
returnc :: a -> Editor a (NilLS ls pst)
```

Although we have determined the types for `>>&` and `returnc`, their function definitions are not so simple. There is a problem with the binding function: at the moment that we construct a dialog for

```
editor1 >>& \x -> editor2
```

we need to know the *static* properties of this editor, i.e. the control definitions, the number of ids needed and its default value (see also section 5.5). Half of this information comes from `editor2`, but we cannot reach it because we should fill in the closing value of `editor1` for `x`, which we do not know yet.

On the other hand, we know that there should not be any `x` in the control definition of `editor2`. Therefore, it would be perfectly all right to fill in *any* value of the right type, even the undefined value $\perp$,[4] as long as we are only extracting the control definition—due to lazy evaluation, this value is never evaluated.

Only at the moment that we construct the composite editor's closing value, we need the real closing value of `editor1` to substitute for `x` (consider the closing value `(new_s,new_i)` from the above example). But at that time, it is not a problem anymore: we have a `WState` from which to obtain it. In our definition of `>>&` we therefore apply its second argument *twice*: the first time to obtain the static properties of `editor2` and the second time to obtain the closing value, a dynamic property.

Instead of using $\perp$ for obtaining the static properties, however, we use the default value of `editor1`. Not only is it a conveniently available value of the right type, but this also spontaneously solves the problem of how to construct a default value for the composite editor: we do it in the same way that the closing value is constructed! To see how this works, first consider our definition of `returnc`:

```
returnc val =
   ( to_code,          opening & closing code
   , val               default value
   , 0                 nr of ids
   )
   where
   to_code [] =        only applied to an empty id list
      ( NilLS          control definition
      , \wstate -> val closing value
      )
```

[4] *especially* $\perp$, because it *is* always of the right type

When used on its own, `returnc val` is an invisible ‘editor’ which always has closing value `val`, the same as its default value (which is a somewhat meaningless notion here). When used in the expression

```
stringEditor "foo" >>& returnc
```

the argument of `returnc` is actually bound twice: first to `""`, the default value of `stringEditor "foo"`, then to its closing value σ (which depends on the user and can be any string). The consequence of this and the way `>>&` works will be that the composite editor also gets default and closing values `""` and σ. In fact, the only difference between the composite editor and its first operand is an extra `NilLS` control, which is invisible to the user.

It starts getting interesting when we do something extra with the argument to `returnc`. We could, for example, join it in a tuple with the number `42`:

```
stringEditor "foo" >>& \x -> returnc (x,42)
```

This composite editor gets default value `("",42)` and closing value `(`σ`,42)`. When we bind the second element of the tuple to an `intEditor` instead of using the constant `42`, we have almost returned to our previous example:

```
intEditor 88 >>& \i ->
   (stringEditor "foo" >>& \s ->
      (returnc (s,i) ))
```

The default value of this composite editor is `("",0)`: a combination of the default values of its components, which is constructed in the same way as the closing values are combined into `(`σ`,`ι`)`. Both have the string first, then the integer—regardless of which editor comes first in the user interface structure. (This happens to be `intEditor` here.)

We now give the implementation of `>>&`. Its essence is:

- The control definitions from the two editors (including initial values) are combined with `:+:`.
- Their id counts are added and the id list is split up.
- Applying the second operand to the default value of the first editor yields a second editor; the default value of the composite editor is the default value of this editor. This second editor is also used to obtain the above control definition and id count.
- Applying the second operand to the closing value of the first editor yields another second editor; the closing value of the composite editor is the closing value of this editor.

```
(>>&) editor1 to_editor2 = (to_code, def2, nr1+nr2)
   where
   (to_code1, def1, nr1) = editor1
   (to_code2s, def2, nr2) = to_editor2 def1
```

(use `def1` to obtain the static properties of `editor2`)

```
   to_code ids =
      (control1 :+: control2, to_closingval)
      where
      (ids1,ids2) = splitAt nr1 ids
      (control1, to_closingval1) = to_code1 ids1
      (control2, _) = to_code2s ids2
      to_closingval wstate = to_closingval2 wstate
         where
         closingval1 = to_closingval1 wstate
         (to_code2d,_,_) = to_editor2 closingval1
         (_,to_closingval2) = to_code2d ids2
```

(use `closingval1` to obtain a dynamic property of `editor2`)

6.3 Defining $\otimes$, $\mathcal{C}$ and $\oplus$ in terms of `>>&` and `returnc`

Probably the best evidence of the expressive power of `>>&` and `returnc` is the fact that we can use them to redefine `:&:`, `convertF` and `:|:` (our implementations of $\otimes$, $\mathcal{C}$ and $\oplus$ from the previous chapter). Moreover, these new definitions are shorter and more readable, because all the 'administration' is taken care of by `>>&`. All three functions now operate on arguments of type `a -> Editor a ct`. The first one is defined like this (we leave out the control definition types for readability):

```
(:&:) infixr 5 :: (a -> Editor a _) (b -> Editor b _)
                 -> ((a,b) -> Editor (a,b) _)
(:&:) to_editor1 to_editor2 =
   \(initval1,initval2) ->
      to_editor1 initval1 >>& \closingval1 ->
      to_editor2 initval2 >>& \closingval2 ->
      returnc (closingval1,closingval2)
```

Note the exact similarity to our first implementation of $\otimes$ for console-based editors (see chapter 4.3). This is also the case for `convertF`:

```
convertF :: (a->b, b->a) (b -> Editor b _) -> (a -> Editor a _)
convertF (forth,back) to_editorB =
   \initvalA ->
      to_editorB (forth initvalA) >>& \closingvalB ->
      returnc (back closingvalB)
```

Like in the previous chapter, we define `:|:`, `:.|:` and `:|.:` in terms of the more general `altD`. This function's first argument is `isLeftDefault`, a boolean value indicating whether the composite editor should have its left default value as its own default value. Using `isLeftDefault`, we first define `choiceEditor`, producing the two radio buttons in the middle of the composite editor:

```
choiceEditor =
   setDefault isLeftDefault
      (convertF bool2int2bool (radioEditorRow ["",""]) )
bool2int2bool = (\b -> if b 1 2, \i -> i==1)
```

This editor is built from a (more general) `radioEditorRow` with two empty labels. After the $\mathcal{C}_f$-transformation ($f : \mathit{Bool} \leftrightarrow \mathit{Int}$), it expresses the choice between left and right using `True` and `False`, respectively. It defaults to the tag that we want the composite editor to default to.

To determine the initial values for the two operand editors, we need a little trick. We have an operand of type `a -> Editor a ct` which we must apply to an initial value to turn it into an `Editor a ct`. However, this initial value must be its default value—which we can only discover *after* applying the function! This problem is similar to the one in the previous section, and we can also solve it by first using $\perp$, but again we have a better alternative. We use circular definitions[5] for `initL` and `initR`, and let the lazy evaluation mechanism do the rest:

```
(initL,initC,initR)   = det_inits initE
det_inits (LEFT val)  = (val, True, defR)
det_inits (RIGHT val) = (defL, False, val)
editorL    = to_editorL initL
(_,defL,_) = editorL
editorR    = to_editorR initR
(_,defR,_) = editorR
```

The whole `altD` definition reads:

```
altD isLeftDefault to_editorL to_editorR = to_editorE

   where
   to_editorE initE =
      editorL >>& \closvalL ->
      editorC >>& \closvalC ->
      editorR >>& \closvalR ->
      returnc (if closvalC (LEFT closvalL) (RIGHT closvalR))
```

[5]Unfortunately, this causes a cycle–in–spine error when extracting a default value from `e1 :&: e2`. It can be solved by using a *lazy* pattern match on `(initval1,initval2)` in the definition of `:&:`.

```
      where
      (initL,initC,initR)   = det_inits initE
      det_inits (LEFT val)  = (val, True, defR)
      det_inits (RIGHT val) = (defL, False, val)
      editorL    = to_editorL initL
      (_,defL,_) = editorL
      editorR    = to_editorR initR
      (_,defR,_) = editorR
      editorC    = choiceEditor initC

   choiceEditor =
      setDefault isLeftDefault
         (convertF bool2int2bool (radioEditorRow ["",""]) )
   bool2int2bool = (\b -> if b 1 2, \i -> i==1)
```

This is still somewhat lengthy, but a lot clearer than our previous definition (which can be found in appendix B). We do not need to bother about id counts and lists, so we can concentrate on distributing the initial values and constructing the closing value. Even the composite editor's default value is constructed automatically (in the same way that its closing value is constructed) once we have given `choiceEditor` the right default value.

As for the implementation of `run_in_dialog`, it remains almost identical for our new editors. It only has an argument of type `a -> Editor a c` now, so this is directly applied to `initval` (this application first took place *inside* the editor). The code can be found in appendix C.

6.4 Consequences of the monadic approach

Using `>>&` and `returnc` makes it easy to specify some $\mathcal{C}$-transformations which result from the following information equivalences:

$$\begin{aligned} X &\sim X \\ X \times Y &\sim Y \times X \\ (X \times Y) \times Z &\sim X \times (Y \times Z) \\ X \times \mathit{Unit} &\sim X \end{aligned}$$

For example, to transform the subject type of the editor

$$\mathsf{editorA} \otimes (\mathsf{editorU} \otimes (\mathsf{editorC} \otimes \mathsf{editorB})) \begin{cases} \mathsf{editorA} \in [\![A]\!] \\ \mathsf{editorB} \in [\![B]\!] \\ \mathsf{editorC} \in [\![C]\!] \\ \mathsf{editorU} \in [\![\mathit{Unit}]\!] \end{cases}$$

into $(A \times B) \times C$, we write a definition like:[6]

[6]This is not correct Clean; the identifiers with an apostrophe are invalid and `()` is not a value.

```
newEditor ((a,b),c) =
   editorA a >>& \a' ->
      (editorU () >>& \_ ->
         (editorC c >>& c' ->
            (editorB b >>& b' ->
               (returnc ((a',b'),c') ))))
```

This way, the user interface structure is built with the `>>&` operator; the subject type structure is visible in the formal function argument at the top and the result at the bottom.

We get another advantage for free: using this monadic construction (several instances of `>>&` ending in a `returnc`), we can just as well put *any* Clean pattern in the function argument and *any* Clean value in the result. This way we can define an editor with an arbitrary Clean type as its subject type, for example `(Int,String,String)`, a user-defined algebraic type[7] like

```
:: Vector3D = Vector3D Int Int Int
```

or a record type:

```
:: MyRecord = {name :: String, age :: Int}
```

However, there are also some disadvantages to this monadic approach:

1. We have to write down the same subject type structure twice: once at the top (as a pattern) and once at the bottom (as a result value).

2. We can only easily define editors of the form

$$\mathcal{C}_f(e_1 \otimes (e_2 \otimes (e_3 \otimes \ldots))).$$

   If we want to define an editor of the form

$$\mathcal{C}_f((\mathsf{editorA} \otimes \mathsf{editorB}) \otimes \mathsf{editorC}),$$

   for example, we have to keep track of intermediate result values like this:

```
editor (a,(c,b)) =
   (editorA a >>& \a' ->
      (editorB b >>& \b' ->
         (returnc (a',b') ))) >>& \(a',b') ->
      (editorC c >>& c' ->
         (returnc (a',(c',b')) ))
```

[7]Note that the editing is always restricted to *one* alternative data constructor, just as with our first naïve definition of `alt` in section 4.4.

3. We can only use one variant of $\otimes$, which combines the control definitions of its operands using `:+:`. If we want $\otimes$ to do something else, we would have to define a new binding function, which is not an easy task.

4. To make use of the information equivalences concerning the $+$ operator (without resorting to `convertF`), we would have to write whole customized variants of `altD`, which is a lot of work. The same is true if we want to define editors for user-defined algebraic types with two or more alternative data constructors.

In the next chapter, we will discuss a second approach to decoupling subject type and user interface structure, which does not possess the first two of these disadvantages (see section 7.7).

7 Decoupling with references

The style of programming we propose in this chapter lies somewhere in between the point-free style from chapter 5 and the monadic style from chapter 6. The subject type is made explicit in the function arguments, but not in a result value. As an example, the dateEditor with reversed interface structure looks like this:

```
dateEditor =
   declare2 \(day,month)->
      monthEditor month :++: dayEditor day
```

Despite the absence of some `returnc` clause, the closing value and default value of this editor are constructed in the right way: a tuple with the day value first and then the month value. That this is possible is due to the fact that the values `day` and `month` in the above example—although they are used like the initial values in the monadic style—are actually not ordinary integer values but *references* to integer values.

7.1 Purely functional references

A reference value acts as a pointer to a value in a larger structure of values. This larger structure of values can be seen as a collection of memory cells: using the pointer one can *get* or *set* a value of a certain cell. Although this suggests an imperative style, references can be incorporated easily in a purely functional language like Clean. The key to this implementation is the following type for references:

```
:: Ref cx a =
   { val :: cx -> a
   , app :: (a->a)->(cx->cx)
   }
```

It is a polymorphic record type with two type variables. Type variable `cx` denotes the type of the *context*, i.e. the larger structure of values (collection of memory cells). Type variable `a` denotes the type of the value which is referenced.

Within the record, `val` (for *value*) is a function which retrieves the value from the context and `app` (for *apply*) is a function which updates the value in the context by applying a function to it (this *value update function* of type `a->a` is given as an argument to `app`). We will show how they are used in an example.

Say we have a context of type `(String,Int)`. This can be seen as a memory structure with two cells: one for values of type `String` and one for values of type `Int`. These two cells are referenced by the respective values `ref1` and `ref2` of type:

```
ref1 :: Ref (String,Int) String
ref2 :: Ref (String,Int) Int
```

Now let's say our context initially contains the values `"foo"` and `49`:

```
c1 = ("foo",49)
```

To retrieve the string value from it, we apply the `val` function in `ref1`. It reduces to `"foo"`:

```
ref1.val c1
⇒ "foo"
```

To retrieve the integer value, we use `ref2`:

```
ref2.val c1
⇒ 49
```

To update the integer value, in this case subtract 7 from it, we use `ref2.app`. It has two arguments: a value update function of type `Int->Int` and the initial context. Its result is the updated context:

```
ref2.app (\x->x-7) c1
⇒ ("foo",42)
```

To *set* a complete new value for the string, e.g. `"bar"`, we simply use `\_->"bar"` as a value update function:[1]

```
ref1.app (\_->"bar") c1
⇒ ("bar",49)
```

Now we know what `ref1` and `ref2` are supposed to do, we can understand how we should define them:

```
ref1 = {val=fst, app=appfst}
ref2 = {val=snd, app=appsnd}
```

[1]This function can also be written as `const "bar"`.

This definition makes use of the well-known tuple projection functions `fst` / `snd` and their counterparts `appfst` / `appsnd` which we have defined ourselves:

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

Note that these functions work on all types of tuples, so the type definitions of `ref1` and `ref2` are actually more general than what we first gave:

```
ref1 :: Ref (a,b) a
ref2 :: Ref (a,b) b
```

Now suppose we are dealing with a general context of type `(a,b)` where `a` itself is a tuple type `(a1,a2)`. We want to create a reference `ref12` to the value of type `a2`. This is the second element of the first element of the whole context, so we can define `ref12` like this (`o` denotes function composition[2] in Clean):

```
ref12 :: Ref ((a1,a2),b) a2
ref12 = {val = snd o fst, app = appfst o appsnd}
```

However, we can also derive `ref12` from `ref1` using the function `splitref2`:

```
(ref11,ref12) = splitref2 ref1
```

The function `splitref2` splits a reference to a tuple into two references: a reference to the first element of the tuple and one to the second element of the tuple. All references have the same context type. It is defined like this:

```
splitref2 :: (Ref cx (t1,t2)) -> (Ref cx t1, Ref cx t2)
splitref2 ref = (r1,r2)
   where
   r1 = {val = fst o ref.val, app = ref.app o appfst}
   r2 = {val = snd o ref.val, app = ref.app o appsnd}
```

We can now derive references to the elements of the second element of the first element of the context with

```
(ref121,ref122) = splitref2 ref12
```

... and so on. This way we can create references to any element in any context of nested tuples. Note that we could also have defined `ref1` and `ref2` using the identity reference `idref` and `splitref2`:

```
idref :: Ref a a
idref = {val = id, app = id}

(ref1,ref2) = splitref2 idref
```

[2] `(f o g) x = f (g x)`

7.2 Editors with references

In our new editor language variant, we use these references to construct the subject type of an editor. An editor is now parameterized by a reference instead of an initial value. It uses this reference to:

1. *get* its initial value from the *initial context*, which is passed to the editor as an inherited attribute;
2. *set* its default value in the *default context*
3. *set* its closing value in the *closing context*

The last two operations are represented by *context update functions* of type `cx -> cx` (where `cx` is the type of the context). They are the result of applying the `app` function in the reference to a *value update function*, and constitute synthesized attributes of the editor; these *functions* replace the default and closing *values*. This has the following consequences for the `Editor` type:

```
:: Editor cx ct :==
   ( cx [Id] ->                     initial context; list of ids
        ( ct                        control definition
        , WState -> (cx -> cx)      closing context update function
        )
   , (cx -> cx)                     default context update function
   , Int                            number of ids
   )
```

Note that the editor's subject type `a` has completely disappeared from the type; its function has been taken over by `cx -> cx`.[3] As an example of an editor of this type, we show the implementation of `stringEditor` again:

```
stringEditor ::
   (Ref cx String) -> Editor cx (EditControl ls pst)
stringEditor ref =
   ( to_code
   , ref.app (const "")
   , 1
   )
   where
   to_code initcx [cid] =
      ( EditControl (ref.val initcx)
           (PixelWidth 80) 1 [ControlId cid]
```

[3]This type has less expressive power; it is less stringent, which implies that we can create 'wrong' implementations of editors. This is, in fact, the case when an editor's context update function does not restrict itself to the editor's 'own memory cell'.

```
        , to_updcv
        )
        where
        to_updcv wstate = ref.app (const text)
           where
           (_, Just text) = getControlText cid wstate
```

The changes between the previous `stringEditor` and this one are:

- The parameter for an initial value is replaced by a parameter for a reference (`ref`).
- In `to_code`, there is an extra functional abstraction for the initial context (`initcx`).
- The initial value in the control definition is now obtained by applying `ref.val` to the initial context `initcx` .
- The default value `""` is replaced by the context update function `ref.app (const "")` .
- The closing value `text` is replaced by the context update function `ref.app (const text)` .

We can now explain how two editors are combined. We do this with the operator `:++:` (its implementation can be found in appendix D). Like our previous operators, it joins the control definitions with `:+:` , splits up the id list and adds the id counts. However, the default and closing values are not explicitly constructed anywhere: for each context, the update functions from both editors are just *composed* after one another.

The result we intend to achieve with this is that the n editors in the expression

$$\texttt{editor}_1 \texttt{ :++: editor}_2 \texttt{ :++: ... :++: editor}_n$$

can be freely permutated, and that we can put parentheses anywhere in the expression—all without changing the functionality of the composite editor.

For these properties to hold, it is essential that *it does not matter in what order two context update functions are composed*: either way, the resulting context should be the same. In other words, when `fx` is a context update function in `editorX` and `fy` is its corresponding context update function in `editorY` , then

$$\texttt{fx} \circ \texttt{fy} = \texttt{fy} \circ \texttt{fx}$$

should hold. If this condition is satisfied, the editors

```
editorX :++: editorY
editorY :++: editorX
```

can be interchanged without influencing the composite default or closing value, and that is what we want. Furthermore, we also want

```
(editorX :++: editorY) :++: editorZ
editorX :++: (editorY :++: editorZ)
```

to be interchangeable, but this is achieved trivially; the equation

$$(\texttt{fx} \circ \texttt{fy}) \circ \texttt{fz} = \texttt{fx} \circ (\texttt{fy} \circ \texttt{fz})$$

holds for any functions `fx`, `fy` and `fz`.

7.3 Editors with ×-constructed subject types

In this section, we examine the tree-like subject type structures built using only ×. At the leaves of these binary trees, there may be atomic types or other type structures; this does not matter to us. As we have already seen, we can use `splitref2` to derive references to these leaves (*leaf references*) from a reference to the root of the tree.

We will show that when we use two different leaf references, the desired commutativity of $\circ$ on two different context update functions holds. Intuitively, this is easy to understand: the references point to different 'memory cells' and updating one of those cells does not influence the other. Therefore it does not matter which one is updated first.

To prove this property mathematically, we will consider only the `app` part of the references and denote these functions with a_1, a_2, a_3, etcetera. Two special cases of `app` functions are root, the reference to the top of the tree, and *id*, the identity function. We construct sets of these functions called *leaf sets*. A leaf set can be constructed in the following ways:

- $\{\mathsf{root}\}$ is a leaf set.
- If $L = \{a_1, a_2, \ldots, a_n\}$ is a leaf set, then

  $$\mathsf{split}_i(L) = (L \setminus a_i) \cup \{a_i \circ \mathit{appfst}, a_i \circ \mathit{appsnd}\}$$

  is also a leaf set, for every $1 \leq i \leq n$.

First, we will prove that every function in a leaf set distributes over $\circ$. This means that there is no difference between updating a value with a composition of two functions, i.e. `ref.app (g o h)`, and composing two different updates with the functions separated, i.e. `(ref.app g) o (ref.app h)`.

Please note that in the following definitions and proofs, we implicitly assume all functions to be correctly typed and total.

Definition 7.1 *A function f* distributes over $\circ$ *iff*

$$f\ (g \circ h) = (f\ g) \circ (f\ h)$$

for every function g and h.

Theorem 7.1 *id distributes over $\circ$.*

Proof $id\ (g \circ h) = g \circ h = (id\ g) \circ (id\ h)$ $\square$

Theorem 7.2 *$appfst$ and $appsnd$ distribute over $\circ$.*

Proof Assume an arbitrary tuple (x, y). Then

$$\begin{aligned} & ((appfst\ g) \circ (appfst\ h))\ (x, y) \\ = \ & (appfst\ g)\ ((appfst\ h)\ (x, y)) \\ = \ & (appfst\ g)\ (h\ x, y) \\ = \ & (g\ (h\ x), y) \\ = \ & ((g \circ h)\ x, y) \\ = \ & (appfst\ (g \circ h))\ (x, y). \end{aligned}$$

Since this is true for any tuple (x, y), we can now say

$$(appfst\ g) \circ (appfst\ h) = appfst\ (g \circ h).$$

The proof for *appsnd* is analogous. $\square$

Theorem 7.3 *If f_1 and f_2 distribute over $\circ$, then $(f_1 \circ f_2)$ distributes over $\circ$.*

Proof Assume that f_1 and f_2 distribute over $\circ$. Then

$$\begin{aligned} & ((f_1 \circ f_2)\ g) \circ ((f_1 \circ f_2)\ h) \\ = \ & (f_1\ (f_2\ g)) \circ (f_1\ (f_2\ h)) \\ = \ & f_1\ ((f_2\ g) \circ (f_2\ h)) \\ = \ & f_1\ (f_2\ (g \circ h)) \\ = \ & (f_1 \circ f_2)\ (g \circ h). \end{aligned}$$

$\square$

Theorem 7.4 *If* root *distributes over $\circ$, then every function in a leaf set constructed from* root *distributes over $\circ$.*

Proof For the base leaf set $\{\mathsf{root}\}$, this property is trivially true. It is sufficient to prove that the construction of any new leaf set preserves distributivity over $\circ$. Assume we are constructing $\mathsf{split}_i(L)$ from a leaf set $L = \{a_1, a_2, \ldots, a_n\}$ for which the property holds. Since a_i, *appfst* and *appsnd* all distribute over $\circ$, $(a_i \circ appfst)$ and $(a_i \circ appsnd)$ also distribute over $\circ$, by theorem 7.3. $\square$

With this result, we can prove the property which we are interested in: the composition of two context update functions resulting from different leaf references is commutative. Remember that a context update function is an app function applied to a value update function. In other words:

Theorem 7.5 *Assume* root *distributes over* $\circ$*. Then, for any two functions* a_i *and* a_j $(i \neq j)$ *in a leaf set constructed from* root*,*

$$(a_i\ g) \circ (a_j\ h) = (a_j\ h) \circ (a_i\ g)$$

for every function g *and* h*.*

Proof For the base leaf set $\{\mathsf{root}\}$, this property is trivially true. We will now prove that the construction of new leaf sets preserves the property. Assume we are constructing a new leaf set $\mathsf{split}_i(L)$ from a leaf set $L = \{a_1, a_2, \ldots, a_n\}$ for which the property holds. This means that we are replacing the function a_i with the two functions $a_i \circ \mathit{appfst}$ and $a_i \circ \mathit{appsnd}$. We need to prove:

1. that the composition of $(a_i \circ \mathit{appfst})\ g_1$ with $a_j\ h$ is commutative, for any g_1, h and a_j $(1 \leq j \leq n,\ j \neq i)$;
2. that the composition of $(a_i \circ \mathit{appsnd})\ g_2$ with $a_j\ h$ is commutative, for any g_2, h and a_j $(1 \leq j \leq n,\ j \neq i)$;
3. that the composition of $(a_i \circ \mathit{appfst})\ g_1$ and $(a_i \circ \mathit{appsnd})\ g_2$ is commutative, for any g_1 and g_2.

We already know that $(a_i\ g) \circ (a_j\ h) = (a_j\ h) \circ (a_i\ g)$ for any g, h and a_j $(1 \leq j \leq n,\ j \neq i)$. When we substitute $\mathit{appfst}\ g_1$ for g, we get

$$\begin{array}{rcl}(a_i\ (\mathit{appfst}\ g_1)) \circ (a_j\ h) & = & (a_j\ h) \circ (a_i\ (\mathit{appfst}\ g_1)) \\ ((a_i \circ \mathit{appfst})\ g_1) \circ (a_j\ h) & = & (a_j\ h) \circ ((a_i \circ \mathit{appfst})\ g_1)\end{array}$$

which proves the first property. The proof of the second property is analogous: we substitute $\mathit{appsnd}\ g_2$ for g. To prove the third property, we will first prove that the composition of $\mathit{appfst}\ g_1$ with $\mathit{appsnd}\ g_2$ is commutative. We assume an arbitrary tuple (x, y) to which we apply it:

$$\begin{array}{rl} & ((\mathit{appfst}\ g_1) \circ (\mathit{appsnd}\ g_2))\ (x, y) \\ = & (\mathit{appfst}\ g_1)\ ((\mathit{appsnd}\ g_2)\ (x, y)) \\ = & (\mathit{appfst}\ g_1)\ (x, g_2\ y) \\ = & (g_1\ x, g_2\ y) \\ = & (\mathit{appsnd}\ g_2)\ (g_1\ x, y) \\ = & (\mathit{appsnd}\ g_2)\ ((\mathit{appfst}\ g_1)\ (x, y)) \\ = & ((\mathit{appsnd}\ g_2) \circ (\mathit{appfst}\ g_1))\ (x, y)\end{array}$$

Since this is true for any tuple (x, y), we can also say

$$(appfst\ g_1) \circ (appsnd\ g_2) = (appsnd\ g_2) \circ (appfst\ g_1).$$

We use this result, together with the fact that a_i distributes over $\circ$, to prove the third property:

$$\begin{array}{rl} & ((a_i \circ appfst)\ g_1) \circ ((a_i \circ appsnd)\ g_2) \\ = & (a_i\ (appfst\ g_1)) \circ (a_i\ (appsnd\ g_2)) \\ = & a_i\ ((appfst\ g_1) \circ (appsnd\ g_2)) \\ = & a_i\ ((appsnd\ g_2) \circ (appfst\ g_1)) \\ = & (a_i\ (appsnd\ g_2)) \circ (a_i\ (appfst\ g_1)) \\ = & ((a_i \circ appsnd)\ g_2) \circ ((a_i \circ appfst)\ g_1) \end{array}$$

□

We translate this result into our editor language. Assume that the following conditions are satisfied:

1. `ref`$_1$, `ref`$_2$, ..., `ref`$_n$ are all different leaf references created by `splitref2` from one root reference `rootref` .
2. `rootref.app` distributes over $\circ$.
3. For every $1 \leq x \leq n$, the expression `to_editor`$_x$ `ref`$_x$ contains only context update functions of the form `ref`$_x$`.app f` .

Then the n editors in the expression

```
       to_editor₁ ref₁
:++:   to_editor₂ ref₂
:++:   ...
:++:   to_editorₙ refₙ
```

can be freely permutated, and we can put parentheses anywhere in this expression, without changing the functionality of the composite editor.

Normally, condition (2) is satisfied because `run_in_dialog` applies its editor argument to `idref` (see next section). For all atomic editors, condition (3) is also satisfied, because they are built that way. The programmer only has to watch out that s/he does not violate condition (1), e.g. by using the same reference twice (see also section 7.7).

7.4 A convenient notation for using `splitref2`

We can define `dateEditor` in the following way:

```
dateEditor date =
   monthEditor month :++: dayEditor day
   where (day,month) = splitref2 date
```

Our editor needs an argument of type `(Ref cx (a,b))`, which it converts into two references of type `(Ref cx a)` and `(Ref cx b)` (where `a` is the type of a day value and `b` is the type of a month value). Since this will be an often used pattern, we define a shorter notation for it which uses the function `declare2`:

```
declare2 ::
   ((Ref cx t1, Ref cx t2) -> e) -> ((Ref cx (t1,t2) -> e))
declare2 to_editor = to_editor o splitref2
```

Now we can write:

```
dateEditor =
   declare2 \(day,month)->
      monthEditor month :++: dayEditor day
```

Only the extra `declare2` before the λ-abstraction shows that we are dealing with references instead of ordinary values. This makes it easy for the programmer to define composite editors like `dateEditor`. Just like our atomic editors, these need a reference argument, so they are actually functions from references to editors, i.e. of the type `(Ref cx a) -> Editor cx ct`. They can now be defined in a similar way that composite editors[4] were defined in the previous chapter.

The function `run_in_dialog` now expects its first argument to be of this type. It applies it to `idref`, a reference to the whole context. The initial value from its second argument `(initval,initpst)` is taken to be the initial context and passed to the editor as an inherited attribute. In the event handler of the OK button, the closing value is determined by applying the *closing context update function* (see section 7.2) from the editor to this same initial context. The implementation of `run_in_dialog` can be found in appendix D.

7.5 Defining $\otimes$ and $\mathcal{C}$ with references

When we define editors of which the interface structure does *not* differ from the subject type structure, we should be able to use $\otimes$ again. Its implementation `:&:` is easily defined in terms of editors with references:

```
(:&:) to_editor1 to_editor2 =
   declare2 \(ref1,ref2)->
      to_editor1 ref1 :++: to_editor2 ref2
```

[4]actually: functions from initial values to editors

The implementation of $\mathcal{C}$ only has to alter the reference that an editor gets. This is done by the function `convertref`:

```
:: Bij a b :== (a->b, b->a)      // bijection

convertref :: (Bij a b) (Ref cx a) -> (Ref cx b)
convertref (forth,back) refA =
   { val = forth o refA.val
   , app = \f -> refA.app (back o f o forth)
   }
```

In the `val` part, the function `forth` is applied to the original value in the context. In the `app` part, before applying a value update function `f`, `forth` is also applied, and afterwards the updated value is mapped back using `back`. To apply a $\mathcal{C}$-transformation to an editor instead of a reference, we use `convertF`:

```
convertF :: (Bij a b) ((Ref cx b) -> e) -> ((Ref cx a) -> e)
convertF f to_editorB = to_editorB o (convertref f)
```

Translated to our leaf sets, the use of `convertref` introduces a new way of constructing a leaf set: we convert one of the `app` functions (thereby replacing the old one). From now on, we will use the abbreviation

$$c_i = \lambda u.\, a_i\,(f^{-1} \circ u \circ f)$$

to denote the `app` function a_i which is converted using the bijection f.

- If $L = \{a_1, a_2, \ldots, a_n\}$ is a leaf set, then

  $$\mathsf{convert}_{i,f}(L) = (L \setminus a_i) \cup \{c_i\}$$

  is also a leaf set, for every $1 \le i \le n$.

Theorem 7.6 *If a_i is distributive over $\circ$, then c_i is also distributive over $\circ$.*

Proof Assume that a_i is distributive over $\circ$.

$$\begin{array}{rl} & (c_i\,g) \circ (c_i\,h) \\ = & (a_i\,(f^{-1} \circ g \circ f)) \circ (a_i\,(f^{-1} \circ h \circ f)) \\ = & a_i\,((f^{-1} \circ g \circ f) \circ (f^{-1} \circ h \circ f)) \\ = & a_i\,(f^{-1} \circ (g \circ f \circ f^{-1} \circ h) \circ f) \\ = & a_i\,(f^{-1} \circ (g \circ h) \circ f) \\ = & c_i\,(g \circ h) \end{array}$$

□

Theorem 7.7 *Assume* $(a_i\ g) \circ (a_j\ h) = (a_j\ h) \circ (a_i\ g)$ *for any function* g. *Then also* $(c_i\ g) \circ (a_j\ h) = (a_j\ h) \circ (c_i\ g)$ *for any function* g.

Proof $c_i\ g = a_i\ (f^{-1} \circ g \circ f)$. Therefore, our conclusion is a special case of our assumption (we fill in $f^{-1} \circ g \circ f$ for g). $\square$

These two theorems imply that constructing a new leaf set using $\mathsf{convert}_{i,f}(L)$ preserves the leaf set properties in theorems 7.4 and 7.5. Therefore, our results from section 7.3 also hold for leaf sets with converted references.

7.6 Implementing $\oplus$

The implementation $\oplus$ is once again a very verbose one which does lot of administration. For that reason, it is left out here and can be found in appendix D.

However, one interesting thing happens in this implementation. The result type of the `altD` function is:

```
(Ref cx (EITHER a b)) ->
   Editor cx (:+: c1 (:+: RadioControl c2) ls pst)
```

In other words, it yields an editor which edits a value of type `EITHER a b` in a context `cx` using three controls. However, these controls themselves do not operate in the context `cx`! Instead, they operate in a new context of type `(Bool,(a,b))`. This way, the three controls can all have their own separate 'memory cell', instead of sharing one of type `EITHER a b`. Hence, the editor arguments of `altD` have the types:

```
(Ref (Bool,(a,b)) a) -> Editor (Bool,(a,b)) (c1 ls pst)
(Ref (Bool,(a,b)) b) -> Editor (Bool,(a,b)) (c2 ls pst)
```

The initial value for this new context is determined in the usual way, using the value of type `EITHER a b` in the initial `cx` context and one of the two default values. These default values are obtained by applying the default context update functions to $\perp$ (we know that it is never evaluated).[5]

To determine the context update functions for the composite editor, the three update functions for the new context are applied first; depending on the `bool` value in the updated context `(bool,(l,r))`, the `EITHER a b` value in the `cx` context is updated with either `const (LEFT l)` or `const (RIGHT r)`.

[5]This is because these functions are all constructed like `ref.app \_->value`. Furthermore, we must also replace the strict pattern matching in `splitref2` by lazy pattern matching.

7.7 Consequences of programming with references

Programming with references is easy and elegant. Distributing references in a $\times$-tree is just like distributing ordinary initial values (only `declare2` is added); the closing values are automatically constructed. Also, one can easily build any layout tree with `:++:`, because unlike the `>>&` operator (see section 6.4), it does not force a certain structure on this tree. We can build the example from section 6.4, an editor of the form

$$\mathcal{C}_f((\mathsf{editorA} \otimes \mathsf{editorB}) \otimes \mathsf{editorC})$$

with subject type $A \times (C \times B)$, like this:

```
declare2 \(a,cb)->
   let (c,b)=splitref2 cb in
      (editorA :++: editorB) :++: editorC
```

However, this example also shows a drawback to using references. We cannot use arbitrary nested pattern matches anymore; we can only match on a top-level tuple.[6] Furthermore, this means that we cannot directly use arbitrary Clean datatypes anymore. We can convert to and from them using `convertF`, but this entails the extra work of writing a bijection.[7]

Another disadvantage of using references is that one can easily create editors with a strange behaviour, if the conditions at the end of section 7.3 are not met. For example, in the editor

```
declare2 \(a,b)->
   stringEditor a :++: intEditor b :++: stringEditor a
```

the same reference `a` is used twice.[8] The composition of the two resulting context update functions is *not* commutative; they both 'write into the same memory cell'. Therefore, the closing value of this editor will contain only the closing value of the first `stringEditor`, because its context update function is applied later (due to the definition of `:++:`). The value in the second `stringEditor` is always ignored.

Similar strange behaviour can result from using `convertF (f,g)` when `f` and `g` do not form a bijection. Unfortunately, these errors cannot be found by the type checker.

[6]Of course, it is possible to define the functions `declare1x2`, `declare2x1` and `declare2x2` to match on the three possible tuples with nesting level 1, etc.

[7]However, we have already made an easy and helpful extension to our language by implementing `splitref`X and `declare`X for $X = 3$ and $X = 4$.

[8]Clean's *uniqueness typing system* may be able to prevent this sharing of a reference. We have not had the time to look further into this possibility.

8 Conclusions and future work

Our principal result is that we have revealed the essence of describing and programming editors, which enables us to express them as the simple program combinations that they really are. This essence is recorded in our small abstract language of editors with the operators $\otimes$, $\oplus$ and $\mathcal{C}$. It focuses on the separation—and the connection derived from it—of *form* (user interface) and *functionality* (program interface) of an editor.

Guided by this abstract design, we have implemented the concept of editors in a functional language, in a number of different ways. The console-based editors in Haskell revealed a correspondence between editors and monads, and indicated the need for a static *default value* in addition to the dynamic value in each editor. In Clean, we discovered that we could build the editor language on top of an object oriented GUI framework by viewing it as an attribute grammar.

Next, we investigated two ways to undo the coupling between user interface structure and subject type structure of composite (graphical) editors. Both involve making the edited values explicit. Our first approach is inspired by monads and lets the programmer pass around the initial value as well as the final value of an editor. In our second approach these are replaced by a single *reference*. We find that this approach, although it is a little less expressive, enables a more elegant and concise programming style.

A common property of all three graphical editor implementations is that they relieve the programmer of the burden of naming objects, writing event handlers and calling get and set functions. They make it possible to quickly program simple editors in a modular, flexible, compositional and concise way.

We owe a great deal of these results to the fact that we were working with a functional language. The ability to use functions as first-class values was crucial in our programming experiments, which aided us in thinking about combining program fragments at a high level of abstraction. Although the structure of this thesis might suggest otherwise, the mathematical editor language emerged in interaction with our implementations.

Future work

While we have laid the foundation for a concise editor language, it is not ready for 'professional' use yet. There is still a lot of room for improvement. We name a few possibilities which come to mind:

- The programmer must definitely be able to exercise more influence on the layout of editors. In the Object I/O Library, this is done by supplying controls with extra layout attributes. Some of these attributes are easily transferred to editors; others are not because they use the control ids which we have hidden.

- It is possible (we have already experimented with it) to define an operator which puts two editor dialogs in sequence. In fact, this is a variant of $\otimes$ which has a lot in common with the first console variant of $\otimes$ in section 4.3. This could produce wizard-like dialog sequences with 'Back' and 'Next' buttons. Furthermore, alternate paths in these wizards can be created with a $\oplus$ variant.

- In many real-life GUIs, a base editor dialog opens another editor dialog when the user presses a button (often labeled 'Advanced...'). This should also be possible in our language (the editor in this new dialog possibly operates in a new references context).

- So far, we can only edit values with a static structure, but it should also be possible to define a *list editor*. This editor would have subject type `[a]` and present the user with a list in which s/he can move, add and remove items using buttons. There is also a *current* item. The user can edit this item in an additional editor with subject type `a`, which is for example located below the list or in a new dialog.

- This list editor can also be used as an editor with subject type `a`. Compare this to *selecting* an address in an address book versus *editing* the entries in an address book.

- Some editing events could have a direct effect, instead of the indirect effect when the OK button is pressed. We have already experimented with this, synchronizing two different editors which edited the same value. To achieve this result, an editor's context update function can be applied in the event handler of the control itself. Furthermore, every editor would need an extra synthesized attribute: a *set* function which copies the editor's associated value from the current context into the control.

- A *default button* can be included in an editor dialog, which sets the editor to its default value. An often seen example of this is the 'Clear' button on WWW forms.

- The limitation of `splitref2` to tuple references can possibly be solved by *generic programming* [6, 8]. A generic `splitref` would transform a reference to an arbitrary product type into a product of references.

- Our editor concept could be extended to include *inputters* (editors which always have their default value as initial value) and *outputters* (editors whose value cannot be altered).

- We should investigate the efficiency of our editor implementations.

- Implementing our references approach in other GUI frameworks such as wxHaskell[2] or Fudgets[7] seems possible and might well provide new insights.

- A visual tool could be constructed for defining editors without any programming whatsoever. Creating a layout of the different sub-editors within an editor would be easier with such a tool.

References

[1] http://www.w3.org/MarkUp/Forms/.

[2] http://wxhaskell.sourceforge.net.

[3] Peter Achten and Rinus Plasmeijer. Interactive Functional Objects in Clean. In C. Clack, K. Hammond, and T. Davie, editors, *Proc. of 9th International Workshop on Implementation of Functional Languages, IFL'97*, number 1467 in LNCS, pages 304–321. Springer-Verlag, Berlin, September 1998.

[4] Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Generic Graphical User Interfaces. In Greg Michaelson and Phil Trinder, editors, *Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03*, LNCS. Edinburgh, UK, Springer, 2003. To appear.

[5] Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. Technical Report NIII-R0408, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands, February 2004.

[6] Artem Alimarine and Rinus Plasmeijer. A Generic Programming Extension for Clean. In Thomas Arts and Markus Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, September 2002.

[7] M. Carlsson and T. Hallgren. 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, 1993. ACM.

[8] D. Clarke and A. Löh. Generic Haskell, Specifically. In J. Gibbons and J. Jeuring, editors, *Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48, Schloss Dagstuhl, July 2003. Kluwer Academic Publishers. ISBN 1-4020-7374-7.

[9] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 154–173. Springer-Verlag, 1987.

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

[11] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968. Correction: Mathematical Systems Theory 5: 95–96, 1971.

[12] 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.

[13] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[14] Rinus Plasmeijer and Marko van Eekelen. *Clean Language Report, version 2.0 – Draft.* Department of Software Technology, University of Nijmegen, 2001. Available at ftp://ftp.cs.kun.nl/pub/Clean/Clean20/doc/CleanRep2.0.pdf.

A Implementation of console editors with defaults

```
module EditorsDefault where

type Editor a =
   ( a            -- default value
   , a -> IO a    -- editing operation
   )

andthen :: Editor a -> Editor b -> Editor (a,b)
(default1,edit1) `andthen` (default2,edit2) = ((default1,default2), editBoth)
   where
   editBoth (initval1, initval2) =
      do
         changedval1 <- edit1 initval1
         changedval2 <- edit2 initval2
         return (changedval1, changedval2)

setDefault :: a -> Editor a -> Editor a
setDefault d (_,edit) = (d,edit)

convertF :: (a->b, b->a) -> Editor b -> Editor a
convertF (forth,back) (defaultB,editB) = (defaultA,editA)
   where
   defaultA = back defaultB
   editA initvalA =
      do
         changedvalB <- editB (forth initvalA)
         return (back changedvalB)

intEditor :: Editor Int
intEditor = (0,edit)
   where
   edit initval =
      do
         putStrLn "Current value:"
         print initval
         putStrLn "Input new value:"
         readLn

unitEditor :: Editor ()
unitEditor = ((),return)
```

```
alt = altL
altL = altD True
altR = altD False

altD :: Bool -> Editor a -> Editor b -> Editor (Either a b)
altD isLeftDefault (defaultL,editL) (defaultR,editR) = (defaultE,editE)
   where
   editE initvalE =
      do
         changedtag <- tagEditor inittag
         chooseEditor changedtag
      where

      (inittag, initvalL, initvalR) = det_inits initvalE
      det_inits (Left val) = ('L', val, defaultR)
      det_inits (Right val) = ('R', defaultL, val)

      chooseEditor 'L' =
         do
            changedval <- editL initvalL
            return (Left changedval)
      chooseEditor 'R' =
         do
            changedval <- editR initvalR
            return (Right changedval)

   defaultE = if isLeftDefault then (Left defaultL) else (Right defaultR)

   tagEditor :: Char -> IO Char  -- has no default value, so no editor!
   tagEditor initval =
      do
         putStrLn "Current tag: "
         putStrLn [initval]
         putStrLn "Input new tag: "
         c <- getChar
         putStrLn ""
         return c

run :: Editor a -> a -> IO a
run = snd
```

B Implementation of point-free graphical editors

```
implementation module EditorsPF

import StdEnv, StdIO

:: Editor a c :==
   ( a [Id] ->         // initial value; control ids
      ( c,             // control definition
        WState -> a )  // closing value, given state of the dialog
   , a                 // default value
   , Int               // nr of controls
   )

:: EITHER a b = LEFT a | RIGHT b
:: UNIT = UNIT

:: Bij a b :== (a->b,b->a) // bijection

convertF :: (Bij a b) (Editor b c) -> Editor a c
convertF (forth,back) editorB = editorA
   where
   (to_codeB, defB, nrB) = editorB
   editorA = (to_codeA, back defB, nrB)
   to_codeA initvalA ids =
      (controlB, back o to_closvalB)
      where
      (controlB, to_closvalB) = to_codeB (forth initvalA) ids

setDefault :: a (Editor a c) -> Editor a c
setDefault newdef (to_code, _, nr) =
   (to_code, newdef, nr)

(:&:) infixr 5 ::
   (Editor a (c1 ls pst)) (Editor b (c2 ls pst))
   -> Editor (a,b) (:+: c1 c2 ls pst)
(:&:) editor1 editor2 = (to_code, (def1,def2), nr1+nr2)
where
   (to_code1, def1, nr1) = editor1
   (to_code2, def2, nr2) = editor2
   to_code (initval1,initval2) ids =
      (control1 :+: control2, to_closingval)
   where
      (control1,to_closingval1) = to_code1 initval1 ids1
      (control2,to_closingval2) = to_code2 initval2 ids2
      (ids1,ids2) = splitAt nr1 ids // split id list in two
      to_closingval wstate =
         (to_closingval1 wstate, to_closingval2 wstate)

(:|:) infixr 5 ::
   (Editor a (c1 ls pst)) (Editor b (c2 ls pst))
   -> Editor (EITHER a b) (:+: c1 (:+: RadioControl c2) ls pst)
(:|:) editL editR = editL :.|: editR

(:.|:) infixr 5 ::
   (Editor a (c1 ls pst)) (Editor b (c2 ls pst))
   -> Editor (EITHER a b) (:+: c1 (:+: RadioControl c2) ls pst)
(:.|:) editL editR = altD True editL editR

(:|.:) infixr 5 ::
   (Editor a (c1 ls pst)) (Editor b (c2 ls pst))
   -> Editor (EITHER a b) (:+: c1 (:+: RadioControl c2) ls pst)
(:|.:) editL editR = altD False editL editR

altD :: Bool (Editor a (c1 ls pst)) (Editor b (c2 ls pst))
   -> Editor (EITHER a b) (:+: c1 (:+: RadioControl c2) ls pst)
altD isLeftDefault editL editR = (to_code, defE, nrL+1+nrR)
where
   (to_codeL, defL, nrL) = editL
   (to_codeR, defR, nrR) = editR
   defE = if isLeftDefault (LEFT defL) (RIGHT defR)
   to_code initE ids =
      (controlL :+: radios :+: controlR, to_closingval)
   where
      (initL,initChoice,initR) = det_inits initE
      det_inits (LEFT val) = (val, 1, defR)
      det_inits (RIGHT val) = (defL, 2, val)
      (controlL,to_closingvalL) = to_codeL initL idsL
      (controlR,to_closingvalR) = to_codeR initR idsR
      (idsL,[idRadio:idsR]) = splitAt nrL ids
      radios =
         RadioControl [empty,empty] (Rows 1) initChoice [ControlId idRadio]
      where
         empty = ("",Nothing,id)
      to_closingval wstate =
         if (radioselection==1) (LEFT closingvalL) (RIGHT closingvalR)
      where
         (_, Just radioselection) = getRadioControlSelection idRadio wstate
         closingvalL = to_closingvalL wstate
         closingvalR = to_closingvalR wstate

stringEditor :: Editor String (EditControl ls pst)
stringEditor =
   (to_code, "", 1)
   where
   to_code initval [cid] =
      ( EditControl initval (PixelWidth 80) 1 [ControlId cid]
      , to_closingval
      )
      where
      to_closingval wstate = text
         where (_, Just text) = getControlText cid wstate

unitLabel :: String -> Editor UNIT (TextControl ls pst)
unitLabel label =
   (to_code, UNIT, 0)
   where
   to_code UNIT [] =
      ( TextControl label []
      , \_ -> UNIT
      )

dropdownEditor :: [String] -> Editor Int (PopUpControl ls pst)
dropdownEditor labels =
   (to_code, 1, 1)
   where
   to_code initval [cid] =
      ( PopUpControl items initval [ControlId cid]
      , to_closingval
      )
      where
         items = zip2 labels (repeat id)
         to_closingval wstate = index
            where (_, Just index) = getPopUpControlSelection cid wstate

checklistEditor :: [String] -> Editor [Bool] (CheckControl ls pst)
checklistEditor labels =
   (to_code, map (const False) labels, 1)
   where
   to_code initval [cid] =
      ( CheckControl items (Rows 1) [ControlId cid]
      , to_closingval
      )
      where
         items = [(l,Nothing,if b Mark NoMark,id) \\ l<-labels & b<-initval]
         to_closingval wstate =
            [isMember i selection \\ _<-initval & i<-[1..]]
            where
               (_, Just selection) = getCheckControlSelection cid wstate

int2string2int :: Bij Int String
int2string2int = (toString,toInt)

intEditor :: Editor Int (EditControl ls pst)
intEditor = convertF int2string2int stringEditor

run_in_dialog ::
   (Editor a (c a (PSt ps)))
   (a,(PSt ps))
   -> (a,(PSt ps))      | Controls c
```

```
run_in_dialog editor (initval,pst)
   # ([winid:cids], pst) = openIds (nr+1) pst
     (control, to_closingval) = to_code initval cids
     dialog =
      Dialog "" (control :+: (buttons winid to_closingval))
         [WindowId winid, WindowClose (noLS closeActiveWindow)]
   # ((_, Just newval), pst) = openModalDialog initval dialog pst
   = (newval, pst)

   where
   (to_code, _, nr) = editor
   buttons winid to_closingval =
      LayoutControl (ok :+: cancel) [ControlPos (Center,zero)]
      where
      ok = ButtonControl "OK" [ControlFunction storecv]
      cancel = ButtonControl "Cancel"
         [ControlFunction (noLS closeActiveWindow)]
      storecv (ls,pst)
         # (Just wstate,pst) = accPIO (getWindow winid) pst
         # ls = to_closingval wstate
         # pst = closeActiveWindow pst
         = (ls,pst)
```

Note the use of Clean's let-before construct, marked with a `#`. The scope of the identifiers in the lhs of a definition following `#` does not include the rhs of the definition itself or the definitions above it, but does include the definitions below it. This way we can use the same name `pst` for the values of the process state at different stages. The same goes for the local state `ls`.

C Implementation of monadic-style graphical editors

```
implementation module EditorsMon

import StdEnv, StdIO

:: Editor a c :==
   ( [Id] ->            // control ids
      ( c,              // control definition
        WState -> a )   // closing value, given state of the dialog
   , a                  // default value
   , Int                // nr of controls
   )

:: EITHER a b = LEFT a | RIGHT b
// can't import normal Either from StdLibMisc because of name clash with StdIO!

:: UNIT = UNIT

:: Bij a b :== (a->b,b->a)

(>>&) infixl 0 ::
   (Editor a (c1 ls pst)) (a -> Editor b (c2 ls pst))
   -> Editor b (:+: c1 c2 ls pst)
(>>&) editor1 to_editor2 = (to_code, def2, nr1+nr2)
   where
   (to_code1, def1, nr1) = editor1
   (to_code2s, def2, nr2) = to_editor2 def1
   to_code ids =
      (control1 :+: control2, to_closingval)
      where
      (control1, to_closingval1) = to_code1 ids1
      (control2, _) = to_code2s ids2
      to_closingval wstate = to_closingval2 wstate
         where
         closingval1 = to_closingval1 wstate
         (to_code2d,_,_) = to_editor2 closingval1
         (_,to_closingval2) = to_code2d ids2
      (ids1,ids2) = splitAt nr1 ids
```

```
returnc :: a -> Editor a (NilLS ls pst)
returnc val =
   ( to_code            // opening & closing code
   , val                // default value
   , 0                  // nr of ids
   )
   where
   to_code [] =         // only applied to an empty id list
      ( NilLS           // control definition
      , \wstate -> val  // closing value
      )

setDefault :: a (a -> Editor a c) -> (a -> Editor a c)
setDefault newdef to_editor1 = setDef o to_editor1
   where
   setDef (to_code, _, nr) =
      (to_code, newdef, nr)

convertF ::
   (Bij a b) (b -> Editor b (c ls pst))
   -> (a -> Editor a (:+: c NilLS ls pst))
convertF (forth,back) to_editorB =
   \initvalA ->
      to_editorB (forth initvalA) >>& \closingvalB ->
      returnc (back closingvalB)

(:&:) infixr 5 ::
   (a -> Editor a (c1 ls pst)) (b -> Editor b (c2 ls pst))
   -> ((a,b) -> Editor (a,b) (:+: c1 (:+: c2 NilLS) ls pst))
(:&:) to_editor1 to_editor2 = to_editorBoth
   where
   to_editorBoth inits =
      to_editor1 initval1 >>& \closingval1 ->
      to_editor2 initval2 >>& \closingval2 ->
      returnc (closingval1,closingval2)
      where
      (initval1,initval2) = inits    // lazy binding to avoid cycle-in-spine

(:|:) infixr 5
(:|:) to_editorL to_editorR = to_editorL :.|: to_editorR

(:.|:) infixr 5
(:.|:) to_editorL to_editorR = altD True to_editorL to_editorR

(:|.:) infixr 5
(:|.:) to_editorL to_editorR = altD False to_editorL to_editorR

altD ::
   Bool
   (a -> Editor a (c1 ls pst))
   (b -> Editor b (c2 ls pst))
   -> ((EITHER a b) -> Editor (EITHER a b)
        (:+: c1 (:+: (:+: RadioControl NilLS) (:+: c2 NilLS)) ls pst))
```

```
altD isLeftDefault to_editorL to_editorR = to_editorE
   where
   to_editorE initE =
      editorL >>& \closvalL ->
      editorC >>& \closvalC ->
      editorR >>& \closvalR ->
      returnc (if closvalC (LEFT closvalL) (RIGHT closvalR))
      where
      (initL,initC,initR)   = det_inits initE
      det_inits (LEFT val)  = (val, True, defR)
      det_inits (RIGHT val) = (defL, False, val)
      editorL    = to_editorL initL
      (_,defL,_) = editorL
      editorR    = to_editorR initR
      (_,defR,_) = editorR
      editorC    = choiceEditor initC
   choiceEditor =
      setDefault isLeftDefault
         (convertF bool2int2bool (radioEditorRow ["",""]) )
   bool2int2bool = (\b -> if b 1 2, \i -> i==1)

stringEditor :: String -> Editor String (EditControl ls pst)
stringEditor initval =
   (to_code, "", 1)
   where
   to_code [cid] =
      ( EditControl initval (PixelWidth 80) 1 [ControlId cid]
      , to_closingval
      )
      where
      to_closingval wstate = text
         where (_, Just text) = getControlText cid wstate

unitLabel :: String UNIT -> Editor UNIT (TextControl ls pst)
unitLabel label UNIT =
   (to_code, UNIT, 0)
where
   to_code [] =
      ( TextControl label []
      , \_ -> UNIT
      )

int2string2int :: (Int -> String, String -> Int)
int2string2int = (toString,toInt)

intEditor :: (Int -> Editor Int (:+: EditControl NilLS ls pst))
intEditor = convertF int2string2int stringEditor

radioEditor ::
   ([String] Int
   -> Editor Int (RadioControl ls pst))
radioEditor = radioEditorG (Columns 1)
```

```
radioEditorRow ::
   ([String] Int
   -> Editor Int (RadioControl ls pst))
radioEditorRow = radioEditorG (Rows 1)

radioEditorG ::
   RowsOrColumns [String] Int
   -> Editor Int (RadioControl ls pst)
radioEditorG rowsorcolumns labels initval = (to_code, 1, 1)
   where
   to_code [cid] =
      ( RadioControl items rowsorcolumns initval [ControlId cid]
      , to_closingval
      )
      where
      items = [ (label,Nothing,id) \\ label <- labels ]
      to_closingval wstate = radioselection
         where
         (_, Just radioselection) = getRadioControlSelection cid wstate

run_in_dialog ::
   (a -> Editor a (c a (PSt ps)))
   (a,(PSt ps))
   -> (a,(PSt ps))      | Controls c
run_in_dialog to_editor (initval,pst)
   # ([winid:cids], pst) = openIds (nr+1) pst
     (control, to_closingval) = to_code cids
     dialog =
      Dialog "" (control :+: (buttons winid to_closingval))
         [WindowId winid, WindowClose (noLS closeActiveWindow)]
   # ((_, Just newval), pst) = openModalDialog initval dialog pst
   = (newval, pst)

   where
   (to_code, _, nr) = to_editor initval
   buttons winid to_closingval =
      LayoutControl (ok :+: cancel) [ControlPos (Center,zero)]
      where
      ok = ButtonControl "OK" [ControlFunction storecv]
      cancel = ButtonControl "Cancel"
         [ControlFunction (noLS closeActiveWindow)]
      storecv (ls,pst)
         # (Just wstate,pst) = accPIO (getWindow winid) pst
         # ls = to_closingval wstate
         # pst = closeActiveWindow pst
         = (ls,pst)
```

D Implementation of graphical editors with references

```
implementation module EditorsRef

import StdEnv, StdIO

:: Ref cx a =                 // ref. to value of type a in context of type cx
   { val :: cx -> a           // retrieve value from context
   , app :: (a->a)->(cx->cx)  // update value in context
   }

:: Bij a b :== (a->b, b->a)      // bijection

:: UNIT = UNIT
:: EITHER a b = LEFT a | RIGHT b

:: Editor cx ct :==
   ( cx [Id] ->               // initial context; list of control ids
      ( ct                    // control definition
      , WState -> (cx -> cx)  // update closing value in context
      )
   , cx -> cx                 // update default value in context
   , Int                      // nr of ids
   )

idref :: Ref a a
idref = {val=id,app=id}

declare2 ::
   ((Ref cx a, Ref cx b) -> thing)
   -> ((Ref cx (a,b)) -> thing)
declare2 f = f o splitref2

declare3 ::
   ((Ref cx a1, Ref cx a2, Ref cx a3) -> thing)
   -> ((Ref cx (a1,a2,a3)) -> thing)
declare3 f = f o splitref3

declare4 ::
   ((Ref cx a1, Ref cx a2, Ref cx a3, Ref cx a4) -> thing)
   -> ((Ref cx (a1,a2,a3,a4)) -> thing)
declare4 f = f o splitref4
```

```
declare1x2 ::
   ((Ref cx a, (Ref cx b1, Ref cx b2)) -> thing)
   (Ref cx (a,(b1,b2)))
   -> thing
declare1x2 f ref = f (ref1, splitref2 ref2)
   where
   (ref1,ref2) = splitref2 ref

declare2x1 ::
   (((Ref cx a1, Ref cx a2), Ref cx b) -> thing)
   (Ref cx ((a1,a2),b))
   -> thing
declare2x1 f ref = f (splitref2 ref1, ref2)
   where
   (ref1,ref2) = splitref2 ref

declare2x2 ::
   (((Ref cx a1, Ref cx a2), (Ref cx b1, Ref cx b2)) -> thing)
   (Ref cx ((a1,a2),(b1,b2)))
   -> thing
declare2x2 f ref = f (splitref2 ref1, splitref2 ref2)
   where
   (ref1,ref2) = splitref2 ref

splitref2 :: (Ref cx (a1,a2)) -> (Ref cx a1, Ref cx a2)
splitref2 r = (r1,r2)
   where
   r1 = { val = fst o r.val, app = r.app o appfst }
   r2 = { val = snd o r.val, app = r.app o appsnd }
   appfst f t = (f x, y) where (x,y) = t
   appsnd f t = (x, f y) where (x,y) = t
   // lazy tuple binding to avoid cycle-in-spine

splitref3 :: (Ref cx (a1,a2,a3)) -> (Ref cx a1, Ref cx a2, Ref cx a3)
splitref3 r = (r1,r2,r3)
   where
   r1 = { val = fst3 o r.val, app = r.app o appfst3 }
   r2 = { val = snd3 o r.val, app = r.app o appsnd3 }
   r3 = { val = thd3 o r.val, app = r.app o appthd3 }
   appfst3 f t = (f x, y, z) where (x,y,z) = t
   appsnd3 f t = (x, f y, z) where (x,y,z) = t
   appthd3 f t = (x, y, f z) where (x,y,z) = t

splitref4 ::
   (Ref cx (a1,a2,a3,a4)) -> (Ref cx a1, Ref cx a2, Ref cx a3, Ref cx a4)
splitref4 r = (r1,r2,r3,r4)
   where
   r1 = { val = fst4 o r.val, app = r.app o appfst4 }
   r2 = { val = snd4 o r.val, app = r.app o appsnd4 }
   r3 = { val = thd4 o r.val, app = r.app o appthd4 }
   r4 = { val = for4 o r.val, app = r.app o appfor4 }
   fst4 (x,y,z0,z1) = x
   snd4 (x,y,z0,z1) = y
   thd4 (x,y,z0,z1) = z0
   for4 (x,y,z0,z1) = z1
   appfst4 f t = (f x, y, z0, z1) where (x,y,z0,z1) = t
   appsnd4 f t = (x, f y, z0, z1) where (x,y,z0,z1) = t
   appthd4 f t = (x, y, f z0, z1) where (x,y,z0,z1) = t
   appfor4 f t = (x, y, z0, f z1) where (x,y,z0,z1) = t

convertF ::
   (Bij a b) ((Ref cx b) -> thing)
   -> ((Ref cx a) -> thing)
convertF f to_editorB = to_editorB o (convertref f)

convertref :: (Bij a b) (Ref cx a) -> (Ref cx b)
convertref (forth,back) refA =
   { val = forth o refA.val
   , app = \f -> refA.app (back o f o forth)
   }

setDefault ::
   a ((Ref cx a) -> Editor cx c)
   -> ((Ref cx a) -> Editor cx c)
setDefault newdef to_editorOld = to_editorNew
   where
   to_editorNew ref = (to_code, ref.app (const newdef), nr)
      where
      (to_code, _, nr) = to_editorOld ref

(:++:) infixr 5 ::
   (Editor cx (c1 ls pst)) (Editor cx (c2 ls pst))
   -> Editor cx (:+: c1 c2 ls pst)
(:++:) editor1 editor2 =
   ( to_code
   , upddef1 o upddef2
   , nr1 + nr2
   )
   where
   (to_code1,upddef1,nr1) = editor1
   (to_code2,upddef2,nr2) = editor2
   to_code initcx ids =
      (control1 :+: control2, to_updcv)
      where
      (ids1,ids2) = splitAt nr1 ids
      (control1, to_updcv1) = to_code1 initcx ids1
      (control2, to_updcv2) = to_code2 initcx ids2
      to_updcv wstate = (to_updcv1 wstate) o (to_updcv2 wstate)

(:&:) infixr 5 ::
   ((Ref cx a) -> Editor cx (c1 ls pst))
   ((Ref cx b) -> Editor cx (c2 ls pst))
   -> ((Ref cx (a,b)) -> Editor cx (:+: c1 c2 ls pst))
(:&:) to_editor1 to_editor2 =
   declare2 \(initval1,initval2) ->
      to_editor1 initval1 :++: to_editor2 initval2
```

```
(:|:) infixr 5 ::
   ((Ref (Bool,(a,b)) a) -> Editor (Bool,(a,b)) (c1 ls pst))
   ((Ref (Bool,(a,b)) b) -> Editor (Bool,(a,b)) (c2 ls pst))
   -> ((Ref cx (EITHER a b)) -> Editor cx (:+: c1 (:+: RadioControl c2) ls pst))
(:|:) to_editorL to_editorR = to_editorL :.|: to_editorR

(:.|:) infixr 5 ::
   ((Ref (Bool,(a,b)) a) -> Editor (Bool,(a,b)) (c1 ls pst))
   ((Ref (Bool,(a,b)) b) -> Editor (Bool,(a,b)) (c2 ls pst))
   -> ((Ref cx (EITHER a b)) -> Editor cx (:+: c1 (:+: RadioControl c2) ls pst))
(:.|:) to_editorL to_editorR = altD True to_editorL to_editorR

(:|.:) infixr 5 ::
   ((Ref (Bool,(a,b)) a) -> Editor (Bool,(a,b)) (c1 ls pst))
   ((Ref (Bool,(a,b)) b) -> Editor (Bool,(a,b)) (c2 ls pst))
   -> ((Ref cx (EITHER a b)) -> Editor cx (:+: c1 (:+: RadioControl c2) ls pst))
(:|.:) to_editorL to_editorR = altD False to_editorL to_editorR

altD ::
   Bool
   ((Ref (Bool,(a,b)) a) -> Editor (Bool,(a,b)) (c1 ls pst))
   ((Ref (Bool,(a,b)) b) -> Editor (Bool,(a,b)) (c2 ls pst))
   -> ((Ref cx (EITHER a b)) -> Editor cx (:+: c1 (:+: RadioControl c2) ls pst))
altD isLeftDefault to_editorL to_editorR = to_editor
   where
   to_editor ref = (to_code, upddef, nrE)
      where
      (refC, refLR) = splitref2 idref
      (refL, refR) = splitref2 refLR
      editorE =
         (    to_editorL refL
         :++: to_editorC refC
         :++: to_editorR refR
         )
      (to_codeE, upddefE, nrE) = editorE
      upddef = ref.app (const (choose defE))
      defE = upddefE undef
      (_, (defL, defR)) = defE
      det_inits (LEFT val) = (True, (val, defR))
      det_inits (RIGHT val) = (False, (defL, val))
      choose (b,(l,r)) = if b (LEFT l) (RIGHT r)
      to_code initcx ids =
         (controlE, to_updcv)
         where
         (controlE, to_updcvE) = to_codeE initsE ids
         initsE = det_inits (ref.val initcx)
         to_updcv wstate = ref.app (const (choose cvE))
            where
            cvE = to_updcvE wstate initsE
   to_editorC =
      setDefault isLeftDefault
         (convertF bool2int2bool (radioEditorRow ["",""]))
   bool2int2bool = (\b -> if b 1 2, \i -> i==1)
```

```
stringEditor :: (Ref cx String) -> Editor cx (EditControl ls pst)
stringEditor ref =
   ( to_code
   , ref.app (const "")
   , 1
   )
   where
   to_code initcx [cid] =
      ( EditControl (ref.val initcx) (PixelWidth 80) 1 [ControlId cid]
      , to_updcv
      )
      where
      to_updcv wstate = ref.app (const text)
         where
         (_, Just text) = getControlText cid wstate

intEditor :: ((Ref cx Int) -> Editor cx (EditControl ls pst))
intEditor = convertF (toString, toInt) stringEditor

unitLabel :: String (Ref cx UNIT) -> Editor cx (TextControl ls pst)
unitLabel label ref =
   ( to_code
   , ref.app (const UNIT)
   , 0
   )
   where
   to_code initcx [] =
      ( TextControl label []
      , \wstate -> ref.app (const UNIT)
      )

radioEditor ::
   ([String] (Ref cx Int) ->
   Editor cx (RadioControl ls pst))
radioEditor = radioEditorG (Columns 1)

radioEditorRow ::
   ([String] (Ref cx Int) ->
   Editor cx (RadioControl ls pst))
radioEditorRow = radioEditorG (Rows 1)

radioEditorG ::
   RowsOrColumns [String] (Ref cx Int) ->
   Editor cx (RadioControl ls pst)
radioEditorG rowsorcolumns labels ref =
   ( to_code
   , ref.app (const 1)
   , 1
   )
   where
   to_code initcx [cid] =
      ( RadioControl items rowsorcolumns (ref.val initcx) [ControlId cid]
      , to_updcv
      )
      where
      items = [ (label,Nothing,id) \\ label <- labels ]
      to_updcv wstate = ref.app (const radioselection)
         where
         (_, Just radioselection) = getRadioControlSelection cid wstate

run_in_dialog ::
   ((Ref a a) -> (Editor a (c a (PSt ps))))
   (a,(PSt ps))
   -> (a,(PSt ps))   | Controls c
run_in_dialog to_editor (initcx,pst)
   # ([winid:cids], pst) = openIds (nr+1) pst
     (control, to_updcv) = to_code initcx cids
     dialog =
      Dialog "" (control :+: (buttons winid to_updcv))
         [WindowId winid, WindowClose (noLS closeActiveWindow)]
   # ((_, Just newcx), pst) = openModalDialog initcx dialog pst
   = (newcx,pst)

   where
   (to_code, _, nr) = to_editor idref
   buttons winid to_updcv =
      LayoutControl (ok :+: cancel) [ControlPos (Center,zero)]
      where
      ok = ButtonControl "OK" [ControlFunction storecv]
      cancel = ButtonControl "Cancel"
         [ControlFunction (noLS closeActiveWindow)]
      storecv (cx,pst)
         # (Just wstate,pst) = accPIO (getWindow winid) pst
         # cx = (to_updcv wstate) cx
         # pst = closeActiveWindow pst
         = (cx,pst)
```

E The complete `doorEditor` example

E.1 Using only the Object I/O library

```
module doorEditorOO
import StdEnv, StdIO

mydialog (name,disturb) [idEdit,idPopUp,idDialog] =
   Dialog "" controls [WindowId idDialog]
   where
   controls =
      EditControl name (PixelWidth 80) 1 [ControlId idEdit]
      :+: PopUpControl labels (bool2int disturb) [ControlId idPopUp]
      :+: ButtonControl "OK"
             [ControlFunction okfun, ControlPos (Center,zero)]
   okfun (ls1,pst1) = (ls2,pst3)
      where
      (Just wstate, pst2) = accPIO (getWindow idDialog) pst1
      (_, Just newtext) = getControlText idEdit wstate
      (_, Just newint) = getPopUpControlSelection idPopUp wstate
      ls2 = (newtext, int2bool newint)
      pst3 = closeActiveWindow pst2

bool2int b = if b 1 2
int2bool i = (i==1)
labels = zip2 ["come on in","do not disturb"] (repeat id)

runmydialog (initval,pst)
   # (ids,pst) = openIds 3 pst
   # ((_, Just newval), pst) =
      openModalDialog initval (mydialog initval ids) pst
   = (newval,pst)

Start world = startIO NDI Void (loop initval) [] world
   where
   loop val pst
      # (val,pst) = runmydialog (val,pst)
      = loop val pst
   initval = ("Sander",True)
```

E.2 Using point-free editors

```
module doorEditor
import StdEnv, StdIO, EditorsPF

doorEditor = stringEditor :&: disturbEditor
   where
   disturbEditor =
      convertF (bool2int,int2bool)
         (dropdownEditor ["come on in","do not disturb"])
      where
      bool2int d = if d 1 2
      int2bool i = (i==1)

Start world = startIO NDI Void (loop initval) [] world
   where
   loop val pst
      # (val,pst) = run_in_dialog doorEditor (val,pst)
      = loop val pst
   initval = ("Sander",True)
```

E.3 Notes

Both implementations open the editor dialog with initial value `("Sander",True)`. When the user closes this dialog, a closing value is returned. To show what this value is, the same dialog is run again, but now with this new value as its initial value. This is performed ad infinitum by the function `loop`.

The function `Start` is the main function, which is evaluated when a Clean module is run. It uses the function `startIO` from the Object I/O library, which starts an Object I/O process.

Note the use of Clean's let-before construct, marked with a `#`. The scope of the identifiers in the lhs of a definition following `#` does not include the rhs of the definition itself or the definitions above it, but does include the definitions below it. This way we can use the same name `pst` for the values of the process state at different stages. The same goes for the local state `ls`.