Generic Editors for the World Wide Web

Rinus Plasmeijer and Peter Achten

Software Technology, Nijmegen Institute for Computing and Information Sciences, Radboud University Nijmegen {rinus, P.Achten}@cs.ru.nl

Abstract. In these lecture notes we present a novel toolkit to program web applications with that have dynamic, complex behavior based on interconnect forms. This toolkit is the iData Toolkit. We demonstrate that it allows programmers to create web applications on a level of abstraction that is comparable with ‘ordinary’ functional style programs. Programmers have the freedom to develop their own data structures and program with them. The iData Toolkit is able to generate a web application from these data types, making use of advanced programming concepts such as generic programming. The programmer need not be aware of this. We have tried to keep the iData Toolkit API as simple as possible.

1 Introduction

The World Wide Web has become an important infrastructure for institutions such as universities, government, and industry and individuals to provide and obtain information from a wide variety of sources. The complexity of web sites range from simple static collections of HTML pages to advanced interactive sites with many interconnections and user feedback. In these notes we intend to target web applications that have complicated relations between an arbitrary number of forms. As a typical example, we have constructed a web shop application for selling CD’s. Fig. 1 contains a screen shot of this application. It displays a number of interconnected form elements, some of which are: application browsing buttons labelled Home, Shop, Basket, OrderInfo and the page to be displayed; search fields and the number of found and displayed items, as well as the range of selection browser buttons; adding elements to the shopping cart and the most recently added item. In addition, it shows that there also elements that are not related with forms, but rather with layout and make up. We consider these elements to be purely functionally dependent from the actual state of the forms.

In these lecture notes, we study web applications from the perspective of the functional programming paradigm. Key aspects that we pay attention to are the functional nature of a web application, exploitation of the expressiveness and richness of the type system, and the power of abstraction and composition. The main goal is to obtain a programming toolkit, the iData Toolkit, that provides web application programmers with a means to express themselves on a high level of abstraction, without compromising the complexity of the applications. The API of the iData Toolkit contains no advanced concepts; we deliberately have kept it as simple as possible. The internal realization on the other hand makes
Fig. 1. A Web shop application, programmed with the iData Toolkit.

good use of advanced functional programming concepts, the most important one being **generic programming** \([11,12]\). We do not discuss the implementation, but focus on the application programmer instead. We have collected a number of examples and exercises so that the reader can obtain practical knowledge and insight in the toolkit.

We use the functional programming language **Clean** \([18,19]\), version 2.1.1. **Clean** is a pure, lazy, functional programming language based on term graph rewriting. It has an expressive type system with support for **generic programming** \([2]\), dynamic types, uniqueness types, strictness types, and more. It efficiently generates efficient code. For over a decade it supports desktop GUI programming with the Object I/O library. With this library its own IDE has been implemented, as well as the proof assistant **Sparkle**. The **Clean** compiler has been written in **Clean** itself. We assume that the reader is already familiar with **Clean** and functional programming. The **Clean** programming environment can be downloaded for free from http://www.cs.ru.nl/clean/.

These notes are structured as follows. In Sect. 2 we analyze the challenges that a web application programmer is confronted with. This analysis is independent of any programming paradigm. In Sect. 3 we provide one of the many possible
answers in the context of the programming paradigm of lazy, strongly typed, functional programming languages. Having motivated our design decisions, we work out our solution with a number of case studies of increasing complexity and completeness of functionality in Sect. 4. Every case ends with a set of exercises that allow the student to practice his skills and understanding. Of course, this work is not the first in this domain, and neither will it be the last. Related work is discussed in Sect. 5. Finally, we come to conclusions in Sect. 6.

2 The Challenge...

When the World Wide Web was conceived by Tim Berners-Lee around 1990, it was intended to be a uniform portal for people to find information around the world [5]. Fig. 2 illustrates the ‘classic’ architecture with a simple Message Sequence Chart (MSQ). A browser application (B) allows users to communicate with a web server (S). The web server retrieves the requested information that is stored somewhere on the server side. This information is encoded in HTML, and is interpreted and displayed by the browser.

<table>
<thead>
<tr>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg1</td>
<td>req1</td>
</tr>
<tr>
<td>msg2</td>
<td>req2</td>
</tr>
<tr>
<td>msg3</td>
<td>req3</td>
</tr>
</tbody>
</table>

Fig. 2. ‘Classic’ W³.

B | S | A₁ | Aₙ |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>msg1</td>
<td>req1</td>
<td>passe</td>
<td>terminate</td>
</tr>
<tr>
<td>msg2</td>
<td>req2</td>
<td>passe</td>
<td>terminate</td>
</tr>
<tr>
<td>msg3</td>
<td>req3</td>
<td>passe</td>
<td>terminate</td>
</tr>
</tbody>
</table>

Fig. 3. ‘Contemporary’ W³.

The need for web sites with dynamic content soon arose after this simple scheme. One of the standards that emerged was the Common Gateway Interface (CGI). The key idea is that instead of the web server retrieving static web pages itself, it communicates with a bunch of applications that it executes that will provide the requested information. In this way, the web server acts as an intermediary between the browser program and the CGI programs. This is depicted in Fig. 3. The collection of applications are labelled A₁ upto Aₙ. The dashed lines indicate that these applications are executed on request, and terminate after having produced the desired result. These applications can be generated by a programming language, or dedicated scripting languages such as php, Perl, or in a functional style, such as WASH/CGI. Complex web applications often consist of several dynamic applications, or separate scripts. This makes reasoning about their behavior and correctness difficult.
The architecture of the web leads to a number of challenges for web application developers:

1. **Cause and Effect**
   The specification of every interactive application must clearly and unambiguously prescribe the behavior of the application in case of particular user actions. It should be obvious that this depends on the state of the application, as pressing the mouse button in an enabled button has an entirely different meaning than pressing it in a scroll bar.
   In traditional desktop GUI programming, both concepts of application state and user actions (events) have been well-defined. The state is partially determined by the underlying system (e.g. widgets, rendering environments, resources, customizable widgets), and partially by the application (think of data structures, variables with scope rules). The set of events is fixed by the underlying system (keyboard and mouse events, rendering requests, message passing events for customization purposes, and so on). This implies that it is clear (but not necessarily easy) how to map the specification of an application to a working implementation.
   The web has no concept of state. This means that a programmer has to prepare his own infrastructure to realize a suitable state. Many techniques have been explored for this purpose: cookies, server side database, data storage in web pages, extensive use of the expressive power of XML. The web has a weak notion of event: forms can trigger parameterized requests to the web server to fetch new pages, depending on the parameters of the request. As a consequence, it is a challenge for a web application programmer to create a correct implementation of a specified interactive application.

2. **Accumulating Behavior**
   This challenge is related to 1. During execution, applications gather data. This data determines the future behavior of the application. Clearly, this requires state that is preserved during the run-time of the application. The web application programmer needs to make sure his data persists between invocations of his application.

3. **User Behavior**
   The web deliberately allows users great freedom in browsing through information that is available all over the world. Users bookmark links and visit them arbitrarily many times later. This implies that web applications can not assume that pages are always reached via a well-defined route. Users simply stop browsing by closing their browser program. This means that web applications are not closed-down gracefully as is the case with desktop GUI applications. These can decide what data should be stored persistently, and what should be garbage collected.

4. **(Dependent) Forms**
   The interactive parts of a web application are defined by forms. A form is a collection of primitive interactive elements such as edit boxes, check boxes, radio buttons, and so on. This is fairly similar to desktop GUls. However, in a desktop GUI these elements can be considered to be objects that can
be manipulated by the application during run-time. Because the web lacks state, this can not be done with web applications. The programmer can not manipulate these elements as if they existed. Instead, they need to be recreated every time a new page needs to be displayed. This becomes even more complicated when forms depend on each other, i.e. data entered in one form influences its own state, or also the existence or state of other forms.

5. Separation of Model and View
Designing an attractive web application that is functionally complete is a difficult task. The maintenance of a web application may require either a change of its presentation, a change of functional requirements, or both. A good separation of presentation and application logic is important to reduce the maintenance effort. Changing the presentation of an application should cause at worst only minor changes in the application logic, and vice versa. Of course, this is not specific for web applications, but also applies to desktop GUI applications.

3 ... A Functional Style Answer
In this section we rise to the challenges that have been presented in Sect. 2. Of course many people have already provided answers to these challenges in many different and excellent ways. We review related work in Sect. 5. In these notes we give an answer from the perspective of the functional programming paradigm. We first introduce the leading functions and types in Sect. 3.1. Having been exposed to the main elements of the iData Toolkit, we show how these elements provide answers to the challenges in Sect. 3.2.

3.1 Introducing the Leading Figures
In this section we introduce the leading figures of the iData Toolkit in a top-down style. Recall the way the contemporary web works (Fig. 3). A web application \( A \) may consist of several smaller scripts, i.e. \( A = \{A_1 \ldots A_n\} \). Our view of the structure of a web application is depicted in Fig. 4. A web application \( A \) should be a single application. Of course, its code may consist of several modules, but the program as a whole is considered to be a single unit that is addressed by the web server as a single executable. In the functional paradigm, a program is a function, depicted as \( f \) in the diagram. Clearly, this \( f \) is in dire need of parameters if it is not to produce the same page at every invocation.

What is the type of \( f \)? Every interactive Clean program is a function of type \( \text{\#World} \rightarrow \text{\#World} \). However, this is not a very precise type. We need a type that expresses more clearly what the purpose is of a web application. In our view, the purpose of a web application is to provide the user with the illusion of an application that does not terminate between user actions, but instead can be regarded as a collection of objects that respond to his actions as if it were a regular desktop GUI application. Put differently, if it had a previous state of type \( \text{St} \), then it must produce a new state of that type as well as a rendering in HTML of that state. This leads to the following type of \( f \):
This type is connected with the type of an interactive Clean program with the following wrapper function:

\[
\text{doHtml} :: (*\text{HSt} \rightarrow (\text{Html}, *\text{HSt})) \rightarrow *\text{World} \rightarrow *\text{World}
\]

The arguments of \( f \) come from the command-line. At every execution, the command-line contains a serialized representation of the current state of the application, as well as the event that has occurred. We show that in our view an event is a change of the value of the state. Put in other words, the user has edited the state. We have set up the system in such a way that edit operations always result in a new value of the same type. This is a powerful means of abstraction that helps programmers to reduce the implementation effort. The underlying technology that has enabled such a type driven and type safe system is the generic programming facility of Clean.

We will show that the function body of \( f \) typically consists of two steps: updating the forms of the web application, and updating the HTML page, using the updated forms. To the programmer a form is an object that has a certain model value that is rendered according to a view value. Both the model data type and the view data type are specified using custom data types. Their relative mapping is of course specified using functions between these data domains. The iData Toolkit has one pivot function that implements a complete form for any given model value and model-view relationship:

\[
\text{mkViewForm} :: \text{FormId} m (\text{HBimap m v}) *\text{HSt} \rightarrow (\text{Html}, *\text{HSt}) | g\text{Html}[m] | v
\]

The generic class gHtml is the silent witness of the fact that the iData Toolkit makes extensive use of generic programming. Its definition encompasses as much as four generic functions:

\[
\text{class gHtml t l gForm, gUpd, gPrint, gParse t}
\]

As a technical aside: at this stage Clean does not allow the definition of generic classes. In this writing we will use it as a shorthand notation. In the true libraries you will see the expanded list of generic functions.
In the remainder of this sequel the exact meaning of \( \text{mkViewForm} \) will be explained. Here we suffice with a rather informal specification based on its type signature: \( \text{FormId} \) identifies the form elements and their main attributes in a web application’s page, \( m \) is the initial model value, and \( \text{(HBinap } m \text{ v)} \) is the collection of relational functions between model and view. The result is a new form \( (\text{Form } m) \) that contains a model value, an HTML implementation, and a flag that tells whether the user has changed the form.

One important form attribute is the *life span* of the form’s state. The programmer has fine grained control over this state. He can decide to make it fully persistent, or persistent only during the session of the application, or make it live only during the page itself.

Because of its generality, \( \text{mkViewForm} \) is not always easy to use. Therefore it has a couple of friends that capture a number of frequently occurring model-view patterns. The simplest one of these friends is \( \text{mkEditForm}: \)

\[
\text{mkEditForm} :: \text{FormId } m \times \text{HSt} \to (\text{Form } m, \text{HSt}) \mid \text{getHtm}\{id\} m
\]

Given the identification of a form id, an initial model value \( m \), \( (\text{mkEditForm } \text{id } m) \) creates a form that renders \( m \) and allows the user to manipulate this value.

### 3.2 The Challenges

We now have introduced the key elements of the iData Toolkit. It is about time to demonstrate how they will aid us in tackling the challenges that have presented in Sect. 2.

**Cause and Effect:** From the above account we can conclude that we have built a standard framework for building web applications that deploy objects with typed state. Also, a very clear notion of events has been introduced: the edit action of a user that changes the current state of an object into another.

**Accumulating Behavior:** The state of objects can be arbitrarily complex. The programmer also has fine grained control over the life span of these objects. States may be fully persistent, only during a session, or only during on page. This is expressed as an attribute of the form object.

**User Behavior:** In the iData Toolkit, the programmer can clearly identify the ‘dangerous’ parts of his web application. If all form states have a page based life span, then this means that the full state of the web application is in its pages. In that case, the application is certain to be correct with respect to arbitrary user behavior. Things get more complicated in case of (session) persistent state. These states are made explicit, and require special attention.

**(Dependent) Forms:** A web application is a single function that updates its form always in the same specified order. This means that the programmer imposes a functional relationship between updated forms and their values. In this way, complicated connections can be realized in a functional style. In addition, forms can be constructed that behave like memory storages, and that can be manipulated as such. This increases their flexibility.
Separation of Model and View: From the start, the iData Toolkit merges the model-view paradigm with the concept of forms. A form is always defined in terms of a visualization of some model data. This is embodied with the powerful function mkViewForm. For simpler situations, wrapper functions are provided that give easier access to this scheme.

4 Case Studies

In this section we construct a number of case studies, each of which focusses on one particular aspect of web programming with the iData Toolkit. As a brief overview, we will go through the following steps: we start with programming HTML directly in the sections 4.1 through 4.3. Once we know how to play with HTML, we can concentrate on programming forms, in the sections 4.4 through 4.6. Finally, we give one larger example in Sect. 4.7.

4.1 Programming HTML

In Sect. 3.1 we have shown that an iData Toolkit web application programmer really creates a function of type (*HSt ^  (Html, *HSt) ). Such a function f is turned into an interactive application by (doHtml f) :: *World ^  *World. The abstract type HSt collects all the form information during the construction of a HTML page. We defer its discussion until Sect. 4.4. Html is the root type of a collection of algebraic data types (ADT) that capture the official HTML.

```
:: Html  = Html Head Rest
:: Head  = Head  [HeadAttr]  HeadTag
:: Rest   = Body  [BodyAttr]  BodyTag
| Frameset [FramesetAttr]  [Frame
:: Frame  = Frame  [FrameAttr]
| NoFrames [Std_Attr]  [BodyTag
:: BodyTag  = A  [A_Attr]  [BodyTag]
| Var [Std_Attr]  String
| STable  [Table_Attr]  [[BodyTag]]
| BodyTag [BodyTag]
| EmptyBody
```

BodyTag contains the familiar HTML tags, starting with anchors and ending with variables (in total there are 76 HTML tags). The latter three alternatives are for easy HTML generation: STable generates a 2-dimensional table of data, BodyTag collects data, and EmptyBody can be used as a neutral element. Attributes are encoded as FooAttr data types.

As an example, the following Html value:

```plaintext
hello = Html (Head ['HdStd [Std_Title "Hello World Example"]]) (Body []) (Txt "Hello World!")
```
is translated to the following HTML:

```html
<html>
<head title = Hello World Example></head>
<body>Hello World!</body>
</html>
```

In order to get rid of some standard overhead HTML code, the following two functions prove to be useful:

```haskell
mkHtml :: String [BodyTag] *HSt -> (Html,*HSt)
mkHtml s tags hSt = (simpleHtml s tags,hSt)

simpleHtml:: String [BodyTag] -> Html
simpleHtml s tags = Html (header s) (body tags)
where header s = Head [Hdr Std [Std_Title s]] []
       body tags = Body [] tags
```

With these functions, the above example can be shortened to:

```haskell
hello = mkHtml "Hello World Example" [Txt "Hello World!"]
```

What needs to be done is to really create a HTML file. This is one of the things done by doHtml. The complete code for this example then is:

```haskell
module fragments
import StdEnv    -- import the standard Clean modules
import StdHtml   -- import the iData Toolkit modules

Start world = doHtml hello world
where hello = mkHtml "Hello World Example" [Txt "Hello World!"]
```

As this example shows, HTML can be encoded straightforwardly into an ADT. There are some minor complications. In Clean, as well as in Haskell [17], all data constructors have to be different. In HTML, the same attribute names can appear in different tags. Furthermore, certain attributes, such as the standard attributes, can be used by many tags. We do not want to repeat all these attributes for every tag, but group them in a convenient way. To overcome these issues, we use the following naming conventions:

- The data constructor name represents the corresponding HTML language element.
- Data constructors need to start with an uppercase character and may contain other uppercase characters, but the corresponding HTML name is printed in lower-case format.
- To obtain unique names, every data constructor name is prefixed in a consistent way with Foo_. When the name is printed we skip this prefix.
- A constructor name is prefixed with ' in case its name has to be completely ignored when printed.
We have defined one generic printing routine \texttt{gHpr} that implements the naming conventions that have been discussed above, and prints the correct HTML code. Its definition is not relevant here.

\begin{verbatim}
generic gHpr a :: *File a -> *File
\end{verbatim}

Our approach has the following advantages:

\begin{itemize}
\item One obtains a grammar for HTML which is convenient for the programmer.
\item The type system eliminates type and typing errors that can occur when working in plain HTML.
\item We can define a type driven generic function for generating HTML code.
\item Future changes of HTML are likely to change the ADT only.
\item We can use the expressive power of our programming language to create complex HTML.
\end{itemize}

\section{4.2 Deriving HTML From Types}

In the previous section we have demonstrated how one can construct HTML pages in a typed style. Although the use of a typed language eliminates many errors, it is not convenient to program web pages in this way. Instead, we like to \textit{generate} HTML for data of arbitrary type automatically. This reduces the effort of creating web pages, reduces the risk of making errors, and increases consistency. Because this transformation is intended to work for every type, it has to be a generic function:

\begin{verbatim}
toHtml :: a -> BodyTag | gForm[\$\$] a
\end{verbatim}

Actually, it uses the generic function \texttt{gForm}. This function is introduced in Sect. 4.4. One of its purposes is to generate HTML for any data value. It is this aspect that is used by \texttt{toHtml}.

Let's start with simple stuff, and move towards more complicated types. Table 5 shows the generated HTML for the basic types \texttt{Bool}, \texttt{Int}, \texttt{Real}, and \texttt{String}. These are obtained by replacing \texttt{expr} in:

\begin{verbatim}
Start world = doHtml (mkHtml "Example" [toHtml expr]) world
\end{verbatim}

\texttt{expr} = \texttt{True}
\texttt{expr} = \texttt{42}
\texttt{expr} = \texttt{42.0}
\texttt{expr} = \texttt{"Hello World!"}

Fig. 5. HTML generated by \texttt{toHtml} for several basic types.

Next we proceed with types that have been composed by the standard type constructors of \texttt{Clean}. Filling in \texttt{expr = ("\$fib \ "$\$40\$" = ",\$fib 40)), which has type \texttt{(String,Int)} yields the following HTML output:
The operator <$ is a useful abbreviation for the frequently occurring case of concatenating a String value with something that can be turned into a String:

\[
 <\$ \quad \text{infixl} :: \text{String} \oplus \text{a} \rightarrow \text{String} \\
 <\$ \quad \text{str} \ x = \text{str} +++ \text{toString} \ x
\]

Lists are internally represented as :: [a] = _Cons a [a] | _Nil. This algebraic structure is clearly visible in case of \( \text{expr} = [1 .. 6] \), which has type [Int]:

Note that the iData Toolkit derives and defines instances of the generic function \( gForm \) for the basic types and \( \{2,3,4\}-\)tuples, but not for lists. For this reason, you need to include the following in your code:

\[
\text{derive} \ gForm \ []
\]

In order to demonstrate the compositional character of the mechanism, let’s create a list of pairs of strings and integers, i.e.: it has type \( \{\text{String,Int}\} \). An example value is \( \text{expr} = [\{ \text{"Nfib" <$n<$" = ", Nfib n} \ \backslash \ n \leftarrow [0.5 .. 30] \}]:\)

Records are algebraic data types that have exactly one (invisible) data constructor, and labelled fields. The iData Toolkit displays the field names, together with the field values, in the same recursive way. Consider for instance the following record type for a simplistic personnel administration, and two values of that type:

\[
:: \text{Person} = \{ \text{name::String}, \text{address::String}, \text{city::String} \} \\
\text{derive} \ gForm \ \text{Person}
\]

\[
\text{peter = \{name="Achten", address="Abersland", city="Wijchen"\}}
\text{rinus = \{name="Plasmeijer", address="Knollenberg", city="Mook"\}}
\]
Then the value `expr` = `peter` is displayed as:

<table>
<thead>
<tr>
<th>Name</th>
<th>Achten</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Abersland</td>
</tr>
<tr>
<td>City</td>
<td>Wijchen</td>
</tr>
</tbody>
</table>

Exercises

1. **Lists, differently**

The example of a list of `Nfib` numbers was created with:

\[
\text{expr} = \left[(\text{"Nfib" } \langle \text{n} \rangle = \text{" \text{Fib}\text{ n}\text{"}}) \ \& \ \& \ n \rightarrow [0..30]\right].
\]

What happens with the generated HTML if you replace `[toHtml expr]` with

\[
[toHtml (\text{"Nfib" } \langle \text{n} \rangle = \text{" \text{Fib}\text{ n}\text{"}}) \ \& \ \& \ n \rightarrow [0..30]\right]?
\]

4.3 **More Fun With HTML**

The previous section has demonstrated that the iData Toolkit is able to derive a HTML representation for any conceivable custom data type \( T \) as long as you don’t forget to include a `derive gForm \( T \)`. It also demonstrates that the resulting HTML representations are not always attractive. In this section we introduce a few `body tag` combinators that give you more control over the layout of the elements of a HTML page.

The main program that we use in this section is slightly different from the previous one:

\[
\text{Start world} = \text{doHtml (mkHtml "Example" [expr]) world}
\]

We start with the combinator `<.=.>` that places two body tag elements next to each other, so \( (b1 <.=.> b2) \) places \( b2 \) next to \( b1 \). Its implementation uses the `BodyTag alternative STable` with which tables (list of rows of body tag elements) can be created. The variant `<>` that works for `bodytag lists` is easily defined:

\[
(<.=.>) \ \text{infixl 5} :: \text{BodyTag BodyTag} \rightarrow \text{BodyTag}
\]

\[
(<.=.>) \ b1 \ b2 = \text{STable} \ | \ \text{Tbl_CellPadding \ (Pixels 0)}
\]

\[
\text{, Tbl_CellSpacing \ (Pixels 0)}
\]

\[
\text{| \ [b1, b2]}]
\]

\[
(<>\) \ \text{infixl 5} :: \text{[BodyTag] \ [BodyTag]} \rightarrow \text{BodyTag}
\]

\[
(<>\) \ b1 \ b2 = (\text{BodyTag b1}) <.=.> (\text{BodyTag b2})
\]
With these combinators, we can easily place two values next to each other. Consider for instance \texttt{expr= toHtml peter <.=.> toHtml rinus}. This yields the following HTML:

\[
\begin{array}{ll}
\text{Name} & \text{Achten} \\
\text{Address} & \text{Abersland} \\
\text{City} & \text{Wijchen}
\end{array} \quad \begin{array}{ll}
\text{Name} & \text{Plasmeijer} \\
\text{Address} & \text{Knollenberg} \\
\text{City} & \text{Mook}
\end{array}
\]

This suggests that if you have a list of items that you want to display in a single row, it is sufficient to turn them into HTML elements first (using \texttt{map toHtml}), and then replacing every \texttt{cons} by \texttt{<.=.>} (by folding \texttt{<.=.>} over the resulting list). This is done with the iData Toolkit function \texttt{mkRowForm}, defined concisely as:

\[
\texttt{mkRowForm : : [BodyTag] \rightarrow BodyTag} \\
\texttt{mkRowForm xs = foldr (\texttt{<.=.>}) \texttt{EmptyBody xs}}
\]

As an example, to produce a horizontal list of integer elements, one can define \texttt{expr= mkRowForm (map toHtml [1 .. 7])}, and get:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

In exactly analogous ways, we can do this for \textit{vertical} layout, and introduce:

\[
\begin{array}{l}
\texttt{(<||>) \quad \texttt{infixl 4 : : [BodyTag] \rightarrow [BodyTag]} \\
\texttt{(<||>) b1 b2 = (\texttt{BodyTag b1}) <.||.> (\texttt{BodyTag b2})}
\end{array}
\]

\[
\begin{array}{l}
\texttt{mkColForm : : [BodyTag] \rightarrow BodyTag} \\
\texttt{mkColForm xs = foldr (\texttt{<||>}) \texttt{EmptyBody xs}}
\end{array}
\]

With this combinator, we can create a more appealing representation of the list of \texttt{nfib} numbers that was given earlier. We define:

\[
\texttt{expr= mkColForm (map toHtml [("Nfib "$n," = ",Nfib n) \ \ n \in \ [0..10]]).}
\]

This yields:

\[
\begin{array}{l}
\text{Nfib 0} = 1 \\
\text{Nfib 1} = 1 \\
\text{Nfib 2} = 3 \\
\text{Nfib 3} = 5 \\
\text{Nfib 4} = 9 \\
\text{Nfib 5} = 15 \\
\text{Nfib 6} = 25 \\
\text{Nfib 7} = 41 \\
\text{Nfib 8} = 67 \\
\text{Nfib 9} = 109 \\
\text{Nfib 10} = 177
\end{array}
\]
Finally, elements can be arranged in a table, using the function `mkSTable`:

```haskell
mkSTable :: [[BodyTag]] → BodyTag
mkSTable table = Table [] (mktable table)
where mktable table = [Tr [] (mkrow rows) \ rows ← table]
      mkrow rows = [Td [] [Td_VAlign Alo_Top,
                        Td_Width (Pixels defpixel),
                        [row]\ rows ← rows]

The exercises below use this function.

Exercises

2. Table headers

The function `mkSTable` displays a table of elements, arranged as a list of rows. Write a function `(augmentTable h v t)` that augments a table `t` (a value of type `[[BodyTag]]`) with a horizontal header `h` (of the proper length) and a vertical header `v` (of the proper length).

`augmentTable :: [BodyTag] [BodyTag] [[BodyTag]] → [[BodyTag]]`

3. The Ackermann function

Write an application that uses the function `augmentTable` to show the Ackermann `i j` values with `i ∈ {0...3}` and `j ∈ {0..7}`. The Ackermann function is defined as follows:

```haskell
Acker :: Int Int → Int
Acker 0 j = j + 1
Acker i 0 = Acker (i - 1) 1
Acker i j = Acker (i - 1) (Acker i (j - 1))
```

The Ackermann function is well-known in theoretical computer science, because it was presented as a counter example of the thesis that every computable function could be expressed as a primitive recursive function. The Ackermann function is well-defined, computable, but not primitive recursive.

The application should look something like:

![Ackermann Table]
4.4 Programming Direct Forms

In the previous sections we have had experience with programming HTML pages that consist of explicit HTML as encoded by the Html type and friends, and generated HTML using the generic function toHtml. It is time now to do form programming. In web terminology, a form is a collection of interactive elements such as buttons, check boxes, radio buttons, edit text fields, and so on. The user can manipulate these elements. Depending on the application (for instance by pressing the submit button), at some point in time the results of these manipulations are sent to the web server which starts the web application and provides it with this information. The application processes the information and responds with a new page that is presented to the application user. In this way, interaction has been achieved between a user and a web application.

The key idea of the iData Toolkit is that a page is represented by a value of some type. Therefore, a form is also represented by a value of some type. User manipulations are really edit operations that modify the value into another value of the same type. Hence, a form is an editor of values of some type. This definition is reflected in the type signature of the function mkEditForm:

\[
\text{mkEditForm} :: \text{FormId} d \times \text{HSt} \rightarrow (\text{Form} d, \text{HSt}) \mid \text{gHTML}\{d\} d
\]

Recall from Sect. 3.1 that the class gHtml is really a (syntactically illegal) shorthand for a collection of four generic functions, one of which is gForm. Above we have used one of the functionalities of gForm. This is a generic function of signature:

\[
\text{generic gForm} d :: \text{FormId} d \times \text{HSt} \rightarrow (\text{Form} d, \text{HSt})
\]

\((\text{gForm id dv hSt})\) creates a form (an editor) that is identified with id, and that has initial value dv. With an identification value of type FormId, the application programmer sets a number of mandatory attributes of each form. The following types are involved:

\[
\begin{align*}
\text{FormId} & = \{ \text{id} :: \text{String} \\
& \quad , \text{lifespan} :: \text{Lifespan} \\
& \quad , \text{mode} :: \text{Mode} \} \\
\text{Lifespan} & = \text{Persistent} \mid \text{Session} \mid \text{Page} \\
\text{Mode} & = \text{Display} \mid \text{Edit}
\end{align*}
\]

The first mandatory attribute is an identification tag \((\text{id} :: \text{String})\). This tag must unambiguously identify the form in the collection of forms used by the web application. This is the responsibility of the programmer. The second mandatory attribute is the life span \((\text{lifespan} :: \text{Lifespan})\) of the form. The life span states how long the current value of the form (its state) lives: Persistent values live ‘forever’ and reside on disk, Session values live during a session, and Page values live only during the life span of the page that they are part of. Finally, the third mandatory attribute is the mode \((\text{mode} :: \text{Mode})\) of the form. All of the examples above displayed values: the user is not able to manipulate them. This is in fact
set by the toHtml function that calls gForm. For a form it makes more sense to allow
user manipulations, and set the mode to Edit instead.

The iData Toolkit provides a few constructor functions to easily create FormId
values:

\begin{verbatim}
  nFormId :: String -> FormId /* Page + Edit */
  sFormId :: String -> FormId /* Session + Edit */
  pFormId :: String -> FormId /* Persistent + Edit */
  ndFormId :: String -> FormId /* Page + Display */
  sdFormId :: String -> FormId /* Session + Display */
  pdFormId :: String -> FormId /* Persistent + Display */
\end{verbatim}

The form that is returned by gForm is a small record type:

\begin{verbatim}
  data Form d = { changed :: Bool,
                 value :: d,
                 form :: [BodyTag]
                           }
\end{verbatim}

A form may have been edited by the user. This is set in the changed field of
the form. Forms always have a value of the type that is associated with them.
This is set in the value field. Finally, a form needs an HTML rendering. This is
set in the form field. In fact, the [BodyTag]s that we have used in the examples
above come from this field. Except for the HSt parameter, we can now explain
the function toHtml:

\begin{verbatim}
  toHtml :: a -> BodyTag | gForm{|*|} a
  toHtml a
  %{ (form,_) = gForm{|*|} {id="__toHtml",lifespan=Page,mode=Display} a ...
   = BodyTag form

  From this we can conclude that in all examples that have been presented so
far, it is OK to replace toHtml with form versions, using mkEditForm, and plug in
their HTML renderings using the form fields of these forms.

As a first example of a web application with a direct form, we create a
FormField Person (Person and rinus were defined at the end of Sect. 4.2). For com-
pleteness, we give the full code:

\begin{verbatim}
  module fragments
  import StdEnv
  import StdHtml

  Start world = doHtml personPage world
  where personPage hSt
        %{ (person,hSt) = mkEditForm (nFormId "person") rinus hSt
            = mkHtml "Person" [ H1 [], BodyTag person.form
                              ] hSt

  :: Person = Person definition here
\end{verbatim}
derive gForm Person; derive gUpd Person; derive gPrint Person; derive gParse Person

The example shows that the HTML rendering of a form \( f \) can be used at any arbitrary location, just by taking the \( f.form \) field of that form. Because \texttt{mkEditForm} relies on a collection of generic functions, we also need to derive instances for these functions for \texttt{Person}. This should become standard idiom when defining new types for forms.

Because we have created an \textit{editable} form, the behavior of this program is quite different from the one that only displays a person. This is what it looks like initially:

![Image of a person form]

This application allows the user to edit any of the fields of a person record. An edit operation is finished as soon as the user 'confirms' editing by leaving the input focus of the edit box, and not during every keystroke or copy-paste action).

Despite its size, this example shows the general structure of web applications with forms. The function of type \( \ast \text{HSt} \to (\text{Html}, \ast \text{HSt}) \), \texttt{personPage} in the example, that defines the page first needs to update its forms, and then updates the HTML that corresponds with the new forms.

\begin{verbatim}
myPage :: \ast \text{HSt} \to (\text{Html}, \ast \text{HSt})
myPage hSt
  \{ (forms,hSt) = updateForms hSt
    = updatePage forms hSt

In the remainder of this sequel, we adopt this scheme and modify the \texttt{updateForms} and \texttt{updatePage} functions. In the example, \texttt{updateForms} is simply
\end{verbatim}

\begin{verbatim}
  (mkEditForm (nFormId "person") rinus)
\end{verbatim}

and \texttt{updatePage} is

\begin{verbatim}
  (\lambda person hSt -> mkHtml ... hSt).
\end{verbatim}

Here is an example of a slightly more interactive web application. We extend the \textit{nfib} table example from Sect. 4.3 with an integer form in which the application user can enter a number \( n \). As a result, the application shows all \textit{nfib}
numbers from 0 to \( n \). For this also a form is used. Let's first construct the two forms:

\[
\text{updateForms :: } *\text{HSt} \rightarrow ((\text{Form \ Int}, \text{Form \ [(String,String,Int)]}), *\text{HSt})
\]

\[
\text{updateForms \ hSt} = \begin{cases} 
  (\text{rangeF}, \text{hSt}) &= \text{mkEditForm (nFormId "range") 10 \ hSt} \\
  (\text{nfibF}, \text{hSt}) &= \text{mkEditForm (ndFormId "nfib") \ ([ ("Nfib \ " <\$\ n, \ " = \", \ Nfib \ n) \\
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ n-[0..\text{rangeF.value}]
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \\text{\ hSt}]
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{hSt})}
\end{cases}
\]

\[
= ((\text{rangeF}, \text{nfibF}), \text{hSt})
\]

\[
\text{derive \ gForm \ []; \ derive \ gUpd \ []}
\]

The integer form \text{rangeF} is straightforward. Its initial value is 10, and it is identified with tag "range". The \text{nfib} table form \text{nfibF} is more interesting, as its construction clearly depends on the value of \text{rangeF}. It is identified by tag "nfib", and is not editable.

Given these two forms, \text{updatePage} needs to create the proper web page:

\[
\text{updatePage :: } (\text{Form \ Int}, \text{Form \ [(String,String,Int)]}) *\text{HSt} \rightarrow (\text{Html}, *\text{HSt})
\]

\[
\text{updatePage \ (rangeF,nfibF) \ hSt} = \begin{cases} 
  \text{mkHtml "NFib" \ [ H1 \ [Hnum_Align Aln_Center] "NFib numbers"} \\
  , \text{Txt "Enter a positive number (not too large please)."} \\
  , \text{BodyTag rangeF.form} \\
  , \text{Br} \\
  , \text{Txt ("The Nfib numbers from 0 to " <\$\text{rangeF.value}" are:"}) \\
  , \text{Br} \\
  , \text{mkColForm (map toHtml nfibF.value)}
\end{cases}
\]

\[
\text{hSt}
\]

The interesting bits are the fact that from \text{rangeF} we use both its form and value, and from the \text{nfibF} only its value. Note the arbitrary order in which these elements are mixed with static HTML code.

The initial look of this application is given below, together with its look after the user has edited the integer form into the value 15.
In the beginning of this section we have defined forms to be collections of interactive elements such as buttons, check boxes, radio buttons, and so on. We introduce these elements right now. As you may gather by now, we have no intention in programming these kinds of elements directly in HTML (which is perfectly possible because we have full HTML at our disposal), but rather program them by means of values of types that provide a higher level of abstraction.

Let's start with push buttons. A push button either has a text label label and a certain width width, and is defined as \((\text{LButton width label})\), or it uses an image file located at path and with dimensions dim and is defined by \((\text{PButton dim path})\). When the user presses the button, it enters the \text{Pressed} state. After processing by the \text{iData Toolkit}, it returns to its previous state. This gives the following, compact definition of a button:

\[
:: \text{Button} = \text{Pressed} \mid \text{LButton Int String} \mid \text{PButton (Int,Int) String}
\]

Fig. 6 gives the result of the following definitions (assuming that the local files "rinus.jpg" and "peter.jpg" exist and contain sensible material):

\[
\text{updateForms :: } \ast\text{HSt} \rightarrow ([\text{Form Button}], \ast\text{HSt})
\]

\[
\text{updateForms hSt} = \begin{cases} 
(rP,hSt) & = \text{mkEditForm (nFormId "rinusB") (PButton dim "rinus.jpg") hSt} \\
(pP,hSt) & = \text{mkEditForm (nFormId "peterB") (PButton dim "peter.jpg") hSt} \\
(rL,hSt) & = \text{mkEditForm (nFormId "rinusL") (LButton (snd dim) "Rinus") hSt} \\
(pL,hSt) & = \text{mkEditForm (nFormId "peterL") (LButton (snd dim) "Peter") hSt}
\end{cases}
\]

\[
= ([rP,pP,rL,pL],hSt)
\]
updatePage :: [Form Button] *HSt → (Html,*HSt)
updatePage [rP,pP,rL,pL] hSt
  # [rP,pP,rL,pL:_] = map (λx → BodyTag f.form) [rP,pP,rL,pL]
  = mkHtml "Buttons" [ H1 [] "Buttons!!", mkSTable [[rP,pP],[rL,pL]] ] hSt

Fig. 6. Several examples of Buttons.

The next form elements are the ‘twin’ definitions for check boxes and radio buttons. Both only offer a choice between being checked or not. Radio buttons usually are grouped together to provide the application user with a single choice out of a limited collection of alternatives. Check boxes usually do the same, but allow several or no choices.

:: CheckBox = CBChecked String | CBNotChecked String
:: RadioButton = RBChecked String | RBNotChecked String

Another way of providing the application user with a single choice from a limited collection is to use a pull down menu. A pull down menu defined by (PullDown (nrVisible,width) (index,items)) displays nrVisible elements, has width width, and a collection of items out of which element index is selected.

:: PullDownMenu = PullDown (Int,Int) (Int,[String])

Finally, for completeness, there exist text input boxes, defined by:

:: TextInput = TI Int Int | TR Int Real | TS Int String

The first argument of these data constructor set the width of the elements, the second the initial value.
We have now discussed all elements of direct form programming. The applications that we can construct now may consist of several forms that are really editors of arbitrary values. These forms, and their values, can already depend on each other in arbitrarily complex ways. The layout of the page is done by direct HTML programming.

As a final example, we extend the *n* fib table example given earlier in this section with a check for illegal input. We do not change the *updateForms* function, but only the *updatePage* function:

```haskell
updatePage :: (Form Int,Form [(String,String,Int)]) *HSt — (Html,*HSt)
updatePage (rangeF,nfibF) hSt
| rangeF.value < 0 || rangeF.value > 40
  || (backF,hSt) = mkEditForm (nFormId "button") (LButton 80 "Back") hSt
  = mkHtml "Wrong Input"
    [ H1 [Hnum_Align Aln_Center] "NFib numbers"
      , Txt "I am truly sorry. This input is beyond my capacity."
      , Br
      , BodyTag backF.form
    ] hSt
| otherwise
  = /* as previously */
```

We first test for the correctness of the input value with the guard

```haskell
| rangeF.value < 0 || rangeF.value > 40
```

If this is not the case, we proceed as previously. If the input value is illegal, then we create a different page in which the application user is politely informed about the incorrect input:

```haskell
| (backF,hSt) = mkEditForm (nFormId "button") (LButton 80 "Back") hSt
  = mkHtml "Wrong Input"
    [ H1 [Hnum_Align Aln_Center] "NFib numbers"
      , Txt "I am truly sorry. This input is beyond my capacity."
      , Br
      , BodyTag backF.form
    ] hSt
```

In order to allow the user to go back and give it another try, a local button is created, especially for this page. This is possible because the *updatePage* function has access to the *HSt* environment. Fig. 7 shows the result after the user has entered incorrect data.

It is sometimes convenient to be able to introduce a form locally within a BodyTag context. In such a context, one does not have access to a *HSt* environment. For this purpose, the function *toHtmlForm* can be used:

```haskell
toHtmlForm :: (*HSt — (Form d,*HSt)) — [BodyTag] | gHTM L{]*|} d
```

Using this function, we could have defined the above exceptional case to the same effect in the following way:

```haskell
updatePage (rangeF,nfibF) hSt
```
Fig. 7. Handling different pages within one application.

```haskell
| rangeF.value < 0 || rangeF.value > 40
  = mkHtml "Wrong Input"
  [ H1 [Hnum.Align Aln.Center] "NFib numbers"
    , Txt "I am truly sorry. This input is beyond my capacity."
    , Er
    , BodyTag backF.form
  ] hSt
  with backF = toHtmlForm (mkEditForm (nFormId "button") (LButton 80 "Back"))
| otherwise
  = /* as previously */
```

**Exercises**

4. *Correct input for the NFib table*

Above we have discussed how the application can check for illegal input in the case of the `nfib` table. Implement a version using a *pull down menu* in which the user can only choose between legal values. Legal values are element of \{0..40\}.

---

**4.5 Programming Model-View Forms**

In the previous section we have introduced the `mkEditForm` function that creates a form with which users can edit values for some data domain. Although this is a powerful abstraction mechanism, it has two shortcomings:

1. The form allows users to change values into arbitrary other values of the data domain. This means that the full range of inhabitants of the domain can be entered by the user. In many cases, forms impose restrictions on the set of admissible values. These restrictions are expressed in a natural way by
means of functions, and their expressiveness goes well beyond the capacity of the type system.

2. The form is derived generically from a data domain, and one of its values. This implies that the presentation and the functionality of the form are strongly coupled. Such a strict coupling of concerns leads to software in which one can not change either the presentation or functionality of a form without having to change the other as well with the same effort.

Based on earlier work, we know that both aspects can be dealt with by means of abstraction [1]. With abstraction, the application works with forms that are modelled by means of values of type m, but that are visualized by means of values of type v. This is a variant of the well-known model-(controller-)view paradigm [14]. What is special about it in this context, is that views are also defined by means of a data model, and hence can be handled generically in exactly the same way as other data models. This is a powerful concept, and we have used it successfully in the past. It turns out that it can be integrated smoothly in the iData Toolkit.

The relation between a model domain m and its view domain v is given by the following collection of functions (FormBimap m v):

\[
\text{FormBimap } m \quad v \\
= \{ \text{toForm} : m \rightarrow \text{Maybe } v \rightarrow v, \text{updForm} : \text{Changed } \rightarrow v \rightarrow v, \text{fromForm} : \text{Changed } \rightarrow v \rightarrow m, \text{resetForm} : \text{Maybe } (v \rightarrow v) \}
\]

Model domain values are transformed to view domain values with toForm. It can use the previous view domain value if necessary. The local behavior of the form that corresponds with the view data model is given by updForm. The Changed parameter indicates whether the value of this form was edited by the user. This record has the same role and value in the function fromForm which transforms the view domain value back to the model domain value. Finally, resetForm is an optional separate normalization after the local behavior function updForm has been applied.

Abstraction is incorporated in the iData Toolkit by a more general function than mkEditForm, viz. mkViewForm. Its type is:

\[
\text{mkViewForm} :: \text{FormId } m \ (\text{FormBimap } m \ v) \ \ast \text{HSt} \rightarrow (\text{Form } m, \ast \text{HSt}) \ | \ \ast \text{HTML[^]} \ | \ v
\]

Note that its signature is almost identical to that of mkEditForm. It has an additional argument of type (FormBimap m v), and it assumes that all the generic machinery is available for the view type v instead of the model type m.

The function mkEditForm is a special case of mkViewForm. It is defined as follows:

\[
\text{mkEditForm formId data hSt}
\]
= mkViewForm formId data
  { toForm = toFormid
    , updForm = case formId.mode of
        Edit  = \_ v -> v
        Display = \_ data
    , fromForm = \_ v -> v
    , resetForm = Nothing
  } hSt

toFormid :: d (Maybe d) -> d

toFormid m Nothing = m

toFormid m (Just v) = v

(toFormid is a useful function that always takes the previous view value, unless there was none.)

Let's construct a slightly more elaborate model-view form. This form should have a simple integer model but a view in which the user can edit integer values by means of a text box or a spin button. We have :: Model := Int. We first design the View type. The view consists of an integer edit box and a spin button. The integer edit box is modelled by the Int type. For the spin button we use two labelled buttons, hence two Button types suffice. Therefore the view type is :: View := (Int,Button,Button). Next, we define the relationship between the model type and the view type, which is expressed as a value of type (FormBimap Model View). We define the four functions:

toForm :: Model (Maybe View) -> View

This function transforms the model into the view. The integer component is simply copied. The two buttons, down and up, are labelled with "-" and "+" respectively. This amounts to:

toForm = \n -> toFormid (n,down,up)
down = LButton (defpixel/6) "-"
up = LButton (defpixel/6) "+

(defpixel is globally used in the iData Toolkit to serve as the default width of elements.)

updForm :: Changed View -> View

This function defines the local behavior of the form. Edit operations on the integer edit box are always legal, due the type safeness of the iData Toolkit. Edit operations on the down (up) button can only be the value Pressed. In case of that operation, the integer value is decreased (increased). We have:

updForm = \_ view -> case view of
          (n,Pressed,_) -> (n-1,down,up)
          (n,_,Pressed) -> (n+1,down,up)
          int_edited -> int_edited

fromForm :: Changed View -> Model

This function transforms the view back to the model. In this case, the integer component is returned.
Finally, this function allows the programmer to reset the view after the new model value has been returned. In this case, this is not necessary, hence Nothing can be returned.

resetForm = Nothing

For completeness, we show the full implementation here:

counterForm :: FormId Int *HSt → (Form In t,*HSt)
counterForm name i hSt = mkViewForm name i counterView hSt
where counterView = { toForm = λ n → toFormid (n,down,up)
    , updForm = λ view → case view of
        (n,Pressed,_) = (n-1,down,up)
        (n,_,Pressed) = (n+1,down,up)
        int_edited = int_edited
    , fromForm = λ (n,_,_) → n
    , resetForm = Nothing
}
where down = LButton (defpixel / 6) "-"
    up = LButton (defpixel / 6) "+

We can now use this model-view form of type (Form Int) and mix it with other forms of that type. Consider an application that takes a number of forms of this type, and presents them below each other, along with a display of the sum of their values. Fig. 8 shows what this application looks like. The updatePage function for such an application is rather straightforward:

updatePage :: [Form In t]*HSt → (Html,*HSt)
updatePage intFs hSt
  = mkHtml "Integer Forms"
    ([ H1 [] "Integer Forms" ] ++ bodies ++ [ toHtml (sum values)]) hSt
where
  (bodies,values) = unzip [(BodyTag form,value) \ {form,value}^intFs]

This function generates the proper page, regardless of the actual content of the list of integer forms. For the screen shot in Fig. 8 we have used the following updateForms function:

updateForms :: *HSt → ([Form Int],*HSt)
updateForms hSt
  = ([intF1,hSt] = mkEditForm (nFormId "simple_int") 1 hSt
  $ [intF2,hSt] = counterForm' (nFormId "counter_int") 2 hSt
  = ([intF1,intF2],hSt)

It should be clear that the order of integer forms as well as their number is quite irrelevant. This example illustrates that model-view forms allow for local behavior, and separation of model and view.
Exercises

5. A boolean model-view form

Create, in an analogous way as done above for the `counterForm`, the following function:

```haskell
boolForm :: FormId Bool *HSt — Form Bool,*HSt
```

This should generate a model-view form with a boolean model type, and as view an unlabelled check box that is checked in case of `true` values (✓), and unchecked in case of `false` values (□).

6. Self correcting forms

In module `htmlFormlib` of the iData Toolkit you can find a number of pre-defined specializations of `mkViewForm`. Two of these functions are `mkSelfForm` and `mkSelf2Form`. Explain the difference between these model-view forms and give an example that illustrates their difference.

7. Storage forms

In module `htmlFormlib` of the iData Toolkit you can find a number of predefined specializations of `mkViewForm`. One of these functions is `mkStoreForm`. Explain what it does and give an example that illustrates its use.

4.6 More Fun With Model-View Forms

In the previous section we have created a number of model-view forms by defining a `Model` and `View` type, and suitable relation functions as `(FormBinmap Model View)`. In this section we show that you can also create new model-view forms on the level of forms themselves.
As a first example, consider a function that creates a list of direct forms from a list of values. It has type:

```
listForm :: FormId [a] *HSt -> (Form [a], *HSt) | gHTML{|*|} a
```

If the value list is empty, no visualization (form field) is required. The value is clearly the empty list, and the user can’t have changed it:

```
listForm _ [] hSt
= ({ changed=False, value=[], form=[], } , hSt)
```

For a non-empty list \([x:xs]\), `listForm` proceeds recursively over `xs`, producing `xsF` and creates the direct form `xF` for `x`. The new form is a rather straightforward composition of these elements. The form is changed if either forms are changed (`changed = xF.changed || xsF.changed`); the value assembles the values in a list (`value = [xF.value; xsF.value]`); the form is the sequential composition of both forms (`form = [BodyTag xF.form; xsF.form]`). The identification values of the intermediate forms are derived from the argument identification value by appending it with their reversed position in the list (nformid). This gives:

```
listForm formid [x:xs] hSt
| (xF, hSt) = mkEditForm nformid x hSt
| (xsF, hSt) = listForm formid xs hSt

j
= ({ changed = xF.changed || xsF.changed
, value = [xF.value; xsF.value]
, form = [BodyTag xF.form; xsF.form]
}, hSt)
```

where \(nformid = \{formid & id = formid.id<$length xs\}\)

A closer inspection at this function shows that it has room for generalization:

- the sequential combination of the forms can be generalized to any specific layout;
- there is no real need to create direct forms, any form creation function should do.

Based on these observations, we create a more general function, `layoutListForm` that is parameterized with a form layout combinator function (of type \([\text{BodyTag}] [\text{BodyTag}] \rightarrow [\text{BodyTag}]\) ), and a form creation function (of type FormId a *HSt -> (Form a, *HSt)). Its definition follows in a trivial way from `listForm` above:

```
layoutListForm :: ([BodyTag] [BodyTag] -> [BodyTag])
                  (FormId a *HSt -> (Form a, *HSt))
                  FormId [a] *HSt -> (Form [a], *HSt) | gHTML{|*|} a

layoutListForm :: ([BodyTag] [BodyTag] -> [BodyTag])
                  (FormId a *HSt -> (Form a, *HSt))
                  FormId [a] *HSt -> (Form [a], *HSt) | gHTML{|*|} a

layoutListForm formid [x:xs] hSt
| (xF, hSt) = layoutListForm layoutF formF formid xs hSt
| (xsF, hSt) = layoutListForm layoutF formF formid xs hSt

j
= ({ changed = xF.changed || xsF.changed
, value = [xF.value; xsF.value]
, form = layoutF xF.form; xsF.form
}, hSt)
```
where nformid = {formid & id = formid.id <$ length xs}

listForm can now be expressed concisely as a special case of this function, as well as a range of other useful functions:

listForm = layoutListForm (λf1 f2 — [BodyTag f1: f2]) mkEditForm
horlistForm = layoutListForm (λf1 f2 — [f1 <-> f2]) mkEditForm
vertlistForm = layoutListForm (λf1 f2 — [f1 <||> f2]) mkEditForm
table_hv_Form = layoutListForm (λf1 f2 — [f1 <||> f2]) horlistForm

The layoutListForm function is useful for combining form creation functions which view types are assembled lists. We now discuss a more general combinator function that abstracts also from this aspect. It's signature is:

layoutIndexForm :: ([BodyTag] [BodyTag] — [BodyTag])
(Int Bool FormId x *HSt — (Form y,*HSt))
y (y y — y)
Int Bool FormId |x| *HSt — (Form y,*HSt)

The first two arguments serve the same purposes as with layoutListForm: the first arguments combines the layout of forms, the second argument creates forms. These forms have a view type y, and therefore we need to have a value of type y in case of the empty list of data, as well as a combinator function of type y y → y. These are the third and fourth argument of layoutIndexForm. The integer argument is required for generating fresh identification values from the given identification value. The boolean argument indicates whether the elements are going to be initialized.

The implementation of layoutIndexFor is analogous to layoutListForm. In case of an empty list of data from which forms need to be generated, a form is returned with the given ‘neutral’ value:

layoutIndexForm _ _ r _ _ _ _ [ ] hSt
= ({changed=False , value=r , form=[]},hSt)

In case of a non-empty list [x:xs], layoutIndexForm proceeds recursively over xs producing xsF, and applies the form creation function to x, yielding xF. The new form is assembled from these two forms. Its changed and form values are computed in an identical way as by layoutListForm. Its value is computed by the value combinator function combineF.

With this general function we can assemble a form which view is defined by a list of buttons, and which model is an associated callback function. This function has signature:
ListFuncBut :: (Bool FormId [(Button, a -> a)] *HSt -> (Form (a -> a),*HSt))

The most important argument is the third one: this argument associates callback functions with buttons. The intention is that for a list of button-callback function pairs \([(b_0, f_0), \ldots, (b_n, f_n)]\) a form is created that has value \(f_i\) whenever the application user has pressed button \(b_i\) and the identity function otherwise. This function can be implemented using the general layoutIndexForm given above:

\[
\text{ListFuncBut} = \text{layoutIndexForm} (\lambda f2 \rightarrow [\text{BodyTag } f1: f2]) \text{ FuncBut id (o) 0}
\]

The lower level function FuncBut creates a (Form (a -> a)) with a Button view. It uses the boolean and the integer to generate a fresh identification value for that element. Function composition is used to combine the callback functions from all button elements.

In a similar way, we can define a form that displays table of buttons, and that returns the callback function of the associated button that has been pressed:

TableFuncBut :: (FormId [(Button, a -> a)] *HSt -> (Form (a -> a),*HSt))

\[
\text{TableFuncBut} = \text{layoutIndexForm} (\lambda f1 f2 \rightarrow [f1 <||> f2]) \text{ FuncBut id (o) 0 False [BodyTag f1: f2])}
\]

We conclude this section with an example that creates a simple integer based calculator (see Fig. 9). The calculator uses a number of buttons to enter integer values and do basic arithmetic. Clearly, we intend to use the TableFuncBut that we have constructed above to display buttons, and obtain callback functions. The callback functions have type \(\text{CalcSt} \rightarrow \text{CalcSt}\) with :: CalcSt := (Int,Int).

We arrange the buttons as:

\[
\text{buttons} = [\begin{array}{c}
\text{btn "7" (set 7)}, \text{btn "8" (set 8)}, \text{btn "9" (set 9)} \\
\text{btn "4" (set 4)}, \text{btn "5" (set 5)}, \text{btn "6" (set 6)} \\
\text{btn "1" (set 1)}, \text{btn "2" (set 2)}, \text{btn "3" (set 3)} \\
\text{btn "0" (set 0)}, \text{btn "C" clear} \\
\text{btn "+" (app (+))}, \text{btn "-" (app (-))}, \text{btn "+" (app (+))}
\end{array}]
\]

where set \(i\ (t,b) = (t, b \times 10 + i)\)

\[
\text{clear} \ (t,b) = (t, 0)
\]

\[
\text{app fun} \ (t,b) = (\text{fun t b, 0})
\]

\[
\text{btn lbl cbf} = (\text{LButton defpix1 / 3) lbl, cbf})
\]

The calculator consists of two forms: one that displays the current value and entered value (displayF) and one that shows the buttons of the calculator (buttonsF). These forms are, as usual, created by the updateForms function:

updateForms :: *HSt -> ((Form (CalcSt -> CalcSt),Form CalcSt),*HSt)

\[
\text{updateForms hSt} = (\text{buttonsF,hSt}) = \text{TableFuncBut (nFormId "calcbut") buttons hSt}
\]

\[
\text{updateForms hSt} = (\text{displayF,hSt}) = \text{mkStoreForm (ndFormId "display") (0,0) buttonsF.value hSt}
\]

= ((buttonsF,displayF),hSt)
With these forms the definition of the page is easily constructed:

```haskell
updatePage :: (Form (CalcSt -> CalcSt), Form CalcSt) *HSt -> (Html,*HSt)
updatePage (buttonsF,displayF) hSt
  = mkHtml "Calculator" [ H1 [], "Calculator"
  , toHtml t <.||.> toHtml b
  ; buttonsF.form
  ] hSt
  where (t,b) = displayF.value
```

4.7 Login Form

We conclude this survey of the iData Toolkit with a larger example (50 loc) that implements a frequently occurring component of web applications, viz. a login form. With such a form applications protect themselves from access by unregistered users. A screen shot of the login page that we develop is given in Fig. 10.

Logins are kept in a record, Login, in which the login name and password are stored. Both a generic and overloaded equality operator on Login values are defined.

```haskell
:: Login = { loginName::String, password::String }
derive gForm Login; derive gUpd Login; derive gPrint Login; derive gParse Login
derive gEq Login
instance == Login where (=) login1 login2 = login1 === login2
```

```haskell
mkLogin :: String String -> Login
mkLogin name pwd = { loginName=name, password=pwd }
```
By now, the standard overhead of an iData Toolkit program should be familiar:

```
module loginAdmin

import StdEnv, StdHtml, GenEq

Start world = doHtml MyPage world

The function MyPage is the function that does the 'real' work. The application
basically switches between two pages: a login page in which a name and password
need to be entered, and a member page that should be reached only if a valid
member has logged in. Because this exercise is not about the actual member
page, we keep it rather minimal, and only display a welcome message:

memberPage :: (Form Login) → (*HSt → ([BodyTag],*HSt))
memberPage loginF = return [Txt ("Welcome "$ loginF.value.loginName)]

The login page uses a login store to keep track of all valid username/password
combinations. For this purpose, a persistent form of type (Form [Login]) is useful.
This form is identified by the tag "loginDB". In order to ensure that exactly this
form is used throughout the application, it is a good discipline to use a single
function that associates a form with its tag:

loginStore :: ([Login] → [Login]) *HSt → (Form [Login],*HSt)
loginStore = mkStoreForm (pFormId "loginDB") []

The application first needs to determine in what stage of its session it actually
is. This depends on the current content of the login form and the database. If the
user has entered valid data, the member page should be presented, and otherwise
the login page should be presented:

MyPage :: *HSt → (Html,*HSt)
MyPage hSt
  & (loginF, hSt) = mkEditForm (sFormId "login") (mkLogin "" "") hSt
The login page allows the user to add his username and password to the database. For this purpose an additional button form is created, making use of the callback scheme that is offered by the ListFuncBut function that was discussed in Sect. 4.6. Of course, if no information was entered (both Login fields are "") then this should not be possible. In that case, the button is in display mode:

\[
\text{addLoginButton} :: \text{Login} \times \text{HSt} \rightarrow (\text{Form (Bool \rightarrow Bool)}, \text{HSt})
\]

\[
\text{addLoginButton} \; \text{value} = \text{ListFuncBut} \; \text{False} \; (\text{formid "addLogin"}) \; \text{pagebuttons}
\]

\[
\text{where}
\]

\[
\text{formid} = \text{if } (\text{value} \neq \text{mkLogin "" "") nFormId ndFormId
\]

If the user has pushed this button (the .changed field is true), then his username/password combination should be added to the persistent database, and the member page should be displayed. If the user did not push the button, then the login page should be displayed again.

\[
\text{loginPage} :: (\text{Form Login}) \times \text{HSt} \rightarrow ([\text{BodyTag}], \text{HSt})
\]

\[
\text{loginPage} \; \text{loginF} \; \text{hSt}
\]

\[
\text{where}
\]

\[
\text{addLogin} :: \text{Login Bool} \rightarrow \text{Login}
\]

\[
\text{addLogin} \; \text{newname} \; \text{added} \; \text{loginDB}
\]

\[
\text{if } \text{added} \&\& \text{newname} \neq \text{mkLogin "" "") \&\& \text{not (isMember newname loginDB)}
\]

Lifting low-level Web programming has triggered a lot of research. Many authors have worked on turning the generation and manipulation of HTML (XML) pages
into a typed discipline. Early work is by Wallace and Runciman [23] on XML transformers in Haskell. The Haskell CGI library by Meijer [15] frees the programmer from dealing with CGI printing and parsing. Hanus uses similar types [10] in Curry. Thiemann constructs typed encodings of HTML in extended Haskell in an increasing level of precision for valid documents [21, 22]. XML transforming programs with GenericHVskell has been investigated in UUXML [3]. Elsman and Larsen [8] have worked on typed representations of XML in ML [16]. Our use of ADTs can be placed between the single, generic type used by Meijer and Hanus, and the collection of types used by Thiemann. It allows the HTML definition to be done completely with separate data types for separate HTML elements.

iData components are form abstractions. A pioneer project to experiment with form-based services is Mawl [4]. It has been improved upon by means of Powerforms [6], used in the <bigwig> project [7]. These projects provide templates which, roughly speaking, are HTML pages with holes in which scalar data as well as lists can be plugged in (Mawl), but also other templates (<bigwig>). They advocate compile-time systems, because this allows one to use type systems and other static analysis. Powerforms reside on the client-side of a web application. The type system is used to filter out illegal user input. The use of the type system is what they have in common with our approach. Because iData are encoded by ADTs, we get higher-order forms/pages for free.

Web applications can be structured with continuations. This has been done by Hughes, with his arrow framework [13]. Queinnec states that “A browser is a device that can invoke continuations multiply/simultaneously” [20]. Graunke et al [9] have explored continuations as (one of three) functional compilation technique(s) to transform sequential interactive programs to CGI programs. Our approach is simpler because for every page we have a complete (set of) model value(s) that can be stored and retrieved generically in a page. An application is resurrected simply by recovering its previous state.

6 Conclusions

In these lecture notes we have described the iData Toolkit. With this toolkit, the programmer can create dynamic web applications that use interconnected forms. Programming these applications can be very hard due to the complex interactions between these forms, and the form programming itself. We have shown how a functional style approach can help reduce the complexity of this problem. The following key ideas have been crucial:

- A web application should be a single function.
- A form should be a type-directed editor.
- Forms should be regarded as objects.
- Web interfaces should be generated from typed specifications.
- There should be a strict separation between model and view.

The result is a toolkit that gives the programmer the freedom to shape the data structures that he really needs for his problem domain, instead of being forced to
squeeze his problem domain in terms of API predetermined data structures. This essentially relies on the generative power of generic programming. Although the implementation of the iData Toolkit relies on generic programming, this is not necessary for the application programmer. We have spent a lot of effort to keep the API of the iData Toolkit as simple as possible.

We hope you have enjoyed this tutorial and the exercises.

Acknowledgements

Javier Pomer Tendillo visited our department during his Erasmus project. He has helped in setting up the iData Toolkit, and find out the nitty-gritty details of HTML programming.

References


