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 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. By this we mean that programmers can develop data structures and functions in the usual way. From the data structures the iData Toolkit is able to generate forms that can be used in a web application. It does this by making use of advanced programming concepts such as generic programming. The programmer need not be aware of this.

1 Introduction

The World Wide Web has become an important infrastructure for institutions such as universities, government, 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 show a novel approach to program web applications that consist of an arbitrary number of forms with arbitrarily complex relations. As a typical example, we have constructed a *web shop* application for selling CDs. 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; *fill shopping cart* button, and *overview* of the most recently added item. In addition, it shows that there are also elements that are not related with forms, but rather with layout and make up. We consider these elements to be purely functionally dependent on 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 iData Toolkit provides web application programmers with a means to express themselves on a high level of abstraction, without compromising the complexity of the applications. In contrast with the internal realization, the API of the iData Toolkit contains no advanced functional programming concepts: we deliberately

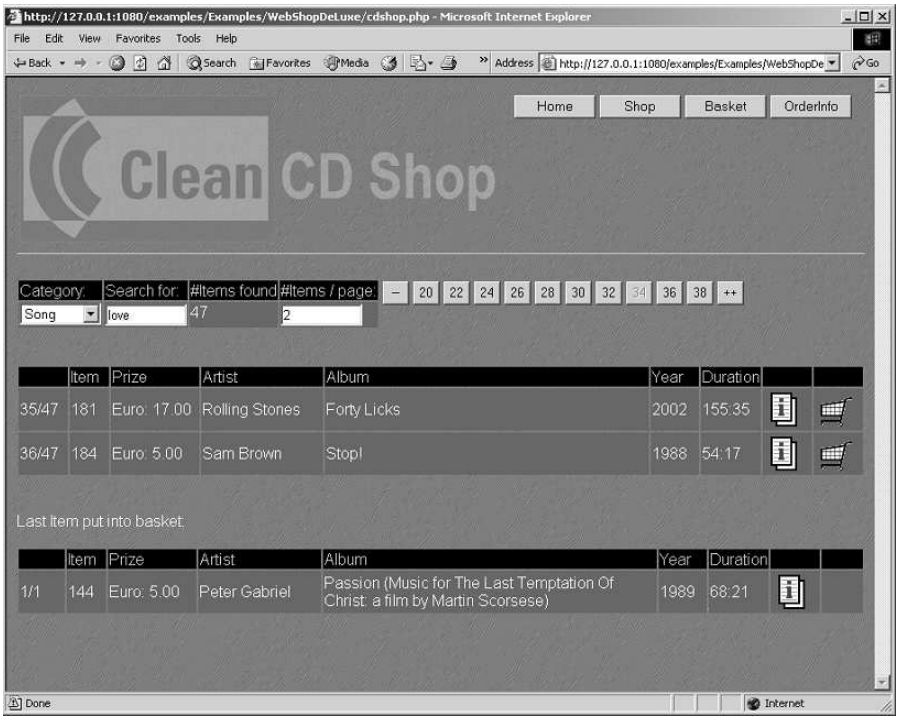


Fig. 1. A Web shop application, programmed with the iData Toolkit.

have kept it as simple as possible. The most important advanced functional programming technique that is used to create the toolkit is *generic programming* [12, 13]. 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 of the toolkit.

We use the functional programming language Clean [19, 20], 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. The Clean compiler 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 [8]. 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. 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*. 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, which is interpreted and displayed by the browser.

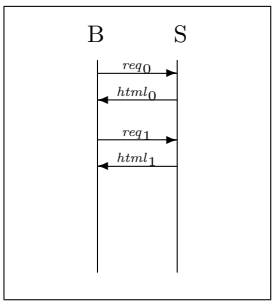


Fig. 2. 'Classic' W^3.

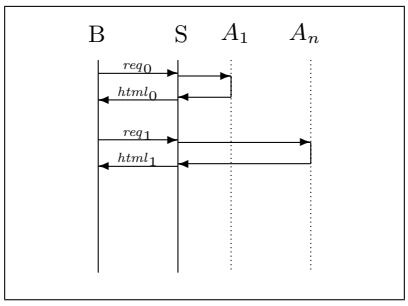


Fig. 3. 'Contemporary' W^3.

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 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 A_1 upto A_n. The dashed lines indicate that these applications are *executed on request*, and *terminate* after having produced the desired result. To the server, it is irrelevant in what system or programming language these applications are written. They can be generated by a wide variety of tools and techniques, such as general purpose programming languages, 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 behavior 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 built-in 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, use 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 GUIs. 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 a built-in state concept, this can not be done with web applications. 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. $A = \{A_1 \dots A_n\}$. Our view of the structure of a web application is depicted in Fig. 4. In our opinion, 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. This is 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 `*World` → `*World`. Here, the type constructor `World` acts as an explicit parameter that represents the external environment of the Clean program. The *uniqueness attribute* `*` in front of it tells us that this environment can not be duplicated or shared: it is passed along in a single-threaded manner. This standard Clean

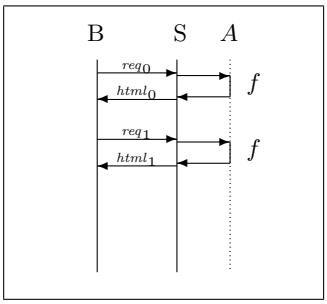


Fig. 4. The iData Toolkit W^3.

style of passing along the environment explicitly is known as *explicit multiple environment passing*, and also as the *world-as-value* paradigm. 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 `HSt`, 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:

```
*HSt → (Html,*HSt)
```

This type is connected with the type of an interactive Clean program with the following library wrapper function:

```
doHtml :: (*HSt → (Html,*HSt)) → *World → *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.

We 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*. The model value is of concern to the application programmer, the view value is of concern to the application user. They are both specified by means of custom data types. Their relative mapping is of course specified using functions between these data domains. These have to be provided by the application programmer. The iData Toolkit has one pivot

function that implements a complete form for any given model value of type `m`, view value of type `v`, and model-view relationship of type (`HBimap m v`):

```
mkViewForm :: FormId m (HBimap m v) *HSt → (Form m,*HSt) | gHtml{|*|} 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:

```
class gHtml t | 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 `mkViewForm` will be explained. Here we suffice with a rather informal specification based on its type signature: `FormId` identifies the form elements and their main attributes in a web application's page, `m` is the initial model value, and (`HBimap m v`) is the collection of relational functions between model and view. The result is a new form (`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, `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 `mkEditForm`:

```
mkEditForm :: FormId m *HSt → (Form m,*HSt) | gHtml{|*|} m
```

Given the identification of a form `id`, an initial model value `m`, (`mkEditForm 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: The account above demonstrates that we have built a standard framework for building web applications that contain 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 go through the following steps: we start with programming HTML directly in sections 4.1 through 4.3. Once we know how to play with HTML, we can concentrate on programming forms, in 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 standard.

```
:: 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 zero element. Attributes are encoded as `FooAttr` data types.

As an example, the value `hello :: Html` defined by

```
hello = Html (Head [`Hd_Std [Std_Title "Hello World Example"]] [])
             (Body [] [Txt "Hello World!"])
```

corresponds with the following HTML code:

```
<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:

```
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 [`Hd_Std [Std_Title s]] []
      body tags   = Body [] tags
```

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

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

The `Html` value is transformed to HTML code and written to file. This is one of the tasks of `doHtml`. The complete code for this example then is:

```
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!"]
```

The collection of data types shows that HTML can be encoded straightforwardly into a set of ADTs. There are some minor complications. In Clean, as well as in Haskell [18], 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 function `gHpr` that implements the naming conventions that have been discussed above, and prints the correct HTML code. Its definition is not relevant here.

Our approach has the following advantages:

– One obtains a grammar for HTML which is convenient for the programmer.
– The type system eliminates type and typing errors that can occur when working in plain HTML.
– We can define a type driven generic function for generating HTML code.
– Future changes of HTML are likely to change the ADT only.
– Because the HTML is embedded in our programming language as a set of ADTs, we can use the language to create complex HTML.

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 *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:

```
toHtml :: a → BodyTag | gForm{|*|} a
```

It uses the generic function `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 `toHtml`.

Let's start with simple stuff, and move towards more complicated types. Table 5 shows what the generated HTML looks like in a web browser for the basic types `Bool`, `Int`, `Real`, and `String`. These are obtained by replacing ***expr*** in:

```
Start world = doHtml (mkHtml "Example" [toHtml expr]) world
```

Next we proceed with types that have been composed by the standard type constructors of Clean. Filling in ***expr***= (`"Nfib "<$40<$" = ",Nfib` 40), which has type (`String,Int`) yields the following HTML output:

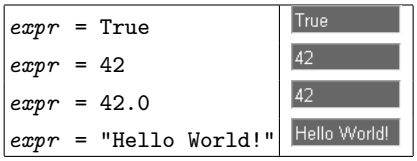


Fig. 5. HTML generated by `toHtml` for several basic types.

The operator `<$` is a useful abbreviation for the frequently occurring case of concatening a `String` value with something that can be turned into a `String`:

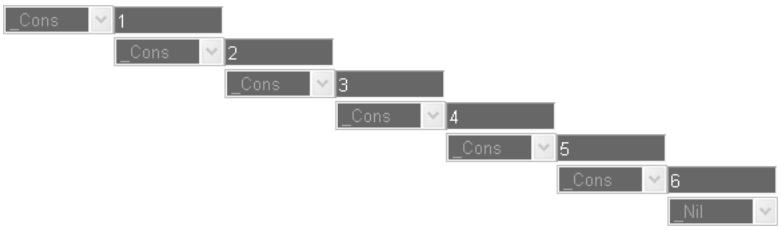
```
(<$) infixl :: !String !a → String | toString a
(<$) str x = str +++ toString x
```

The *strictness annotation* `!` just in front of the type constructors states that `<$` is strict in both arguments.

Within the Clean compiler, the internal representation of lists is

```
:: [a] = _Cons a [a] | _Nil
```

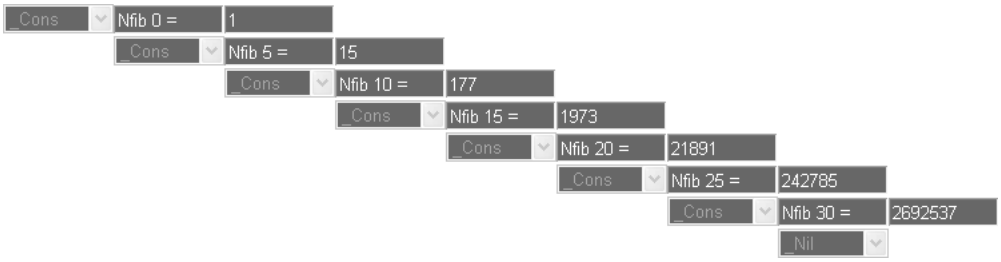
This algebraic structure is clearly visible in case of ***expr***= `[1..6]`, which has type `[Int]`:



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:

```
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 `[(String,Int)]`. An example value is ***expr***= `[("Nfib "<$n<$" = ",Nfib n) \\ n ← [0,5..30]]`:



In Clean, 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:

```
:: Person = { name::String, address::String, city::String }
derive gForm Person

peter = {name="Achten",     address="Abersland",   city="Wijchen"}
rinus = {name="Plasmeijer",address="Knollenberg",city="Mook"}
```

Then the value ***expr***= `peter` is displayed as:

Exercises

1. *Lists, differently*

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

expr= `[("Nfib "<$n<$" = ",Nfib n) \\ n ← [0,5..30]]`.

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

`[toHtml ("Nfib "<$n<$" = ",Nfib n) \\ n ← [0,5..30]]`?

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 data type ***T*** as long as you 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:

```
Start world = 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:

```
(<.=.>) infixl 5 :: BodyTag BodyTag → BodyTag
(<.=.>) b1 b2 = STable [ Tbl_CellPadding (Pixels 0)
                       , Tbl_CellSpacing (Pixels 0)
                       ] [[b1,b2]]

(<=>) infixl 5 :: [BodyTag] [BodyTag] → BodyTag
(<=>) b1 b2 = (BodyTag b1) <.=.> (BodyTag b2)
```

With these combinators, we can easily place two values next to each other. Consider for instance *expr*= `toHtml peter <.=.> toHtml rinus`. This yields the following HTML:

Name:	Achten	Name:	Plasmeijer
Address:	Abersland	Address:	Knollenberg
City:	Wijchen	City:	Mook

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 `map toHtml`), and then replacing every *cons* by <.=.> (by folding <.=.> over the resulting list). This is done with the iData Toolkit function `mkRowForm`, defined concisely as:

```
mkRowForm :: [BodyTag] → BodyTag
mkRowForm xs = foldr (<.=.>) EmptyBody xs
```

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

1	2	3	4	5	6	7

In exactly analogous ways, we can do this for *vertical* layout, and introduce:

```
(<.||.>) infixl 4 :: BodyTag BodyTag → BodyTag
(<.||.>) b1 b2 = STable [ Tbl_CellPadding (Pixels 0)
                        , Tbl_CellSpacing (Pixels 0)
                        ] [[b1],[b2]]

(<||>) infixl 4 :: [BodyTag] [BodyTag] → BodyTag
(<||>) b1 b2 = (BodyTag b1) <.||.> (BodyTag b2)

mkColForm :: [BodyTag] → BodyTag
mkColForm xs = foldr (<.||.>) EmptyBody xs
```

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

```
expr= mkColForm (map toHtml [("Nfib "<$n," = ",Nfib n) \\ n ← [0..10]]).
```

This yields:

Nfib 0	=	1
Nfib 1	=	1
Nfib 2	=	3
Nfib 3	=	5
Nfib 4	=	9
Nfib 5	=	15
Nfib 6	=	25
Nfib 7	=	41
Nfib 8	=	67
Nfib 9	=	109
Nfib 10	=	177

Finally, elements can be arranged in a table, using the function `mkSTable`:

```
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]          \\ row ← 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).

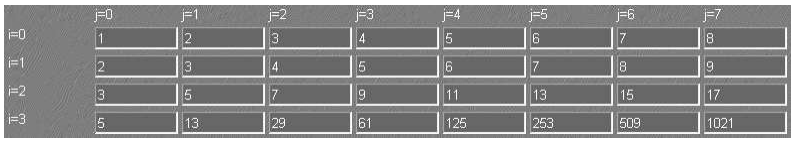
```
augmentTable :: [BodyTag] [BodyTag] [[BodyTag]] → [[BodyTag]]
```

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

```
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:



	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1	2	3	4	5	6	7	8
i=1	2	3	4	5	6	7	8	9
i=2	3	5	7	9	11	13	15	17
i=3	5	13	29	61	125	253	509	1021

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*. Therefor, 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`:

```
mkEditForm :: FormId d *HSt → (Form d,*HSt) | gHTML{|*|} 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`. This is a generic function of signature:

```
generic gForm d :: FormId d *HSt → (Form d,*HSt)
```

(`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:

```
:: FormId   = { id       :: String
              , lifespan :: Lifespan
              , mode     :: Mode   }
```

```
:: Lifespan = Persistent | Session | Page
:: Mode     = Display    | Edit
```

The first mandatory attribute is an *identification tag* (`id::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* (`lifespan::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* (`mode::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:

```
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
```

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

```
:: Form d = { changed :: Bool
            , value   :: d
            , form    :: [BodyTag] }
```

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`:

```
toHtml :: a → BodyTag | gForm{|*|} a
toHtml a
   # ({form},_) = gForm{|*|} {id="__toHtml",lifespan=Page,mode=Display} a ...
   = BodyTag form
```

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 (`Form Person`) (`Person` and `rinus` were defined at the end of Sect. 4.2). For completeness, we give the full code:

module `fragments`

```
import StdEnv
import StdHtml

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

:: Person = // Person definition here
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 `mkEditForm` relies on a collection of generic functions, we also need to derive instances for these functions for `Person`. This should become standard idiom when defining new types for forms.

Because we have created an *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:

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 `*HSt` → (`Html`,`*HSt`), `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.

```
myPage :: *HSt → (Html,*HSt)
```

```
myPage hSt
  ♯ (forms,hSt) = updateForms hSt
  = updatePage forms hSt
```

In the remainder of this sequel, we adopt this scheme and modify the `updateForms` and `updatePage` functions. In the example, `updateForms` is simply

$$(\texttt{mkEditForm (nFormId "person") rinus})$$

and `updatePage` is

$$(\lambda\texttt{person hSt} \rightarrow \texttt{mkHtml ... hSt}).$$

Here is an example of a slightly more interactive web application. We extend the *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 *nfib* numbers from 0 to n. For this also a form is used. Let's first construct the two forms:

```
updateForms :: *HSt → ((Form Int,Form [(String,String,Int)]),*HSt)
updateForms hSt
  ♯ (rangeF,hSt) = mkEditForm (nFormId "range") 10 hSt
  ♯ (nfibF, hSt) = mkEditForm (ndFormId "nfib") [  ("Nfib "<$ n," = ",Nfib n)
                                                \\ n←[0..rangeF.value]
                                                ]  hSt
  = ((rangeF,nfibF),hSt)

derive gForm []; derive gUpd []
```

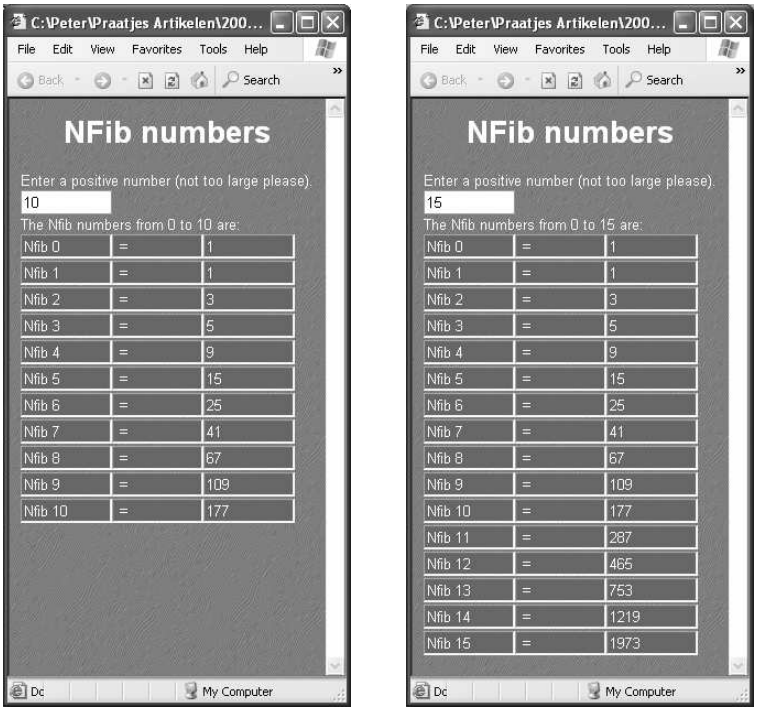
The integer form `rangeF` is straightforward. Its initial value is 10, and it is identified with tag `"range"`. The *nfib* table form `nfibF` is more interesting, as its construction clearly depends on the value of `rangeF`. It is identified by tag `"nfib"`, and is not editable.

Given these two forms, `updatePage` needs to create the proper web page:

```
updatePage :: (Form Int,Form [(String,String,Int)]) *HSt → (Html,*HSt)
updatePage (rangeF,nfibF) hSt
  = mkHtml "NFib" [ H1 [Hnum_Align Aln_Center] "NFib numbers"
                  , Txt "Enter a positive number (not too large please)."
                  , BodyTag rangeF.form
                  , Br
                  , Txt ("The Nfib numbers from 0 to "<$rangeF.value<$" are:")
                  , Br
                  , mkColForm (map toHtml nfibF.value)
                  ] hSt
```

The interesting bits are the fact that from `rangeF` we use both its `form` and `value`, and from the `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 (`LButton width label`), or it uses an image file located at `path` and with dimensions `dim` and is defined by (`PButton dim path`). When the user presses the button, it enters the `Pressed` state. After processing by the iData Toolkit, it returns to its previous state. This gives the following, compact definition of a button:

```
:: Button = Pressed | LButton Int String | 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):

```
updateForms :: *HSt → ([Form Button],*HSt)
updateForms hSt
    # (rP,hSt)  = mkEditForm (nFormId "rinusB") (PButton dim "rinus.jpg")   hSt
    # (pP,hSt)  = mkEditForm (nFormId "peterB") (PButton dim "peter.jpg")   hSt
    # (rL,hSt)  = mkEditForm (nFormId "rinusL") (LButton (snd dim) "Rinus") hSt
    # (pL,hSt)  = mkEditForm (nFormId "peterL") (LButton (snd dim) "Peter") hSt
    = ([rP,pP,rL,pL],hSt)
```

```
where dim        = (60,80)

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

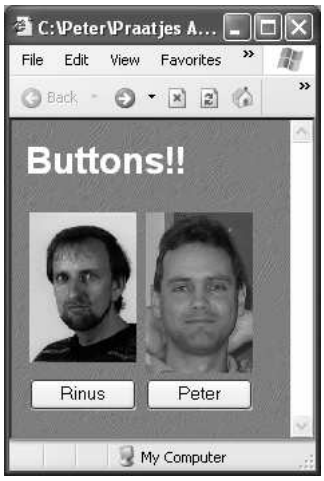


Fig. 6. Several examples of `Button`s.

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 *nfib* 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:

```
updatePage :: (Form Int,Form [(String,String,Int)]) *HSt → (Html,*HSt)          1.
updatePage (rangeF,nfibF) hSt                                                    2.
| rangeF.value < 0 || rangeF.value > 40                                          3.
    # (backF,hSt) = mkEditForm (nFormId "button") (LButton 80 "Back") hSt        4.
    = mkHtml "Wrong Input"                                                       5.
             [ H1 [Hnum_Align Aln_Center] "NFib numbers"                         6.
             , Txt "I am truly sorry. This input is beyond my capacity."         7.
             , Br                                                                8.
             , BodyTag backF.form                                                9.
             ] hSt                                                               10.
| otherwise                                                                      11.
    = /* as previously */
```

The guard at line 3 tests for incorrect input values. 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. This is done in lines 4–10. 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.

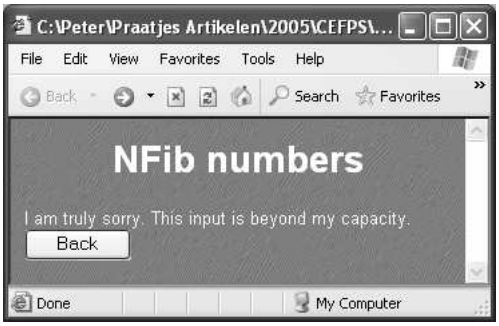


Fig. 7. Handling different pages within one application.

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:

```
toHtmlForm :: (*HSt → (Form d,*HSt)) → [BodyTag] | gHTML{|*|} d
```

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

```
updatePage (rangeF,nfibF) hSt
| 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."
             , Br
             , 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 [15]. 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 for desktop GUI programming. 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`):

```
:: FormBimap m v
   = { toForm    :: m → Maybe v → v
     , updForm   :: Changed → v → v
     , fromForm  :: Changed → v → m
     , resetForm :: Maybe (v → v)  }
:: Changed
   = { isChanged :: Bool
     , changedId :: String          }
```

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:

```
mkViewForm :: FormId m (FormBimap m v) *HSt → (Form m,*HSt) | gHTML{|*|} 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:

```
mkEditForm formId data hSt
  = mkViewForm formId data
       { toForm   = toFormid
       , updForm  = case formId.mode of
                      Edit    = λ_ v → v
                      Display = λ_ _ → data
       , fromForm = λ_ 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 = λn → 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 = λ_ 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:

```
fromForm = λ_ (n,_,_) = n
```

resetForm :: Maybe (View → View)
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 Int,*HSt)
counterForm name i hSt = mkViewForm name i counterView hSt
```

```
where counterView = { toForm    = λn → 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 Int] *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]
```

Fig. 8. Mixing various (`Form Int`)s.

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:

```
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 predefined 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 (`FormBimap 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
   ♯ (xsF,hSt)  = listForm   formid xs hSt
   ♯ (xF, hSt)  = mkEditForm nformid x hSt
   = ({ 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 [BodyTag] [BodyTag] → [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 _ _ _ [] hSt
   = ({changed=False, value=[], form=[]}, hSt)
layoutListForm layoutF formF formid [x:xs] hSt
   ♯ (xsF,hSt) = layoutListForm layoutF formF formid xs hSt
   ♯ (xF, hSt) = formF nformid x hSt
   = ({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 `layoutIndexForm` 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`.

```
layoutIndexForm layoutF formF r combineF n b formid [x:xs] hSt
  # (xsF,hSt) = layoutIndexForm layoutF formF r combineF (n+1) b formid xs hSt
  # (xF, hSt) = formF n b formid x hSt
  = ({changed = xF.changed || xsF.changed
     ,value   = combineF xsF.value xF.value
     ,form    = layoutF xF.form xsF.form
     },hSt)
```

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:

```
ListFuncBut = layoutIndexForm (λf1 f2 → [BodyTag f1:f2]) 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))
TableFuncBut
  = layoutIndexForm
      (λf1 f2 → [f1 <|> f2])
      (layoutIndexForm (λf1 f2 → [BodyTag f1:f2]) FuncBut id (o))
      id (o) 0 False
```

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 as the view, and program with callback functions as the model. The callback functions have type `CalcSt → CalcSt` with `:: CalcSt :== (Int,Int)`.

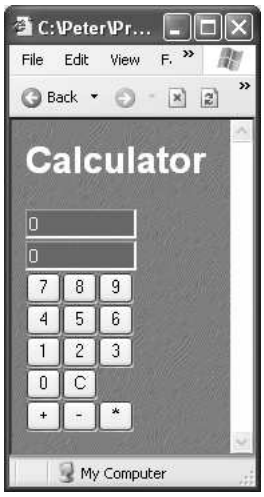


Fig. 9. A simple integer based calculator.

We arrange the `buttons` as:

```
buttons = [ [btn "7" (set 7),   btn "8" (set 8),   btn "9" (set 9)  ]
          , [btn "4" (set 4),   btn "5" (set 5),   btn "6" (set 6)  ]
          , [btn "1" (set 1),   btn "2" (set 2),   btn "3" (set 3)  ]
          , [btn "0" (set 0),   btn "C"  clear                      ]
```

```
          , [btn "+" (app (+)), btn "-" (app (-)), btn "*" (app (*))]
          ]
where set   i (t,b) = (t      , b*10 + i)
      clear   (t,b) = (t      , 0)
      app fun (t,b) = (fun t b, 0)

      btn lbl cbf   = (LButton (defpixel / 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)
updateForms hSt
   # (buttonsF,hSt) = TableFuncBut (nFormId "calcbut") buttons hSt
   # (displayF,hSt) = mkStoreForm (ndFormId "display") (0,0) buttonsF.value hSt
   = ((buttonsF,displayF),hSt)
```

With these forms the definition of the page is easily constructed:

```
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 of type `Login` in which the login name and password are stored. Both a generic and overloaded equality operator on `Login` values are defined.

```
:: 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

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
```

Fig. 10. The initial login page.

```
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
   # (loginDBF,hSt) = loginStore id hSt
   # (page,    hSt) = if (isMember loginF.value loginDBF.value)
                         (memberPage loginF hSt)
                         (loginPage  loginF hSt)
```

```
   = mkHtml "Login" [BodyTag page] 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:

```
addLoginButton :: Login *HSt → (Form (Bool → Bool),*HSt)
addLoginButton value = ListFuncBut False (formid "addlogin") pagebuttons
where pagebuttons    = [ (LButton defpixel "addLogin", const True) ]
      formid         = if (value ≠ 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.

```
loginPage :: (Form Login) *HSt → ([BodyTag],*HSt)
loginPage loginF hSt
   # (addloginF,hSt) = addLoginButton loginF.value hSt
   # (loginDBF, hSt) = loginStore (addLogin loginF.value addloginF.changed) hSt
   | isMember loginF.value loginDBF.value
                     = memberPage loginF hSt
   | otherwise       = ( [ Txt "Please log in ..."
                         , Br, Br
                         , BodyTag loginF.form
                         , Br
                         , BodyTag addloginF.form
                         ], hSt)
where
   addLogin :: Login Bool [Login] → [Login]
   addLogin newname added loginDB
      | added && newname ≠ mkLogin "" "" && not (isMember newname loginDB)
                  = [newname:loginDB]
      | otherwise = loginDB
```

The application that we have created enforces a user to either enter a valid username/password combination or add a new, non-existing, combination. Only in these cases, the user reaches the member page.

5 Related Work

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 [24] on XML transformers in Haskell. The Haskell CGI library by Meijer [16] frees the programmer from dealing with CGI printing and parsing. Hanus uses similar types [11]

in Curry. Thiemann constructs typed encodings of HTML in extended Haskell in an increasing level of precision for *valid* documents [22, 23]. XML transforming programs with GenericH∀skell has been investigated in UUXML [3]. Elsman and Larsen [9] have worked on typed representations of XML in ML [17]. 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 [14]. Queinnec states that "A browser is a device that can invoke continuations multiply/simultaneously" [21]. Graunke *et al* [10] 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 in finding out the nitty-gritty details of HTML programming. The authors like to thank the anonymous referee for improving the presentation of this document.

References

1. P. Achten, M. van Eekelen, and R. Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. In *Practical Aspects of Declarative Programming, PADL04*, volume 3057 of *LNCS*, pages 39–55. Springer, 2004.
2. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. 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, Sept. 2002.
3. F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, June 2004.
4. D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a Domain Specific Language for Form-based Services. In *Usenix Conference on Domain Specific Languages*, Oct. 1997.
5. T. Berners-Lee. World wide web seminar. http://www.w3.org/Talks/General.html, 1991.
6. C. Brabrand, A. Møller, M. Ricky, and M. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, 2000.
7. C. Brabrand, A. Møller, and M. Schwartzbach. The <bigwig> Project. In *ACM Transactions on Internet Technology (TOIT)*, 2002.
8. M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.
9. M. Elsman and K. F. Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *LNCS*, pages 224–238. Springer-Verlag, June 2004.
10. P. Graunke, S. Krishnamurthi, R. Bruce Findler, and M. Felleisen. Automatically Restructuring Programs for the Web. In M. Feather and M. Goedicke, editors, *Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE'01)*. IEEE CS Press, Sept. 2001.
11. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

12. R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.
13. R. Hinze and S. Peyton Jones. Derivable Type Classes. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *ENTCS*. Montreal, Canada, Elsevier Science, 2001.
14. J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.
15. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
16. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
17. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
18. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
19. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishing Company, 1993. ISBN 0-201-41663-8.
20. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.
21. C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings Fifth International Conference on Functional Programming (ICFP'00)*, Sept. 2000.
22. P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In S. Krishnamurthi and C. Ramakrishnan, editors, *Practical Aspects of Declarative Languages: 4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 192–208, Portland, OR, USA, January 19-20 2002. Springer-Verlag.
23. P. Thiemann. A Typed Representation for HTML and XML Documents in Haskell. *Journal of Functional Programming*, 2005. Under consideration for publication.
24. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the Fourth ACM SIGPLAN Intnl. Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 1999. ACM.