PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is an author's version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/33057

Please be advised that this information was generated on 2019-10-17 and may be subject to change.
iData For The World Wide Web
– Generic Programming Techniques for
High-Level Server-Side Web Scripting –

Rinus Plasmeijer¹, Peter Achten¹, and Javier Pomer Tendillo²

¹ Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen {rinus, P.Achten}@cs.ru.nl
² Universidad Politécnica de Valencia japoten@inf.upv.es

Abstract. In this paper we present the iData Toolkit. This toolkit offers web
programmers a novel approach to programming interactive, dynamic
web-sites with state on a high level of abstraction. The used concepts are
inspired on our previous work on GUI programming. The key idea is to
program applications with pure functional data models and functions
from which the desktop GUI is derived automatically. In this paper we
transfer these high level concepts to the web and show that web applica­
tions can be programmed in the same style. In addition we incorporate a
number of improvements to the programming method. Because web tech­
nology is completely different from desktop GUI programming, we had
to design an entirely new implementation method. The essential parts of
this implementation rely on generic programming techniques. This has
resulted in a concise and flexible implementation. The iData Toolkit is
an excellent case study in applying generic programming techniques.

1 Introduction

In this paper we present the iData Toolkit. This toolkit offers web
programmers a novel approach to programming interactive, dynamic web-sites with state. This
work is inspired on our previous work on programming desktop GUI applications.
In that project, we have developed the GEC Toolkit [1, 2, 4, 5]. Key concepts are
the automatic generation of GUI applications from any data type; these data
types are models of a GUI application; manipulating the GUI is really editing
the model value of that GUI; behavior is defined by means of pure functions on model
values. As a result, a GEC Toolkit programmer works only with conventional
data types and functions, instead of low-level widget handling. This brings GUI
programming within easy reach of fairly novice functional programmers.

The main goal of the iData Toolkit project is to achieve the same level of
abstraction for HTML-based, dynamic web-applications that have state. This
is a technical challenge because the web has to be programmed very differently
from the average desktop GUI. Hence, our previous solution and implementation
cannot be applied. In particular, web-technology lacks basic support for state,
sessions, and dynamic behavior. Many solutions to solve this problem have been
proposed (related work, Sect. 7). In our approach the interactive parts of an HTML page, forms, are modelled by means of pure functional data models, iData. Every iData can be transformed generically to HTML, so that they can be plugged in anywhere in an HTML page. HTML pages are typed, using algebraic data types. When the user manipulates an HTML page, he is editing iData components in much the same way as a GEC Toolkit desktop GUI is edited.

In the implementation we rely almost exclusively on generic programming [14,15]. With generic programming, one defines a kind indexed family of over­load­ed functions. Their expressive power comes from the fact that from one single function family a concrete implementation for any type can be derived automatically and that one can deviate from this general recipe for arbitrary types. This project is an excellent example of the expressive power and concis­eness of generic programming. Generic programming has been built in in Clean [22,6] and Generic Haskell [18]. Here we use Clean. We assume that the reader is familiar with functional and generic programming.

Contributions of this research are: (1) We show that the level of abstraction of the iData Toolkit can be made equal to that of the GEC Toolkit. The programmer expresses herself only in terms of pure functional data models and functions. (2) The new solution relies essentially on generic programming. This yields concise and maintainable code. (3) We obtain in the iData Toolkit a better separation of page layout vs. the data models than in the GEC Toolkit. We also improve the integration of the model-view paradigm in the iData Toolkit.

This paper is structured as follows. In Sect. 2 we give a motivating example. The iData Toolkit is presented in a number of steps. In Sect. 3, we introduce the basic framework and provide a collection of algebraic data types to program static HTML pages in a type-safe way. In Sect. 4, we show how to generate forms automatically for every data type. Sect. 5 explains how to make the state of these forms persistent. Finally, in Sect. 6 we add local behaviour and communication to these forms. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 Running Example

To give an idea which level of abstraction we want to achieve when writing web applications, we first give a very simple example of an interactive Clean GUI application using the GEC Toolkit. Suppose that one has defined the type Tree and the function toTree that converts a list into a balanced tree.

\[
\text{Tree a = Node (Tree a) a (Tree a) | Leaf a} \]

\[
\text{toTree :: [a] \rightarrow (Tree a) | Ord a} \]

The base of the GEC Toolkit is a generic function that creates an editor for any (user-defined) data type. Here, two of these editors are created automatically.

\[
\text{t : Tree a = C_i (T_i a)} \]

\[
\text{t :: T a = \overline{C_i (T_i a)}} \]

\[
\text{I} \text{Ord a} \]

The symbol \(\overline{\text{I}}\) appends overloading class restrictions to a function type.
\begin{verbatim}
Start :: *World \rightarrow *World
Start world = startGEC editors [1,5,2] world
  where editors = edit "List" >>> arr toTree >>> display "Balanced Tree"

Editors are created for types [Int] and (Tree Int) with (edit "List") and
(display "Balanced Tree"). Initially, the list [1,5,2] is shown. Both editors show
their values, but only the first allows the user to edit the value. Editors can
be combined with arrows [16, 5]. In this case, any change in the list value is passed
to the function toTree using the arrow combinators >>> and arr. The resulting
balanced tree is passed to the tree editor and displayed. Fig. 1 shows how these
editors initially look like.

Fig. 1. The GEC Toolkit version of our running example. List values, edited in List,
are transformed to a balanced tree and displayed in Balanced Tree.

The example shows that no knowledge of low level primitives is required and
that editors are easily combined. All input is guaranteed to be type correct. We
believe this technology can be used for real world applications: editors can be
specialized to create any user defined look [1, 2], complicated circuits of editors
can be defined [3], and editors can be made for higher order types [4].

An open question is whether such a technology can also be used for making
real world web applications. That this is technically possible is what we want
to demonstrate in this paper. Realistic applications constructed with our new
technique will be presented in a separate paper.

3 Programming Static Web Pages

3.1 Basic Communication Architecture

Before we can make interactive web pages we need to provide basic support that
enables the communication between a Clean executable on the server side and
a browser on a client side. When a page is requested by a browser, a simple
php script is executed which starts a Clean application. The output of the Clean
executable is HTML-code for creating the page which is echoed by the php script

\footnote{Start is the main function of every Clean program.}
\footnote{Clean uses explicit multiple environment passing for handling pure effects. The ex­
ternal world is made explicit by the type *World.}
\footnote{In an expression, list values are always delimited by [ and ].}

\begin{verbatim}

\end{verbatim}
to the browser. The page also stores the state of the application. After deliver­ing the HTML-code, the Clean application stops execution. Each time the user changes the page in the browser by filling in forms or by pushing buttons, update information together with the state information is posted and passed by the php script to the same Clean application which is now started with this additional information given to the executable’s command-line arguments. In this way the application can react interactively. Details are explained later on.

3.2 Type-safe Programming of HTML Code

The Clean executable delivers HTML-code. Instead of defining a set of functions (as for instance done in the WASH/CGI framework [24]), we have chosen to capture the official HTML in an algebraic data type (ADT), with root type 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.

Our approach has the following advantages. (1) One obtains a grammar for HTML which is convenient for the programmer. (2) The type system eliminates type and typing errors that can occur when working in plain HTML. (3) We can define a type driven generic function for generating HTML code. (4) Future changes of HTML are likely to change the ADT only.

Basically, HTML can be encoded straightforwardly into an ADT. There are some minor complications. In Clean, as well as in Haskell [21], 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 don’t 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. (1) The data constructor name represents the corresponding HTML language element. (2) 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. (3) 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. (4) A constructor name is prefixed with ‘ in case its name has to be completely ignored when printed. In this way any indirection to any collection of commonly used attributes can be made in the data type without causing any side effects when printed.

The generic printing routine gHpr implements these naming conventions and prints the correct HTML code. Its definition is straightforward polytypical code;
only the CONS instance is special since it has to handle the conventions mentioned above. This results in a universal HTML printer in only 17 lines of code (loc).

Derived instances can be created for most of these types (73). Types such as HeadTag and BodyTag are not quite regular and require specialization.

generic\(^9\) gFpr a :: PtoF a → PtoF
:: PtoF :=\(^{10}\) (*File → *File\(^{11}\) )

3.3 Defining a Simple Static Web Page in Clean

To be able to write interactive Web applications, we need to get hold of the final state of the application. The abstract type HSt is used for this purpose. HSt is opaque, uniquely attributed, and provides no creation function. This is a standard Clean pattern to pass around state safely, an alternative for monadic programming. Every interactive Clean program has type Start :: *World → *World. We have therefore defined an appropriate wrapper function in the library:

doHtml :: (*HSt t) *World t *World

The programmer’s task remains to define a function that, given an initial state HSt, delivers an HTML page and a final state. How this state handling is used is explained later. If we just want to create a static HTML page, the final state is the same as the initial one. Below we show a Clean program generating a simple Hello World HTML page. The essential part of this code is the mypage function. In subsequent examples we only redefine this function. mkHtml is a utility function to create a standard HTML page with a given title and body.

\[
\text{Start} :: *\text{World} → *\text{World} \\
\text{Start world} = \text{doHtml mypage world}
\]

\[
\text{mypage} :: *\text{HSt} → (\text{Html,*HSt})
\]

\[
\text{mypage hst} = \text{mkHtml "Hi Folks!" [B [] "Hello World"] hst}
\]

\[
\text{mkHtml :: String [BodyTag] *HSt → (Html,*HSt)}
\]

\[
\text{mkHtml title body hst} = (\text{Html (Head [] [Hd_Title title]) (Body [] body)},\text{hst})
\]

4 Generating Forms for Static Web Pages

Programming HTML pages becomes much more complicated when forms are involved. Forms are parts of HTML pages that allow the user to enter information that can be submitted subsequently. In this section we show how these forms can be generated automatically for any custom data type. The main implication of this is that a programmer does not program forms, but rather defines a data model for that form, for which the appropriate HTML code is generated. After generation, these forms can be used in a static HTML page as discussed above.

The generic function gForm generates a form from a data model of type a:

\[^9\] generic \(\text{f a :: T a}\) declares a kind indexed family of functions \(\text{f}\) that are overloaded in \(\text{a}\) with type scheme \(\text{T a}\).

\[^{10}\] :: \(\text{T a'} :\Leftrightarrow T a\) declares that type \(\text{T a'}\) is a synonym for type \(\text{T a}\).

\[^{11}\] Updatable files are made explicit with the type *File.
The type `HMode` indicates whether a value is only pretty printed (HDisplay), or if it can also be edited in the browser (HEdit). The generic function `gForm` returns the corresponding HTML representation for the form as a value of type BodyTag. It also returns the value itself. This might seem a bit odd, but it turns out to be convenient for sophisticated user specialization. The abstract type `HSt` is used to collect the state of the forms. `FormId` is used to give a unique identification to each form. It is the responsibility of the programmer to do this.

### 4.1 Generating Forms for Basic Types

For basic types, `gForm` creates a basic form that can either be edited by the user or only displays the value. We show the code for integers, for other basic type the code is analogous. (`Value` is used as a union type for basic types. `UpdValue` also includes selected constructor names – last alternative.)

```plaintext
12 12 gForm |Int|12 mode formid i hst
13 13 (form,hst) = mkBasicInputForm mode formid (IV i) (UpdI i) hst
14 = ((i ,form) ,hst)
```

This definition shows that editable forms are uniquely identified with a triplet consisting of the unique form identifier (formid), the position of the value in the generic tree (`cntr`), and the value that is edited (`updval`). In addition, whenever the user edits the value (`OnChange`), the script `callClean =: "toclean(this)"` is called. Basic forms in HDisplay mode are read-only (`Inp_ReadOnly`), and use a standard background color to show the user that they cannot be edited.

---

12 `f[[T]]... = ...` specializes the generic function `f` for type `T`.
13 `# p = t` is a non-recursive let-definition in which the scope of `p` extends downwards, but not to `t`. This notation makes explicit multiple environment passing more readable.
14 The operator `++` concatenates two String values.
(De)serialization functions of Clean data types are given below. They are based on existing generic library functions for printing (gPrint) and parsing (gParse). Note that what decodeInfo tries to parse depends on its type context.

```haskell
encodeInfo :: a → String | gPrint[a] 15 a
decodeInfo :: String → Maybe a | gParse[a] a
```

### 4.2 Generating Forms for the other Generic Types

For the generic constructors (UNIT, PAIR, EITHER, FIELD, OBJECT, and CONS) gForm proceeds polytypically. UNIT values are displayed as EmptyBody. (PAIR a b) values are placed below each other. (EITHER a b) values proceed either their left or right value. (FIELD x) values show the read-only field name, and next it the field value. (OBJECT o) values proceed recursively. The form that corresponds with (CONS c) values requires an implementation that requires slightly more HTML programming than forms for basic types. It generates a pull down menu with all data constructors. In HEdit mode, the user can select one of these data constructors. Changes are handled in the same way as with basic types, except that the selected constructor name is passed as argument.

### 4.3 Generating Forms for Special Types

Finally, gForm has been specialized for several standard form elements. For instance, buttons are created by CHButton values:

```haskell
:: CHButton = CHPressed
  | CHButton Int String // button with text label
  | CHButtonPict (Int,Int) String // button with image

derive16 gForm CHButton
```

### 4.4 Running Example

Below we give the web page version of the running example, which initial appearance is shown in Fig. 2 (a). It uses gForm to create the list editor (form_list) and the balanced tree editor (form_tree). Note that the web page is not interactive yet: editing the list has no effect. This is dealt with in the next section.

```haskell
mypage :: *HSt → (Html,*HSt)
mypage hst
  | ((list,form_list),hst) = gForm HEdit "List" [1,5,2] hst
  | ((tree,form_tree),hst) = gForm HDisplay "Balanced Tree" (toTree list) hst
  = mkHtml "Balancing Tree From List"
    [ Br, Txt "List:“, form_list
    , Br, Txt "Balanced Tree:“, form_tree ] hst
```

15 $f[k]$ selects the overloaded function of kind $k$ of the generic function family $f$.
16 `derive f T` exports an instance of type $T$ for the generic function $f$. In an implementation, an instance of $T$ for $f$ is generated generically.
5 Generating Forms for Dynamic Web Pages

Forms are not programmed in HTML, but are specified by the web programmer in terms of data models. These forms are constructed in such a way that they call the callClean script when they are edited. As discussed in Sect. 3.1, this causes the Clean application to be executed. It has to determine what part of the page has been edited, and generate a new page that correctly reflects the changed page. The result is that the application appears to have persistent state.

From this informal explanation, it is clear that the application has to be able to update the value from any of its data models that represent forms (as these are the only sources of edit operations). This is done with the generic function upd (Sect. 5.1). Given this function, the application can reconstruct new data models for all forms in its HTML page, and therefore it can also automatically generate the correct next version of that page (Sect. 5.2). The resulting behavior is illustrated with the running example (Sect. 5.3).

5.1 Updating Form Data Models

The only sources of edit operations in an HTML page are forms. One page can contain several forms. Each form has been modelled by means of a data type. Generically speaking, the only edit operations that can occur are the change of a basic type and the selection of a data constructor for one of these form model types. In Sect. 4 we have shown that every generic component of the data model is identified by a triplet. Whenever an edit action has occurred, the application is provided with this triplet, as well as the complete state of all forms in the HTML page. The generic function upd first searches for the correct part of the
data model that represents the form that was the source of the edit operation. When found, that part of the data model obtains a new value. If this involves a data constructor, then new values need to be created for its arguments. For these purposes gUpd uses the UpdMode data type (see Sect. 4.1 for UpdValue).

\[
generic \text{gUpd} \ a :: \ UpdMode \ a \rightarrow \ (UpdMode, a)
\]

\[
\text{(UpdMode, a)} :: \ UpdMode = \text{UpdSearch} \text{UpdValue} \text{Int} \quad // \text{search for indicated position and update it}
\]

\[
\begin{aligned}
&\mid \text{UpdCreate} \quad [\text{ConsPos}] \quad // \text{create new values if necessary} \\
&\mid \text{UpdDone} \quad // \text{just copy the remaining stuff}
\end{aligned}
\]

The working of gUpd is best illustrated with the case for integers. An existing value is replaced with new somewhere in a generic value \(a\) at position \(cnt\) if \(cnt = 0\), otherwise it is not changed and the position is decreased (alternatives 1–2 of gUpd). The default value for new integers is 0 (alternative 3).

\[
\begin{align*}
gUpd{\{\text{Int}\}} \ (\text{UpdSearch} \ (\text{UpdI new}) \ 0) \ _ &= (\text{UpdDone}, \text{new}) \\
gUpd{\{\text{Int}\}} \ (\text{UpdSearch} \ \text{val} \ \text{cnt}) \ _ &= (\text{UpdSearch} \ \text{val} \ (\text{cnt}-1), \text{i}) \\
gUpd{\{\text{Int}\}} \ (\text{UpdCreate} \ 1) \ _ &= (\text{UpdCreate} \ 1, 0) \\
gUpd{\{\text{Int}\}} \ \text{mode} \ _ &= (\text{mode}, \text{i})
\end{align*}
\]

The remaining code of gUpd proceeds polytypically except when it hits on an \(\text{OBJECT } o\), then its new value is determined by the name of the selected data constructor \(\text{cname}\). At that point, gUpd switches from searching mode into creation mode, in order to create arguments of the data constructor if necessary. The function getConsPath \(:\text{GenericConsDescriptor} \rightarrow \text{ConsPos}\), with \(\text{ConsPos} = \text{ConsLeft} | \text{ConsRight}\) yields the route to the desired data constructor.

\[
\begin{align*}
gUpd\{\text{OBJECT of desc}\} \ gUpdo \ (\text{UpdSearch} \ (\text{UpdC}\ \text{cname}) \ 0) \ \text{OBJECT } o \\
\% \ (\text{node}, o) = gUpdo \ (\text{UpdCreate} \ \text{path}) \ o \\
= (\text{UpdDone}, \text{OBJECT } o)
\end{align*}
\]

\[
\text{where } \text{path} = \text{getConsPath} \ (\text{hd} \ \text{[cons} \ \text{\ldots cons.gtd_conses} \ | \ \text{cons.gcd_name} == \text{cname} ]^{18})
\]

5.2 Generating Dynamic Web Pages

The first thing the application needs to do is to determine the reason why it has been started. There can only be three reasons: (1) A form was edited, and the application had a previous state. The new state must be calculated, given the update and the previous state, using gUpd. (2) No form was edited, and the application had a previous state. The previous state must be regenerated. (3) No form was edited, and the application had no previous state. The application must be initialized. The function updateFormInfo performs this case analysis. It must deserialize the input data that has been passed to the application in its command-line arguments and look for the form with the given identification. For this purpose it uses the function decodeInput (see Sect. 4.1 for UpdValue):

\[
^{17} f\{(\text{OBJECT of } d)\} \text{ gives } f \text{ access to information about the type in the record } d.
\]

\[
^{18} [f \ v \ \text{\ldots } v \leftarrow \text{l} \ | \ p \ v] \text{ is the list comprehension that creates a new list of values } f \ v \text{ where each } v \text{ comes from a list } l \text{ provided that predicate } p \text{ holds.}
\]
decodeInput :: FormId → (Maybe FormUpdate, Maybe a) \& gParse\[\#\] a
:: FormUpdate := (FormId, UpdValue)

It reports whether a form has been edited in its first result, and the form’s
model value, if it can be parsed, in the second result. It should be noted that
this makes the system \textit{type safe}: if the user has entered incorrect data (e.g. 42.0
instead of 42 for an integer form, then parsing fails, and the previous (correct)
value is restored. Given this result, updateFormInfo is able to determine the reason
of executing the application:

\[\text{updateFormInfo} :: \text{FormId} \rightarrow (\text{Bool}, \text{Maybe a}) \& \text{gUpd\[\#\]} a \& \text{gParse\[\#\]} a\]

\[\text{updateFormInfo formid} = \text{case decodeInput formid of}\]
\[(\text{Just (pos, updval), Just oldstate}) \rightarrow (\text{True, Just (snd (gUpd\[\#\] (UpdSearch updval pos) oldstate)))}\]
\[(\_, \text{Just oldstate}) \rightarrow (\text{False, Just oldstate})\]
\[\text{else} = (\text{False, Nothing})\]

Finally, generatePage brings everything together. It can be used conveniently
by the programmer as the higher-order argument of doHTML (Sect. 3):

class gHTML d \& gForm, gUpd, gPrint, gParse d19

\[\text{generatePage} :: \text{HMode} \rightarrow (\text{GForm d}) \& \text{gHTML\[\#\]} d\]
\[\text{generatePage mode formid initdata (inidx, lhsts)}\]
\[\rightarrow ((\text{updview}, \text{body}), (\text{nr}, [\text{formid, mystore}: \text{lhsts}]20))\]
\[\rightarrow (\text{gForm\[\#\]} \text{mode formid newview} (\text{0}, [\text{formid, viewtostore}: \text{lhsts}]))\]
\[\text{where newview} = \text{case updateFormInfo formid of}\]
\[\text{True, Just newview} = \text{newview}\]
\[\text{False, Just oldview} = \text{oldview}\]
\[\text{False, Nothing} = \text{initdata}\]
\[\text{viewtostore} = \text{encodeInfo newview}\]

Actually, generatePage is not part of the implementation, but defined here
only for presentational purposes. The real function that is used by programmers
is mkViewHGEC which is more general, but also more complicated (Sect. 6.1).

5.3 Running Example

The only modification in the \textit{code} of the running example (Sect. 4.4) is to replace
gForm calls with generatePage. The \textit{behavior}, however, is significantly different.
In the previous version, the page did not respond to user actions, but now edit
operations are correctly displayed in the client browser (Fig. 2(b)).

19 This notation is not legal Clean, but we use it nevertheless as a shorthand.
20 In a pattern, \([e_1, \ldots, e_n: l]\) (with \(n > 0\)) denotes a list that starts with elements \(e_1, \ldots, e_n\) and that has a remaining list \(l\).
Abstracting and Composing Dynamic Web Pages

In the previous section we have shown how to construct dynamic web pages that have a persistent state. We now provide the finishing touch: (a) An edited form should react to edited values and perhaps change them into other values. (b) An edited form should send edited value to other forms, thus establishing a circuit of communicating forms. Based on our work on the GEC Toolkit, we know that (a) can be dealt with by means of abstraction [2] (Sect. 6.1), and that (b) can be dealt with by means of editor composition based on Arrows [3,16] (Sect. 6.2).

6.1 Form Abstraction

One way to introduce local behavior to a form that is modelled by means of a value of type d, is to provide a function of type d \to d. A more general, and more powerful, way to do this is to use abstraction. With abstraction, the application works with forms that are modelled by means of values of type d, but that are visualized by means of values of type v. This is a variant of the well-known model-view-controller paradigm [17]. 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 GEC Toolkit. It turns out that it can be integrated smoothly in the iData Toolkit. The relation between a domain d and its view v is given by the following collection of functions (HBimap d v):

\[
\begin{align*}
\text{toHGEC} & : d \to (\text{Maybe } v) \\
\text{updHGEC} & : \text{Bool } \to v \\
\text{fromHGEC} & : \text{Bool } \to v \\
\text{resetHGEC} & : \text{Maybe } (v \to v)
\end{align*}
\]

Domain data models are transformed to views with toHGEC. It can use the previous view data model if necessary. The local behavior of the form that corresponds with the view data model is given by updHGEC. Its boolean argument is true iff the reason for evaluation was an edit action of the user. The boolean has the same role in the function fromHGEC which transforms the view data model back to the domain data model. Finally, resetHGEC is an optional separate normalization after the local behavior function updHGEC has been applied.

Abstraction is incorporated in the iData Toolkit by generalizing generatePage into the following real library function, mkViewHGEC. Its type is:

\[
\text{mkViewHGEC} : (\text{HBimap } d v) \to \text{HMode} \to \text{(GForm } d) \to \text{gHTML} \to v
\]

Its signature is almost identical to that of generatePage. It has an additional argument of type (HBimap d v), and it assumes that all the generic machinery is available for the view data type v instead of d. Its implementation has the same structure as generatePage, but is more verbose because it needs to call the relation functions at several places. We omit its code due to lack of space.

The function mkViewHGEC is a powerful tool to create form abstractions with. Frequently occurring patterns of this function are captured with the functions below. We use them in Sect. 6.2 to implement our combinator library. mkApplyEdit-
HGEC creates a form which behavior copies edited values, but yields the first argument otherwise; \texttt{mkEditHGEC} creates a straight editor form; \texttt{mkStoreHGEC} creates a form that only has simple behavior \((d \rightarrow d)\).

\[
\begin{align*}
\text{mkApplyEditHGEC} & : ~ d \rightarrow (GForm d) \mid \text{gHTML} d \\
\text{mkEditHGEC} & : ~ \text{HMode} \rightarrow (GForm d) \mid \text{gHTML} d \\
\text{mkStoreHGEC} & : ~ (d \rightarrow d) \rightarrow (GForm d) \mid \text{gHTML} d
\end{align*}
\]

### 6.2 Form Composition

The function \texttt{mkViewHGEC} and its specialized versions are sufficient for programmers to construct web-applications that consist of several \texttt{iData} elements which values depend on each other using arbitrary functions. Such programs have the same structure as the running example in Sect. 5.3, but use \texttt{mkViewHGEC} (and its specialized versions) instead of \texttt{generatePage}. Analogous to the GEC Toolkit, we advocate the use of a combinator library, based on arrows, to combine \texttt{iData} components in a page. This transforms the running example into its final shape:

\[
\begin{align*}
\text{mypage} & : \ast \text{HSt} \rightarrow (\text{Html}, \ast \text{HSt}) \\
\text{mypage} \text{hs} & \triangleright\triangleright (\text{...,}[\text{body_list, body_tree:...}], \text{hs}) = \text{startCircuit} \text{ circuit \{1,5,2\} hs} \\
& = \text{mkHTML} \text{ "Balancing Tree From List"} \\
& \quad \begin{array}{c}
\text{Br, Big [ ] "List:"}, \text{body_list} \\
\text{Br, Big [ ] "Balanced Tree:", body_tree } \text{hs}
\end{array}
\end{align*}
\]

where \(\text{circuit} = \text{edit "List")} \triangleright\triangleright \text{toTree} \triangleright\triangleright \text{display "Balanced Tree"}\)

The value \(\text{circuit} : : (\text{HCircuit} \ [\text{Int}] \ (\text{Tree} \ \text{Int}))\) is a combinator expression built with the following arrow based combinator library.

\[
\begin{align*}
\text{class Arrow arr where arr} & : : (a \rightarrow b) \rightarrow \text{arr a b} \\
\langle \triangleright\triangleright \rangle \text{infixr 1} & : : (\text{arr a b}) \ (\text{arr b c}) \rightarrow \text{arr a c} \\
\text{first} & : : (\text{arr a b}) \rightarrow \text{arr (a,c) (b,c)}
\end{align*}
\]

\[
\begin{align*}
\text{instance Arrow HCircuit} \\
& : : \text{HCircuit} a b \\
\text{edit} & : : \text{FormId} \rightarrow \text{HCircuit} a a \mid \text{gHTML}[\cdot] a \\
\text{display} & : : \text{FormId} \rightarrow \text{HCircuit} a a \mid \text{gHTML}[\cdot] a \\
\text{store} & : : \text{FormId} a \rightarrow \text{HCircuit} (a \rightarrow a) a \mid \text{gHTML}[\cdot] a \\
\text{startCircuit} & : : (\text{HCircuit} a b) a \ast \text{HSt} \rightarrow ((b,[\text{BodyTag}]), \ast \text{HSt})
\end{align*}
\]

The combinators \texttt{arr,\triangleright\triangleright} and \texttt{first} do not create \texttt{iData}. These are created with the combinators \texttt{edit, display, and store} which call the \texttt{mkViewHGEC} specialization functions that have been introduced in Sect. 6.1 in that order. Finally, \texttt{startCircuit} takes a circuit structure and an initial input value, and creates it.

### 6.3 Running Example

The running example in Sect. 6.2 is a real \texttt{iData Toolkit} application. If we compare this version with the GEC Toolkit in Sect. 2, we can see exactly identical circuit definitions. This means that we have reached our goal of obtaining the same level of abstraction as in the GEC Toolkit. We have improved the separation
of concerns in the iData Toolkit. In the iData version, we can easily rearrange the layout of the circuit components, as is clearly demonstrated in the call to mkHtml. The iData for list and tree can be swapped, replaced, and arbitrarily replicated. In the GEC Toolkit this can only be achieved by changing the circuit definition.

7 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 [26] on XML transformers in Haskell. The Haskell CGI library by Meijer [19] frees the programmer from dealing with CGI printing and parsing. Hanus uses similar types [13] in Curry. Thiemann constructs typed encodings of HTML in extended Haskell in an increasing level of precision for valid documents [24, 25]. XML transforming programs with GenericHaskell has been investigated in UUXML [7]. Elsman and Larsen [11] have worked on typed representations of XML in ML [20]. 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 [8]. It has been improved upon by means of Powerforms [9], used in the <bigwig> project [10]. 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 [16]. Queinnec states that “A browser is a device that can invoke continuations multiply/simultaneously” [23]. Graunke et al [12] 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.

8 Conclusions

There are many tools and script languages for developing web pages. For interactive web services many pages have to be produced in sequence that interact with the user in a consistent and reliable way. Defining such behavior is difficult. With the iData Toolkit interactive web applications can be specified on a very high level of abstraction. One does not have to worry about low level form
handling, forms are generated automatically given a type, the response of the user is guaranteed to be type correct, the output of one form can be used as input for another form, complicated circuits of forms can be defined using arrow combinators, and the consistent behavior of an interactive session is guaranteed. Programming an interactive web page has the same flavour as programming desktop GUIs using the GEC Toolkit. The iData Toolkit offers a better separation between control of lay-out (the view model) and functionality (the data model).

A high level of abstraction has to be realized using the very low level web technology. Yet the implementation of the iData Toolkit is surprisingly simple, elegant and efficient. This is mainly due to the support for generic programming in Clean. Generic functions are used for generating HTML code, for serialisation (printing) and de-serialisation (parsing) of values of any Clean type (which is used to store a collection of states of an application into an HTML page), for the conversion of Clean data into interactive HTML forms, and the automatic update of values of any type when a form is changed. This makes the iData Toolkit an excellent case study in generic programming for the real world.

References


