# The Implementation of iData

## A Case Study in Generic Programming

Rinus Plasmeijer and Peter Achten

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

**Abstract.** The iData Toolkit is a toolkit that allows programmers to create interactive, dynamic web applications with state on a high level of abstraction. The key element of this toolkit is the iData element. An iData element is a form that is generated automatically from a type definition and that can be plugged in in the web page of a web application. In this paper we show how this automatic generation of forms has been implemented. The technique relies essentially on *generic programming*. It has resulted in a concise and flexible implementation. The iData Toolkit is an excellent demonstration of the expressive power of modern generic (poly-typical) programming techniques.

## 1 Introduction

In this paper we present a novel approach to programming *forms* in dynamic web applications. The low level view, and standard definition, of a form is that of a collection of (primitive) interactive elements, such as text input fields, check boxes, radio buttons, pull down menus, and so on, that provide the application user with a means to exchange structured information with the web application. Seen from this point of view, and if programmed that way, creating forms results in a lot of low level HTML coding. A high level view of forms is to think of them as being editors of structured values of appropriate type. From the type, the low level realization can be derived automatically. This can be done once by the toolkit developer. Seen from that point of view, and if programmed that way, creating forms is all about creating data types. This results in a lot less code plumbing and no HTML-coding at all.

In the iData Toolkit project, we have adopted the high level view of forms described above. We call these high level forms iData. An iData allows the web application user to edit data of some specified data type. The current value of that data type is the *state* of the iData. The rendering of an iData is a form in the low level view. Rendering is determined completely by the type of the state of the iData. Having an explicit concept of state allows us to provide the programmer with fine grained control over its *persistency*. The way a web application works and looks is not exclusively determined by its iData. It also contains non-interactive web elements, such as text, headers, tables, and so on. In the iData Toolkit these are created by the programmer by means of a set of data types that

serve as a typed HTML programming language. In summary, the iData Toolkit has the following main features:

- An iData is a typed interactive unit that can be plugged in a web page.
- An iData has typed state. The programmer controls its persistence.
- Web pages are programmed with data types.

An approach as sketched above can be implemented in any programming language with good support for data types and type-driven programming. Modern functional programming languages such as Clean [21, 3] and Haskell [20] come with highly expressive type systems. One example of type-driven programming is *generic programming*[13, 14, 2], which has been built in in Clean and GenericH∀skell [17]. In this paper we use Clean. We assume the reader is familiar with functional and generic programming. Clean specific language features are explained in the text.

Server side web applications are launched by the web server as soon as a request is received from a web browser. The application produces the requested web page, and then terminates. Because of this behavior, every web application needs to create a solution to *store*, *reload*, and *update* its *intermediate state*. Many solutions to this problem have been invented. Two of them are server side storage, and enhancing the web page with state information that is invisible to the application user. We adopt a combination of these two solutions.

We show that generic programming provides us with concise and flexible solutions for most of the major aspects of web programming: serialization and deserialization, storage and retrieval of intermediate application states, printing and parsing of the HTML data type language, tracking down and fixing edited data structures of arbitrary type. In all, the iData Toolkit project is an excellent case study in generic programming.

This paper is structured as follows. We first present the HTML programming language in Sect. 2. With this language, the application developer can create arbitrary HTML pages. Next, we show how the iData Toolkit can automatically generate forms out of iData elements in Sect. 3. These iData elements have a state and a visualization that can be used by the application developer in the HTML pages. When a user manipulates a page with iData, the action of the user needs to be recovered, as well as the states of the iData elements, and a new collection of iData elements with possibly modified state need to be created. We show in Sect. 4 how this can be done generically. The last step is to obtain a good separation of the iData logic and its visualization in Sect. 5. Sect. 6 gives a small example to give an impression of the expressiveness of the iData Toolkit. We discuss related work in Sect. 7 and conclude in Sect. 8.

## 2 Programming Static Web Pages

A server side web application is started by the web server as soon as a request is received from a client web browser. The web application receives its input data on *stdin* and sends its output data on *stdout*. The output data contains the

web page that is sent back by the web server to the client web browser. Hence, the purpose of every web application is to produce a web page that reflects the application's response to the given input. In addition, it has to retrieve state information that is either encoded in the input date, or has been stored on disk. These standard actions are performed by the library function

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

(In Clean, function arguments are separated by whitespace instead of →. The main function `Start` of an interactive application has type `*World → *World`. Clean uses explicit multiple environment passing for handling pure effects. The `World` value represents the external environment of an application. The uniqueness attribute `*` in front of the type constructor guarantees *single-threaded* access to values of this type [6, 7].)

The application specific behavior is provided by the programmer with the function of type

```
(*HSt → (Html,*HSt))
```

The abstract type `HSt` collects all states during the construction of an HTML page. We defer its discussion until Sect. 3.2. Here, we focus on the `Html` type, which is the root type of a collection of algebraic data types that capture 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* (`A`) and ending with *variables* (`Var`) (in total there are 76 HTML tags). The latter three alternatives are for easy HTML generation: `STable` generates a 2-dimensional table of elements, `BodyTag` turns lists of elements into a single element, and `EmptyBody` can be used as a zero element. Attributes are encoded as `FooAttr` data types.

The library function `mkHtml :: String [BodyTag] *HSt → (Html,*HSt)` creates a simple HTML page with given title and content.

```
mkHtml :: String [BodyTag] *HSt → (Html,*HSt)
mkHtml title tags hst
  = (Html (Head [`Hd_Std [Std_Title title]] []) (Body [] tags),hst)
```

Consider the following example of a tiny "Hello world" page.

```
Start :: *World → *World
Start world = doHtml (mkHtml "Hello World Example" [Txt "Hello World!"]) world
```

The corresponding HTML code is

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

Basically, HTML can be encoded straightforwardly into a set of algebraic data type. There are some minor complications. In Clean, as well as in Haskell, 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. **(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.

This 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 algebraic data types only.

The generic printing routine `gHpr` implements the naming conventions discussed above, and prints the correct HTML code.

```
generic gHpr a :: *File a → *File
```

(**generic** $g\ a$ :: $T\ a$ declares a kind indexed family of functions $g$ that are overloaded in $a$ with type scheme $T\ a$. `*File` represents a file on disk that can be updated in place, guarded by the uniqueness attribute.) 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 less than 20 *loc*. For completeness we show its code.

```
gHpr{|String|}       file s          = file <<< s
// Other basic type instances proceed analogously
gHpr{|[]|}     gx    file xs         = foldl gx file xs
gHpr{|UNIT|}         file _          = file
gHpr{|PAIR|}   ga gb file (PAIR a b) = gb (ga file a) b
gHpr{|EITHER|} gl gr file (LEFT   l) = gl file l
gHpr{|EITHER|} gl gr file (RIGHT  r) = gr file r
gHpr{|OBJECT|} go    file (OBJECT o) = go file o
gHpr{|CONS of t|} gc file (CONS   c)
| t.gcd_name.[0] == '‘' = gc file c
| t.gcd_arity == 0      = file <+ " " <+ print t.gcd_name
| t.gcd_arity == 1      = gc (file <+ " " <+ print t.gcd_name <+ " = ") c
| otherwise             = gc (file <+ " " <+ print t.gcd_name         ) c
where print             = toLower o stripprefix
```

(<<< is an overloaded operator that writes its second argument to the first argument of type `*File`. `o` is function composition. `<+` is a shorthand for `gHpr{|*|}`.

$g\{|\kappa|\}$ selects the overloaded function of kind $\kappa$ of the generic function family $g$.) Derived instances can be created for most of the HTML types (73). Types such as `HeadTag` and `BodyTag` are not quite regular and require specialization (requiring 8 *loc* and 90 *loc* respectively).

## 3 Rendering iData

In the previous section we have shown how an iData Toolkit application developer can program 'raw' HTML code. This HTML code may contain forms, but as we have explained, we propose to create forms automatically from the type of the data that they are supposed to represent. In this section we show how to render forms from the state of iData. We first show how HTML rendering of iData is taken care of in Sect. 3.1, and then explain what state handling is required for this in Sect. 3.2.

### 3.1 Generating Forms from Types

Given a model value of type `m`, then a form is generated by `gForm`:

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

The form is represented by the record type (`Form m`). The `changed` field of this record holds if the user has edited the form. The `value` field is its current value. The `form` field is the actual HTML rendering of the form.

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

Because the state of forms need to be stored (either in the web page or on disk), they have to be identified unambiguously. This is what `FormId` values are for. It is the task of the application developer to use unambiguous names (`String`s). `FormId` values are created with one of the functions $\{\mathtt{n},\mathtt{s},\mathtt{p}\}[\mathtt{d}]$`FormId :: String →` `FormId`. In addition to the name, the programmer has control over the life span and edit mode of the iData element.

```
:: FormId   = { id::String, lifespan::Lifespan, mode::Mode }
:: Lifespan = Page | Session | Persistent
:: Mode     = Edit | Display
```

The life span of an iData element is determined by $\{\mathtt{n},\mathtt{s},\mathtt{p}\}$: its value is garbage collected automatically after each page creation (`n`), is stored persistently during a session (`s`), or independent of sessions (`p`). By default, values can be edited in the browser. If they should be displayed only, then one of the $\{\mathtt{n},\mathtt{s},\mathtt{p}\}$`dFormId` functions can be used.

For basic types, `gForm` creates basic forms. We show the code for integers, for other basic types the code is analogous. (`Value` is used as a union type for basic types. `UpdValue` also includes selected constructor names – last alternative.)

```
gForm{|Int|} formid i hSt
   # (form,hSt) = mkInput formid (IV i) (UpdI i) hSt
```

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

:: UpdValue = UpdI Int | UpdR Real | UpdB Bool | UpdS String | UpdC String
:: Value    = IV   Int | RV   Real | BV   Bool | SV   String | NQV  String

mkInput :: FormId Value UpdValue *HSt → (BodyTag,*HSt)
mkInput formid val updval hSt=:{cntr}
  = ( Input [ Inp_Type Inp_Text, Inp_Value val, Inp_Size defsize
            : case mode of Edit    = [ Inp_Name    identify
                                     , `Inp_Std    [EditBoxStyle]
                                     , `Inp_Events [OnChange callClean]]
                           Display = [ Inp_ReadOnly ReadOnly
                                     , `Inp_Std    [DisplayBoxStyle] ]] ""
    , {hSt & cntr=cntr+1} )
where identify = encodeInfo (formid.id,cntr,updval)
```

(♯ is a non recursive let definition which scope extends downwards, but not to its right hand side. $e$=:$p$ binds variable $e$ to pattern $p$; [$e_1, \ldots, e_n$:$l$] (with $n > 0$) denotes a list that starts with elements $e_1$ upto $e_n$ and that has a remaining list $l$; $\{r \ \& \ \overrightarrow{f_i = v_i}\}$ is a record equal to $r$, but with fields $f_i$ having values $v_i$; $r.f$ selects field $f$ of record $r$.) Basic forms in `Display` mode are read-only, and show this to the user. Basic forms in `Edit` mode need to ressurect the web application on the server side, and provide it with the proper information. Whenever the user edits the value (`OnChange`), the script `callClean =: "toclean(this)"` is called. This script sends the states of all forms and an *identification triplet* of the edited element back to the server, causing the application to be started with the new data. The identification triplet consists of the unambiguous form identifier (`formid.id`), the position of the value in the generic representation (`cntr`), and the value that is edited (`updval`). Together with the collection of all states, this is sufficient to recover the old state and compute the next state of the web application (Sect. 4).

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 recursively and display either their left or right value. (`OBJECT o`) values proceed recursively. The form that corresponds with (`CONS c`) values requires more HTML programming.

```
gForm{|CONS of t|} gc formid (CONS c) hst=:{cntr}
   ♯ (nc,hst)     = gc formid c {hst & cntr=cntr+1}
   = ( { changed = nc.changed
       , value   = CONS nc.value
       , form    = [ STable [Tbl_CellPadding (Pixels 0)
                            ,Tbl_CellSpacing (Pixels 0)]
                            [[selector,BodyTag nc.form]]]
       }, hst )
where
    allConses= map (λn → n.gcd_name) t.gcd_type_def.gtd_conses
    consIndex= allConses??t.gcd_name
    selector = Select [Sel_Name "CS" : cstyle]
```

```
                [Option
                   [ Opt_Value (encodeInfo (formid.id,cntr,UpdC cons))
                   : if (j == consIndex) [Opt_Selected Selected:ostyle] ostyle]
                   cons \\ cons <- allConses & j <- [0..]]
    (cstyle,ostyle)
            = case formid.mode of
                Edit    -> ([ 'Sel_Std  [Std_Style width, EditBoxStyle]
                            , 'Sel_Events [OnChange callClean]] ,[])
                Display -> ([ 'Sel_Std  [Std_Style width, DisplayBoxStyle]
                            ,  Sel_Disabled Disabled] ,['Opt_Std [DisplayBoxStyle]])
    width   = "width:" +++ toString defpixel +++ "px"
```

It generates a pull down menu which entries correspond with all data constructors. In `Edit` 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. All in all, `gForm`'s implementation requires 140 *loc*.

Finally, `gForm` has been specialized for several standard form elements. We do not discuss their implementation. It is basically in the same style as the `Int` instance defined above.

### 3.2 Storing Form States

In the iData Toolkit, the state of a web application is the set of the states of all iData. While a page is generated, these states are collected in the abstract type `HSt`. It extends the Clean environment `world :: World` with a global counter `cntr :: InputId` to generate position values in the generic representation of the states, and the form `states :: *FormStates` that are constructed for a page.

```
:: *HSt      = { cntr::InputId, states::*FormStates, world::*World }
:: InputId :== Int
```

(:: $T\ \overrightarrow{a}$ :== $T'\ \overrightarrow{a}$ declares that type $T\ \overrightarrow{a}$ is a synonym for type $T'\ \overrightarrow{a}$.) `FormStates` stores the serialized states of forms together with their `FormId` value and if they have been changed (either by the user or by the web application). `FormStates` is basically an association list with a look up function `findState` and update function `replaceState`. These require the `World` environment in case of `Persistent` forms. The boolean result of `findState` is true iff a previous state was present. Finally, these functions are overloaded because of their use of the generic serialization functions `gParse` and `gPrint`.

```
findState    :: FormId   *FormStates *World
        -> (Bool,Maybe a,*FormStates,*World) | gParse{|*|} a
replaceState :: FormId a *FormStates *World
        -> (             *FormStates,*World) | gPrint{|*|} a
```

(| appends overloading class restrictions to a function type.)

In addition, `FormStates` stores the *edit* operation of the user that caused the application to be launched. The edit operation is determined by the element that has been changed (the identification triplet discussed in Sect. 3.1), and the

new value that has been entered by the user. This information is retrieved from `FormStates` by the function

```
getUserEdit :: *FormStates → ((Maybe a,Maybe b),*FormStates)
                           | gParse{|*|} a & gParse{|*|} b
```

This function is overloaded in its first result because the data is stored in serialized form. For convenience, the identification string of the form that has been edited by the user is stored separately. It is retrieved by

```
getUpdateId :: *FormStates → (String,*FormStates)
```

# 4 Creating iData

In the previous section we have shown that the rendering of an iData is a form. If the application programmer plugs these forms in the web page of the application, then they become available to the application user, who can start manipulating them. Every manipulation that changes the current value of a form triggers the execution of the application on the server side. The application has to figure out why it has been launched. There can be only three reasons:

1. *No form was edited, and there was no previous state.* Initialize all forms.
2. *No form was edited, and there are previous states.* Recover all previous states.
3. *One form was edited, and it had a previous state.* Calculate the new state, given the update information and the recovered previous state.

It is not the task of the programmer to determine these actions. This is delegated to each application of the pivotal iData creation function, `mkViewForm`. The programmer uses this function to create all of his iData. Because of the complexity of `mkViewForm`, we first present a slightly simplified version, viz. `simplified_mkViewForm`. The full implementation of `mkViewForm` follows in Sect. 5.

In Sect. 4.1 we first show the generic function `gUpd` that can update a selected part of any data structure with a new value of the correct type. This essential tool is used by `simplified_mkViewForm` in Sect. 4.2 to compute a new state of a form in case it has been edited by the user.

## 4.1 Updating the State of iData

The function `gUpd` constructs the new model value of type `m` of a form. It must be a generic function because it needs to traverse the generic data representation of the old model value in order to locate the generic element that has been changed. This location has been passed to the application with the identification triplet, as explained in Sect. 3.1.

```
generic gUpd m :: UpdMode m → (UpdMode,m)

:: UpdMode = UpdSearch UpdValue InputId | UpdCreate [ConsPos] | UpdDone
```

The `UpdMode` type represents the two passes `gUpd` goes through: (`UpdSearch newv cnt`) represents the search for the generic element at location `cnt` with new value `newv`, and (`UpdCreate path`) represents the creation of new values for a selected data constructor that can be found at `path` (`:: ConsPos = ConsLeft | ConsRight`).

We illustrate the working of `gUpd` for basic types with the case for integers (the other cases for basic types are analogous). An existing value is replaced with `new` somewhere in a generic value 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).

```
gUpd{|Int|} (UpdSearch (UpdI new) 0) _ = (UpdDone,new)
gUpd{|Int|} (UpdSearch val cnt)      i = (UpdSearch val (cnt-1),i)
gUpd{|Int|} (UpdCreate l)            _ = (UpdCreate l,0)
gUpd{|Int|} mode                     i = (mode,i)
```

The remaining code of `gUpd` proceeds polytypically except for `OBJECT`s:

```
gUpd{|OBJECT of desc|} gUpd_obj (UpdSearch (UpdC cname) 0) (OBJECT obj)
   # (mode,obj) = gUpd_obj (UpdCreate path) obj
   = (UpdDone,OBJECT obj)
where path = getConsPath (hd [cons \\ cons ← desc.gtd_conses
                                 |  cons.gcd_name == cname ])
```

(`[`$f\ v$ `\\` $v \leftarrow l$ `|` $p\ v$ `]` is the *list comprehension* that creates a new list of values $f\ v$ where each $v$ comes from a list $l$ provided that predicate $p$ holds.) In this case its new value is determined by the name of the selected data constructor (`cname`). At that point, `gUpd` switches from searching mode into creation mode, in order to create arguments of the data constructor. The function `getConsPath` `:: GenericConsDescriptor → [ConsPos]` yields the route to the desired data constructor.

### 4.2 Updating the iData

In this section we define a simplified version of the `mkViewForm` function, viz. `simplified_mkViewForm`. At the beginning of Sect. 4, we mentioned the three situations in which forms need to be updated. The function `findFormInfo` performs this case analysis. It must deserialize the input data that has been passed to the web application and look for the form with the given identification. For this purpose it uses the function `decodeInput`:

```
:: FormUpdate :== (InputId,UpdValue)

decodeInput :: FormId *FormStates *World
            → (Maybe FormUpdate,(Bool,Maybe m,*FormStates,*World)) | gParse{|*|} m
decodeInput formid fs world
   # (updateid,fs) = getUpdateId fs
   | updateid == formid.id
      = case getUserEdit fs of
          ((Just (sid,pos,UpdI i),newi),fs) // case distinction on Int
             # prev_state = findState {formid & id=sid} fs world
```

```
                  # ni          = case newi of (Just ni) → ni; _ → i
                  = (Just (pos,UpdI ni),prev_state)
            (_,fs) = ... // case distinction on other basic types
    | otherwise
      = (Nothing, findState formid fs world)
```

This function checks whether the iData element that is identified by `FormId` has been edited by the user. If so, its exact location in the generic representation is returned (of type `FormUpdate`), as well as its current value (the result of using `findState` - see Sect. 3.2). For this reason, `decodeInput` requires the `FormStates` and `World` environments. It should be noted that `findState` may fail to parse the input. This makes the system *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 the result of `decodeInput`, `findFormInfo` is able to determine the reason of executing the application (the numbers to the right coincide with the cases in the beginning of this section):

```
findFormInfo :: FormId *FormStates *World → (Bool,Maybe m,*FormStates,*World)
                                          | gUpd{|*|}, gParse{|*|} m
findFormInfo formid formStates world
   = case decodeInput formid formStates world of
      (Just (cnt,newv),(changed,Just m,formStates,world))                (3.)
         # m = if changed (snd (gUpd{|*|} (UpdSearch newv cnt) m)) m
         = (True, Just m,  formStates,world)
      (_,(_,Just m,formStates,world))                                    (2.)
         = (False, Just m, formStates,world)
      (_,(_,_,formStates,world))                                         (1.)
         = (False, Nothing,formStates,world)
```

The `simplified_mkViewForm` function brings everything together:

```
class gHTML m | gForm, gUpd, gPrint, gParse m

simplified_mkViewForm :: FormId m *HSt → (Form m,*HSt) | gHTML{|*|} m
simplified_mkViewForm formid init_m {states,world}
   = calcnextView init_m (findFormInfo formid states world)
where
   calcnextView :: m (Bool,Maybe m,*FormStates,*World) → (Form m,*HSt)
                                                       | gHTML{|*|} m
   calcnextView init_m (isupdated,maybe_m,states,world)
      # m              = case maybe_m of Nothing    = init_m
                                         Just new_m = new_m
      # hSt            = {cntr=0,states=states,world=world}
      # (mform,{states,world})
                       = gForm{|*|} formid m hSt
      # (states,world) = replaceState formid mform.value states world
      # mform          = {changed=isupdated,value=m,form=mform.form}
      # hSt            = {cntr=0,states=states,world=world}
      = (mform,hSt)
```

`simplified_mkViewForm` first determines the reason why the web application was started using `findFormInfo`. Given this information, it can generate the form for the correct value `m`, using `gForm`. Finally, the new value of the form is stored in the `FormStates` data structure, and the form and the updated administration are returned.

## 5 iData Abstraction

In the previous section we have shown how to construct web pages with iData elements. When the user manipulates these iData elements, the application responds with the appropriate update action and generates a new page. The iData in a web page present a direct visualization of their state values. Two final aspects are lacking:

1. Applications usually impose restrictions on edited values that go beyond the expressiveness of the type system. For this reason they need to be able to inspect edited values themselves, and perhaps change them into other values.
2. iData elements need to be able to present their values in any manner that is suitable to the application. In general this requires a different type than their state type. This implies that presentation concerns are not well separated from logic concerns.

Based on earlier work, we know that both aspects can be dealt with by means of *abstraction* [1]. We improve upon the method by providing a seamless integration of abstraction with the iData Toolkit. With abstraction, the application works with iData that have state 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 [16]. What is special about it in this context, is that views are also defined by means of a data type, and hence can be handled generically in exactly the same way. This is a powerful concept, and we have used it successfully in the past.

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

```
:: IBimap m v = { toView    :: m → Maybe v → v
                , updView   :: Changed → v → v
                , fromView  :: Changed → v → m
                , resetView :: Maybe (v → v) }
:: Changed    = { isChanged :: Bool
                , changedId :: String        }
```

Model values are transformed to views with `toView`. It can use the previous view value if available. The local behavior of an iData element is given by `updView`. Its first argument records if it has been changed (`isChanged :: Bool`), and the unambiguous name of the iData element that has been changed (`changedId :: String`). This argument of type `Changed` has the same role in the function `fromView` which transforms view values back to model values. Finally, `resetView` is an

optional separate normalization *after* the local behavior function `updView` has been applied.

Abstraction is incorporated in the iData Toolkit by generalizing `simplified_mkViewForm` into the following real library function, `mkViewForm`. Its type is:

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

Its signature is almost identical to that of `simplified_mkViewForm`. It has an additional argument of type (`IBimap m v`), and it assumes that all the generic machinery is available for the view data type `v` instead of `m`. Its implementation has the same structure as `simplified_mkViewForm`. The function `calcnextView` is more verbose because it needs to render the *view* instead of the *model* value. This also explains why it is in general possible that the creation of an iData element with model value `m` returns an iData element with different output value `m'`. This is clearly illustrated by the highlighted sections in the adapted definition of `calcnextView` below.

```
mkViewForm :: FormId m (IBimap m v) *HSt → (Form m,*HSt) | gHTML{|*|} v
mkViewForm formid init_m bm {states,world}
  = calcnextView init_m bm (findFormInfo formid states world)
where
  calcnextView :: m (IBimap m v) (Bool,Maybe v,*FormStates,*World)
                 → (Form m,*HSt) | gHTML{|*|} v
  calcnextView init_m bm (isupdated,maybe_v,states,world)
    # v                          = bm.toView init_m maybe_v
    # v                          = bm.updView isupdated v
    # m                          = bm.fromView isupdated v
    # v                          = case bm.resetView of
                                     Nothing    = v
                                     Just reset = reset v
    # hSt                        = {cntr=0,states=states,world=world}
    # (vform,{states,world})     = gForm{|*|} formid v hSt
    # (states,world)             = replaceState formid vform.value states world
    # mform                      = {changed=isupdated,value=m,form=vform.form}
    # hSt                        = {cntr=0,states=states,world=world}
    = (mform,hSt)
```

The function `mkViewForm` is a powerful tool to create form abstractions with. Frequently occurring patterns of this function have been captured with wrapper functions. Consider `mkEditForm` below. It can be used as a 'store' in `Display` mode, or as a straight editor in `Edit` mode.

```
mkEditForm :: FormId m *HSt → (Form m,*HSt) | gHtml{|*|} m
mkEditForm formid=:{mode} m hst
  = mkViewForm formid m
      { toForm    = λnew old → case old of (Just v) → v; _ → new
      , updForm   = case mode of Edit → λ_ v → v; Display → λ_ _ → m
      , fromForm  = λ_ v → v
      , resetForm = Nothing } hst
```

## 6 Example

In order to get an impression of the expressiveness of the iData Toolkit, we give a small example of a web application with which the user can edit arguments of type `args` that are applied to a given function of type `args → res`. The application displays the results of type `res`. The function (`apply name args f`) generates the page for function `f` with given `name` and initial arguments `args`:

```
apply :: String args (args → res) *HSt → (Html,*HSt) | gHtml{|*|} args
                                                     & gHtml{|*|} res
apply name args f hSt
    # (argsF,hSt) = mkEditForm (nFormId "args") args hSt
    # (resF, hSt) = mkEditForm (ndFormId "res") (f argsF.value) hSt
    = mkHtml "Function Application"
        [ STable [] [[Txt name:argsF.form], [Txt " = ":resF.form]] ] hSt
```

Two iData are created. The first is the *args* form (`argsF`) for editing arguments. The second is the *res* form (`resF`) for displaying the result, hence a *display* `FormId` is used. It uses the value of the `argsF` form to compute the proper result. In the page, the forms are placed in two subsequent rows with an additional label.

In Fig. 1 two examples of this application are shown. The first example tests

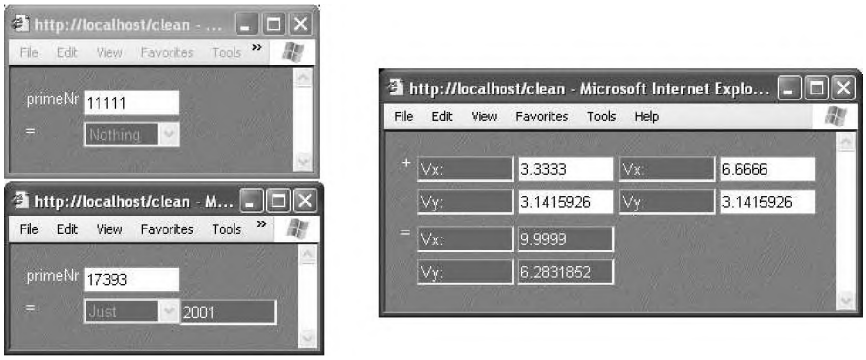


**Fig. 1.** **(a)** Determine the prime index number. **(b)** Summing vector values.

the function `primeNr :: Int → Maybe Int`. If its argument is a prime number, then it tells you which one it is (first, second, etc., where 2 is the first prime number). If it is not a prime number, then `Nothing` is returned.

```
Start world = doHtml (apply "primeNr" 1 primeNr) world
```

The second example illustrates the fact that forms are generated from types. In this example `+` is overloaded for a new type for 2-dimensional vectors, defined as `:: V = {vx::Real,vy::Real}`. The program is generated with:

```
Start world = doHtml (apply "+" (z,z) (λ(a,b) → a+b)) world
where z = {vx=0.0,vy=0.0}
```

## 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 [25] on XML transformers in Haskell. The Haskell CGI library by Meijer [18] frees the programmer from dealing with CGI printing and parsing. Hanus uses similar types [12] in Curry. Thiemann constructs typed encodings of HTML in extended Haskell in an increasing level of precision for *valid* documents [23, 24]. XML transforming programs with GenericH$\forall$skell has been investigated in UUXML [4]. Elsman and Larsen [10] have worked on typed representations of XML in ML [19]. 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 [5]. It has been improved upon by means of Powerforms [8], used in the <bigwig> project [9]. 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 [15]. Queinnec states that "A browser is a device that can invoke continuations multiply/simultaneously" [22]. Graunke *et al* [11] 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 high level of abstraction. Web applications consist of static HTML parts, usually for presentation purposes, and interactive forms, for user interaction. This distinction is explicit in the toolkit. We provide an abstract version of forms, iData. Forms are generated from iData, and can be plugged in in arbitrary HTML pages. The HTML pages are constructed using a library of algebraic data types. This eliminates many type errors, and provides good documentation for programmers. The programmer can create intricate relationships between iData, using

standard functional programming techniques. Although the implementation of the toolkit using advanced programming techniques, we have kept the API of the toolkit as simple as possible. Basic knowledge of functional programming is sufficient to get started.

A high level of abstraction has to be realized using the very low level web technology. Yet the implementation of the iData Toolkit is concise, elegant, and efficient. This is mainly due to the support for generic programming in Clean. Generic functions are used for generating HTML code, for serialization and de-serialization of values of any Clean type, 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.

## Acknowledgements

Jan Kuper coined the name iData for our editor components. Pieter Koopman provided input for the gUpd function. Paul de Mast kindly provided us with a web server application written in Clean which has allowed us to readily test the iData Toolkit. Javier Pomer Tendillo, as an Erasmus guest, has been very helpful in setting up the iData Toolkit, and find out the nitty-gritty details of HTML programming.

## 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. *Generic Functional Programming - Conceptual Design, Implementation and Applications.* PhD thesis, University of Nijmegen, The Netherlands, 2005. ISBN 3-540-67658-9.
3. 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.
4. 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.
5. 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.
6. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, volume 6, pages 579–612, 1996.
7. E. Barendsen and S. Smetsers. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. World scientific, 1999.

8. 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.
9. C. Brabrand, A. Møller, and M. Schwartzbach. The <bigwig> Project. In *ACM Transactions on Internet Technology (TOIT)*, 2002.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.
16. 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.
17. A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 141–152. ACM Press, 2003.
18. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
19. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
20. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
21. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/~clean/.
22. 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.
23. 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.
24. P. Thiemann. A Typed Representation for HTML and XML Documents in Haskell. *Journal of Functional Programming*, 2005. Under consideration for publication.
25. 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.