iData For The World Wide Web
Programming Interconnected Web Forms

Rinus Plasmeijer and Peter Achten

Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, Toernooiveld 1, 6525ED Nijmegen, Netherlands
rinus@cs.ru.nl, phone: +31 (0)24 3652644, fax: +31 (0)24 3652525
P.Achten@cs.ru.nl, phone: +31 (0)24 3652483, fax: +31 (0)24 3652525

Abstract. In this paper we present the iData Toolkit. It 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 can be regarded as a self-contained object that stores values of a specified type. Generic programming techniques enable the automatic generation of HTML-forms from these types. These forms can be plugged into the web application. The iData elements can be interconnected. Complicated form dependencies can be defined in a pure functional, type safe, declarative programming style. This liberates the programmer from lots of low-level HTML programming and form handling. We illustrate the descriptive power of the toolkit by means of a small, yet complicated example: a project administration. The iData Toolkit is an excellent demonstration of the expressive power of modern generic (poly-typical) programming techniques.

Keywords server side web programming, web forms, functional programming

1 Introduction

The World Wide Web is experiencing a rapid growth of web based applications. For many companies their web based services are their only contact with clients. These clients gain access to these applications via a wide variety of browsers. In addition, they tend to use these applications in a browsing style: clients clone windows, move back and forward through visited links, visit different sites, and so on. These aspects impose strong demands on web applications in order to assure correct behavior. It is important that these web applications are constructed in a well-understood way, and are based on solid foundations. In the iData Toolkit project we are working on a framework for these purposes. The main features of the toolkit are:

– The toolkit is based on a single concept, that of *i*nteractive *Data*, or iData.
– The web application is a single program (perhaps consisting of several modules) instead of a collection of loosely coupled script pages.

– The toolkit is defined in a pure functional programming language, and uses features such as strong typing and expressive type systems.
– The toolkit offers a good separation of concerns. The computational content of the web application is separated from the presentation in a clear way.
– The programmer has fine grained control over the life span of the application's state. State can be stored persistently, session based, or page based.

We focus on two challenges when programming the web: the first is how to program *forms* in a concise, abstract, and well-defined way, and the second is how to *interconnect* these forms. Forms are the interactive parts of web applications. In them, users can enter data, using a variety of interactive elements such as text input fields, (radio) buttons, and pull down menus. An application page generally consists of several forms which content may depend on each others state. We guarantee that user input is always type correct, and that the server side application always receives the correct data.

We meet the above challenges by imposing a typed discipline on the untyped world of web programming in a novel way. In our framework, forms are really *editors of values*. Because we use a strongly typed programming language, these *values have a well-defined type*. We *derive the form automatically from the type* of the value of an editor using *generic programming techniques* [13, 14, 2]. Such an editor is an iData. This results in a powerful abstraction: iData Toolkit programmers do not program forms, but instead design types and values of iData. We have implemented this approach earlier for Graphical User Interfaces [1]. The implementation of the iData Toolkit [20] is both entirely different and a major improvement of the previous work.

Generic programming has been built in in Clean [21, 3] and GenericH∀skell [17]. We use Clean. Clean details are explained in the text. We assume that the reader is familiar with functional and generic programming.

Contributions presented in this paper are:

– We present a single programming concept, the iData, with which dynamically, interconnected, type-safe web applications can be developed.
– We show that iData offer a high level of abstraction because they are programmed in terms of data models. Forms are rendered automatically from the type of these data models.
– iData can be interconnected type-safely, as if they were stateful objects.

This paper is structured as follows. In Sect. 2 we introduce the iData Toolkit by means of a few key examples. We show what steps an iData Toolkit application programmer goes through by discussing a case study of a small, yet complex and realistic example of a web form in Sect. 3. We discuss related work in Sect. 4. Finally, we conclude in Sect. 5.

2 The concept of iData

An iData element is an object with two major components: **(i)** a *state*, or *value*, which type is determined by the programmer, and **(ii)** a *form*, or *rendering*, which is derived automatically by the toolkit from the state and its type.

The programmer manipulates the iData in terms of the state and its type, whereas the application user manipulates the iData in terms of a low-level form. Clearly, the iData Toolkit needs to mediate between these two worlds: every possible type domain has to be automatically mapped to editable forms, and every user action on these forms has to be automatically mapped back to the original type domain, with a possibly different value.

In this section we explain the main concepts of the iData Toolkit by means of a few key examples. Please notice that although the code of these examples has a static flavour, each of these examples are complete interactive web applications. First we discuss the architecture of server side web applications.

2.1 Architecture

The applications that we study reside on web servers. They are launched by the web server whenever a (remote) web browser program sends a request for an HTML page. It is the task of the application to *compute* an HTML page, and then *terminate*. The web server takes care that the generated HTML page is sent back to the web browser program.

In our approach, a web application consists of two parts: the declaration of the interconnected iData elements, and the generation of the HTML page that contains (a subset of) the automatically derived forms of these iData elements.

Interconnection of iData is programmed in a pure functional data dependency style. This gives the program a highly declarative flavor. Yet, the application is started from scratch every time a web form is altered by the user. The current state of the program is completely determined by the iData elements. They are re-created each time the program is started, and are able to recover their current state (possibly changed by the user). To make this possible, the serialized state of an iData is stored either at the server on disk or in the HTML page. All iData elements therefore automatically always contain a type correct value reflecting the latest changes made by a user. For recalculation of a page, and even of a complete web site, the same algorithm can be re-used taking the current iData states as starting point, enabling the highly declarative style of programming.

iData Toolkit applications compute HTML pages. There are many possible approaches to handle this (Sect. 4). This aspect of the iData Toolkit was not our priority We have chosen an approach that fits in our framework, i.e. an approach that uses data types to model output. The HTML that is computed by the application is encoded with algebraic data types, using a *types-as-grammar* approach [25]. This has the following advantages. **(1)** We get a complete context free grammar for HTML which is convenient for the programmer. **(2)** The type system eliminates type and typing errors that can occur in plain HTML. **(3)** Compiling an instance of this data type to HTML code is done by a compact type driven generic function. **(4)** Such a generic implementation is very robust, future changes of HTML are likely to change the type definitions only. A snapshot of the algebraic data types is:

```
:: 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] | ...
           | H1     [Hnum_Attr]  String    | ...
           | Var    [Std_Attr]   String
           | STable [Table_Attr] [[BodyTag]] | BodyTag [BodyTag] | EmptyBody
```

The last three data constructors of `BodyTag` are not part of HTML, but are provided for programming convenience. The data constructor `STable` generates a 2-dimensional table, the data constructor `BodyTag` turns a list of body tag elements into a single body tag, and `EmptyBody` can be used as an empty element.

The code below shows the standard overhead of every iData Toolkit program:

```
module FLOPS2006Examples
import StdEnv, StdHtml                                          1.

Start :: *World → *World                                        2.
Start world = doHtml example world                              3.
```

The proper library modules need to be imported (line 1). Lines 2–3 declare the main function of every Clean program. The *uniqueness attribute* `*` just in front of `World` guarantees that values of this type are always used in a *single threaded manner*. Clean uses *uniqueness typing* [6, 7] to allow destructive updates and side-effects. The opaque type `World` represents the entire external environment of the program. The iData program is given by the function `example :: *HSt → (Html,*HSt)`. The wrapper function `doHtml` turns this function into a common Clean program. It initializes the `HSt` value with all serialized values that can be found in the HTML page, and includes the `World` as well. This implies that every iData Toolkit application has full access to the external world, and can, for instance, connect to databases and so on. Below, we only show the `example`*i* functions, and skip the standard overhead.

2.2 iData **Have Form**

The first example demonstrates the fact that iData elements are type driven. A simple `Int` iData is created (Fig. 1(**a**)).

```
example1 :: *HSt → (Html,*HSt)
example1 hst
    ♯ (nrF,hst) = mkEdit (nIDataId "nr") 1 hst
    = mkHtml "Int editor" [ H1 [] "Int editor", BodyTag nrF.form ] hst
```

Passing multiple environments around explicitly is supported syntactically in Clean by means of ♯-definitions. These are non-recursive *let*-definitions, which scope extends to the bottom, but not the right-hand side. This is the standard approach in Clean. Even though the examples in this paper do not exploit the flexibility of multiple environment passing (by for instance connnecting to a database system), we present them in this style. The function `mkEdit` (Sect. 2.5) declares an iData element `nrF::IData Int` with initial value `1::Int`. The element is identified with the value `(nIDataId "nr")::IDataId` (Sect. 2.5). The `IData` record holds the `form` rendering of the iData, its `value`, and a boolean that states iff this particular iData element has been `changed` by the user:

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

Key features that are illustrated in this small example are the declaration of an iData element (`nrF`) from an identification value and an initial value, and that this iData element has an automatically derived rendering in terms of a form that can be addressed by `nrF.form`. It is a general property of an iData that a user can only enter input that is type-safe. When a user creates wrong input, the previous value (of correct type) is restored. If an initial value of some other type would have been specified as argument of `mkEdit`, a corresponding, yet completely different iData element is generated, with a rendering that allows only input of the appropriate type. Finally, the declaration of iData is robust against ill-typed use: only if the current `HSt` value contains a serialized representation of a value of the correct type, then the iData uses the deserialized value of the correct type; otherwise it relies on its initial value. Hence, all iData declarations with the same label but different type use their own initial value. If the declaration of an iData *updates* the `HSt`, then it may be the case that the type of the reference is modified accordingly. It is the responsibility of the application programmer to use unambiguous names throughout his program. Although this approach is not fail-safe, it is easy to explain to programmers, and problems can be avoided by using separate declaration functions (Sect. 3.2). We are still investigating if better solutions exist or need to be created.

Note that the definition of the web page, given by the function `mkHtml :: String [BodyTag] *HSt → (Html,*HSt)`, is cleanly separated from the declaration of the iData. At this spot we can freely mix HTML code that is automatically generated from iData elements with "ordinary" hand-written HTML code.

2.3 iData Have Value

In this example we show that, besides a form, iData also have a value (Fig. 1**(b)**).

```
example2 hst
    ♯ (nrFs,hst) = seqList [mkEdit (sumId nr) nr \\ nr ← [1..5]] hst
    = mkHtml "Numbers" [ H1 [] "Numbers", sumtable nrFs ] hst

sumtable nrFs = STable [] (                [nrF.form  \\ nrF ← nrFs] ++
                           [[toHtml (sum [nrF.value \\ nrF ← nrFs])]])
sumId i       = nIDataId ("sum" <$ i)
```

Five iData elements are activated: `nrFs :: [IData Int]` (`seqList` *fs st* threads a state value *st* through a list of state transformer functions *fs* and collects their results and the final state). The function `sumtable` places their *forms* in a column, underneath of which the sum of their *values* is displayed. The value of an iData is given by the `value` field of that iData. The library function `toHtml` uses the generic form rendering function we also use for the iData to render values of arbitrary type into HTML. The overloaded operator `<$` appends a `String` version of its second argument to its first argument.

2.4 iData Have Sharing

Repeated use of the same iData declaration refers to a shared iData object. A first advantage of this scheme is that an iData can be seen as a store of a value of a certain, arbitrary type. Where values are actually being stored depends on the kind of iData created (see 2.5). Hence, we do not need to introduce a separate concept to store data. A second advantage is that both the value and rendering of iData can be used arbitrarily many times in a HTML page without causing ambiguity problems. We illustrate the latter by replicating the column of integer iData and their sum in the example below (Fig. 1**(c)**):

```
example3 hst
    ♯ (nrFs,hst) = seqList [mkEdit (sumId nr) nr \\ nr ← [1..5]] hst
    = mkHtml "Numbers"
        [ H1 [] "Numbers", STable [] [[sumtable nrFs],[sumtable nrFs]] ] hst
```

Editing any of the iData elements also automatically affect the other iData in the same row. The sum is displayed twice, at the bottom of both columns.

2.5 iData Have Model-View Separation

So far we have seen that the rendering one obtains for free from an iData element is completely determined by its type. What if we don't like this particular rendering? Suppose that for this particular example we want to replace the default integer editor boxes by iData elements that are counters. These counters have different self contained behavior: pressing the buttons should increment/decrement the integer value. This warrants good separation between model (integer value) and view (counter). Indeed, in the example code we only have to replace `mkEdit` by `counterIData` that we will define later on to obtain the desired program that displays five counters instead of five integer editors (Fig. 1**(d)**).

```
example4 hst
    ♯ (nrFs,hst) = seqList [counterIData (sumId nr) nr \\ nr ← [1..5]] hst
    = mkHtml "Numbers" [ H1 [] "Numbers", sumtable nrFs ] hst
```

The counter iData ensures that its integer value is incremented/decremented at every corresponding button press. Although we have created an iData element with a completely different behaviour (view), it still returns an integer value to the programmer. The model has not changed, and therefore nothing else in program has to be altered, since iData elements are self contained and fully compositional. But how can a programmer define these counters?

Creating iData forms. The one *pivotal* library function `mkIData` allows the definition of all sorts of iData elements one can imagine. It has type signature:

```
mkIData :: IDataId m (IBimap m v) → IDataFun m
         | gForm{|⋆|}, gUpd{|⋆|}, gPrint{|⋆|}, gParse{|⋆|} v
:: IDataFun m :== *HSt → (IData m,*HSt)
```

With `mkIData` any model-view mapping can be made. The polymorphic type variable `m` stands for *model*, and the generically overloaded type variable `v` stands for *view*. Class restrictions on this generic variable `v` appear after `|`. It shows that `mkIData` obtains its power by making use of four generic functions (of kind ⋆):

– `gForm` creates a form from a view type,
– `gUpd` converts any change made by the user with a browser in a form to a corresponding change in the view value,
– `gPrint` serializes values of any type for iData storage, and
– `gParse` de-serializes values of any type for iData recovery.

For the programmer all these generic functions addressed in the context restriction have as small consequence that he has to tell Clean to automatically derive these generic functions (see [2, 3]) for the user defined types that he wants to view. In order to visualize a user defined view type `Type`, somewhere in the program the programmer has to define

```
derive gForm Type; derive gUpd Type; derive gPrint Type; derive gParse Type
```

Clean function types show their *arity* by separating arguments with white-space, Therefore, `mkIData` requires *three* arguments. Let's take a closer look at these arguments. The *first* argument of `mkIData` is of type `IDataId`.

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

The `id` field of this record type is used to unambiguously identify iData elements. The programmer creates them by making up `String` identifiers, which is a typical way of identifying forms in web applications. It is the task of the programmer to use names in such a way that every use of (`mkIData id`) refers to the same iData element of some type `m`. We already saw in the sharing example that one can refer multiple times to the same iData element. The `lifespan` field controls the *life span* of the corresponding iData value: its value is either remembered as long as the same page is being viewed (`lifespan` = `Page`), or during a browser session (`lifespan` = `Session`), or independently of sessions (`lifespan` = `Persistent`). Persistent storage simply means that instead of storing a serialized representation of the value of an iData in the HTML page, the serialized value is written to and read from disk on the server side. Finally, the *edit mode* of iData can be set. This mode is typically editable (`mode` = `Edit`), but sometimes they should only display constant values (`mode` = `Display`). For convenience, for any kind of thinkable `IDataId` combination, a library function is offered {n,s,p}[d]`IDataId :: String → IDataId`. Here, `n`, `s`, `p` represent the `lifespan` values `Page`, `Session`, and `Persistent` respectively. If `d` is included, the `mode` is `Display`, otherwise it is `Edit`.

The *second* argument of `mkIData` is its initial value. This initial value is used when an iData element is created for the first time or if no matching iData was found in the `HSt` environment. This happens for instance when a web page is viewed for the first time.

The *third* and final argument of `mkIData` is the most complicated one because it is used to define the model-view abstraction. This allows the application to work 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 our approach, is that a view is also determined by its data type. The type can be regarded as a model of a view, and hence can be handled generically in exactly the same way! This is clearly expressed in the type signature of `mkIData`, which states that the generic machinery must be available for the view model `v`.

The mapping between a model `m` and its view `v` has to be given by defining an instance of the following record type `IBimap m v`:

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

The record contains model-view conversion functions and functions to enable the desired self contained behavior. Model values are transformed to views with `toView`. It can use the previous view value if available. The self contained behavior of an iData element is handled by `updView`. Its first argument records if the view has been changed by the user. The same argument is passed to `fromView` which transforms updated view values back to model values. Finally, `resetView` is an optional separate normalization of the updated view value.

The *result* of `mkIData` is an `*HSt` environment function of type `IDataFun m` that yields a (`IData m`) value. The abstract type `*HSt` is constructed by the iData Toolkit immediately after the application has been restarted and contains the serialized states of all views. The non-persistent view states are stored in the HTML form and transmitted whenever the page is changed. Persistent view states reside on disk on the server side and are read when needed.

Integer as model, Counter as view. Now that we have explained the most important function of the iData Toolkit, we can show how a self contained counter can be defined as a view for an integer model.

First of all, we need some button machinery. In the iData Toolkit, all imaginable input forms, such as labelled buttons, image buttons, radio buttons, and pull down menus, are predefined by specializing types to these input forms. In Sect. 2.6 we show how programmers can use the very same specialization mechanism for their own purposes. As an example we show the predefined type for a pull down menu and a button. Both are used in Sect. 3.

```
:: PullDownMenu = PullDown (Int,Int) (Int,[String])
:: Button       = Pressed | LButton Int String | PButton (Int,Int) String
```

A value (`PullDown` (v,w) $(i,elts)$) is shown as a pull down menu of width w that displays v elements of *elts*. The index of the selected element is i. A value

(`LButton` *w l*) creates a *w* pixels wide button with label *l*. A value (`PButton` *(w,h) p*) creates a button that is *w* pixels wide and *h* pixels high, and that has a picture at file path *p*. Whenever a button is pressed, its iData value is set to `Pressed`.

Second of all, we need to specify layout. By default, arguments of data constructors are placed in a column, top- and right-aligned with the data constructor. As we have seen in the examples above, tables are useful to specify different layouts. For convenience, we have introduced a number of types to lay out elements in rows and columns. Furthermore, 2,3,4-tuples layout their elements in a row.

Elements such as the above can be used by defining iData of values of these types. A `Counter` for an integer value can be constructed by adding an up and down button to it. This results in the following (synonym) type:

```
:: Counter :== (Int,Button,Button)
```

We can now straightforwardly define `counterIData` in terms of `mkIData`. To express the mapping between an integer model and a counter view, we need to define `toView`, `updView`, `fromView`, and `resetView`:

```
counterIData :: IDataId Int → IDataFun Int
counterIData iDataId i          = mkIData iDataId i ibm
where ibm       = { toView      = λn v → useOldView (n,down,up) v
                  , updView     = λ_ v → updCounter v
                  , fromView    = λ_ (n,_,_) → n
                  , resetView   = Nothing }
      (up,down) = (LButton (defpixel / 6) "+",LButton (defpixel / 6) "-")

      updCounter :: Counter → Counter
      updCounter (n,Pressed,_) = (n - 1,down,up)
      updCounter (n,_,Pressed) = (n + 1,down,up)
      updCounter noPresses     = noPresses

useOldView :: a (Maybe a) → a
useOldView new (Just old)= old
useOldView new Nothing   = new
```

Frequently Used Views The function `mkIData` is a very powerful function with which any model-view abstraction can be defined. Frequently used patterns are predefined in the library. Two examples used in this paper are:

```
mkEdit :: IDataId m → IDataFun m | gForm{|*|},gUpd{|*|},gPrint{|*|},gParse{|*|} m
mkEdit iDataId m = mkIData iDataId m { toForm    = useOldView
                                     , updForm   = modeUpd iDataId.mode m
                                     , fromForm  = λ_ v → v
                                     , resetForm = Nothing }
where modeUpd Edit     _     _ newv = newv
      modeUpd Display initm _ _     = initm

mkSelf2::IDataId m (m → m) → IDataFun m | gForm{|*|},gUpd{|*|},gPrint{|*|},gParse{|*|} m
```

```
mkSelf2 iDataId m f = mkIData iDataId m { toForm    = useOldView
                                        , updForm   = λ_ v → f v
                                        , fromForm  = λ_ v → v
                                        , resetForm = Nothing }
```

The `mkEdit` function was used in examples 2.2 and 2.3. It can be used as a 'store' in `Display` mode, or as a straight editor in `Edit` mode. iData can also be used to create an intelligent store with custom behavior. iData that are declared with `mkSelf2` memorize a value, initialized with the second argument of the function. When declared, the iData applies the argument function `f` to its value (by `updForm`). In this way stores can preserve properties: e.g. one can ensure that a stored list is always sorted by defining a sort function as parameter. Because iData can be shared, the programmer is able to parameterize `mkSelf2` with different function arguments. In this way, the stored state can be manipulated from the outside. In combination with the `pdIDataId` function, this results in a persistent memory store which obeys these properties.

2.6 iData Have Specialization

iData can be specialized, just as generic functions can. The generic mechanism can render a value of any type. With specialization one can overwrite the default way this is done. This cannot only be used to create buttons and the like when certain types are being used, but it can also be used to customize the look and feel of any user defined type. By using specialization one can separate the handling of the functionality of the web page (by the programmer) from the way things look (by the designer of a site).

Suppose the designer likes the counters in Sect. 2.5 much better than the default integer editors that were used in Sect. 2.2 and 2.3. Assume that he wants to ensure that, throughout the program, these counters are being used instead of the plain integer boxes. For this purpose he needs to specialize the generic form rendering function `gForm` for the `Int` type. This is done by:

```
gForm{|Int|} iDataId i hst = specialize asCounter iDataId i hst
where asCounter :: IDataId Int → IDataFun Int
      asCounter iDataId i   = λhst
          ♯ (counterF,hst) = counterIData iDataId i hst
          = ( { changed    = counterF.changed
              , value      = fst3 counterF.value
              , form       = counterF.form }, hst )
```

The `asCounter` function that defines the specialization uses `counterIData` as defined in 2.5. The library function

```
specialize :: (IDataId a → IDataFun a) IDataId a → IDataFun a | gUpd{|⋆|} a
```

is able to 'plug in' the specialization function into any arbitrary other iData structure. Given this specialization for `Integers`, in any place where an iData of an `Integer` value is needed, a counter iData will be made. In such a setting, the programs 2.2, 2.3, and 2.4 all have self contained counters instead of integer boxes without any change to be made in the presented code.

3 Case Study: Project Administration Web Application

As a case study we construct a small, yet complicated form that could be part of a site with which (simple) projects are administrated. It consists of a dynamic number of interconnected sub-forms. Its size is 250 *loc* (including empty lines): 50% handles the problem domain, 12% specialization, and 38% form programming. The screenshot in Fig. 2 shows the application with all sub-forms active.

3.1 Basic Logic of the Project Adminstration

We assume that the types and algorithms needed to do the actual project administration have been defined and designed separately, without any knowledge of the web interface that has to be created on top of it. Let's assume that to administrate projects, the following, self-explanatory, types have been defined.

```
:: Project     = { plan ::ProjectPlan, status::Status, members::[Worker] }
:: ProjectPlan = { name ::String, hours ::Int }
:: Status      = { total::Int,    left  ::Int }
:: Worker      = { name ::String, status::Status, work::[Work] }
:: Work       :== (Date,Int)
:: Date        = Date Int Int Int
```

We assume that for the maintenance of the project adminstration, suitable initialization, update, and retrieval functions are defined on these data structures, such as `initProject :: String Int → Project`. Their definitions are straightforward. We use them where needed, but skip their definition for lack of space.

3.2 Designing Forms by Defining Types

The screenshot in Fig. 2 reveals that we have defined at least three iData input forms: one to define a new *project*, one to add a *worker* to an existing project, and one to assign worked *hours* for an existing worker of an existing project. These are located below each other. The details view on the right hand side is not an iData, but just displays all information of one existing project. Clearly, the iData have a strong interconnected behavior: only if a project exists, then workers can be added to it; working hours can be assigned only for workers on projects they participate in. We show how to specify these dependencies.

For each iData we need a corresponding type, since iData's forms are generated type driven. Sometimes, we can use (a combination of) existing types, and sometimes we need to define new types. Because iData can be shared, it is good practice to define a separate declaration function for each of them. In case of straight editors, we use the `mkEdit` function. If we need to be able to impose properties on the state of an iData, we use the `mkSelf2` function. So, for every iData we give the type and define a creation function.

The *project* form can be of type `ProjectPlan`, which is given above. The creation function is `projectIData`. Initially we assume that no projects are planned.

```
projectIData :: IDataFun ProjectPlan
projectIData = mkEdit (nIDataId "project") (initProjectPlan "" 0)
```

For the *worker* form we define a new type `WorkerPlan`. It holds all `project` names a worker is involved in, the worker's `name`, and the `hours` that should be added to a particular project. The *worker* form must know all projects that have been entered. The function `adjWorkers :: [Project] WorkerPlan → WorkerPlan` updates the worker plan with all new project plan names. For this reason, it is a customizable editor created with `mkSelf2`.

```
:: WorkerPlan    = { project::ProjectList, name::String, hours::Int }
:: ProjectList :== PullDownMenu

workerIData :: (WorkerPlan → WorkerPlan) → IDataFun WorkerPlan
workerIData f
   = mkSelf2 (nIDataId "worker") (initWorkerPlan "" 0 0 initProjects) f
```

For the *hours* form we define the type `DailyWork`. For a given project (in `projectId`), and a given worker (in `myName`), it stores how many hours have been worked on a particular date. Because this form depends on the current list of projects and associated workers, this declaration function is also created with `mkSelf2`. The function `adjDailyWork :: [Project] DailyWork → DailyWork` updates the daily work value with all currently registrated project-worker combinations.

```
:: DailyWork   =   { projectId   ::ProjectList, myName::WorkersList
                   , hoursWorked::Int,         date  ::Date }
:: WorkersList :== PullDownMenu

hoursIData :: (DailyWork → DailyWork) → IDataFun DailyWork
hoursIData f
   = mkSelf2 (nIDataId "hours") (initDailyWork 0 0 initProjects) f
```

Of course we need to store the whole project administration of type `[Project]`. This can be achieved by using a persistent iData. Again, we make its declaration function `adminIData` parametrized. By now, the pattern should be clear.

```
adminIData :: ([Project] → [Project]) → IDataFun [Project]
adminIData f = mkSelf2 (pdIDataId "admin") initProjects f
```

Finally, the user manipulates the forms of the application. Changes to the database are committed by pressing one of the buttons to add a *project*, *worker*, or *hours*. The corresponding actions are given by the functions

```
addNewProject  :: ProjectPlan [Project] → [Project]
addNewWorkplan :: WorkerPlan  [Project] → [Project]
addDailyWork   :: DailyWork   [Project] → [Project]
```

The library function `ListFuncBut` associates `m → m` functions with buttons, and yields an `(IData (m → m))` which value is either one of the selected functions or the identity function.

```
ListFuncBut :: Bool IDataId [(Button, m → m)] → IDataFun (m → m)
```

With this function, we can concisely specify the buttons of the application:

```
btnsIData::DailyWork WorkerPlan ProjectPlan → IDataFun ([Project] → [Project])
btnsIData daylog workplan project
  = ListFuncBut False (nIDataId "mybuttons")
             [ (LButton defpixel "addProject", addNewProject  project )
             , (LButton defpixel "addWorker",  addNewWorkplan workplan)
             , (LButton defpixel "addHours",   addDailyWork   daylog  ) ]
```

3.3 Interconnecting iData

To create the desired web application we need to do two things: we have to declare and interconnect all iData and we have to deliver an HTML page that contains the renderings of these iData. We do not discuss the latter aspect: it is not essentially different from the tiny examples given in Sect. 2. Interconnecting the iData is what matters:

```
example hst
   ♯ (projectF,hst) = projectIData   hst                                  1.
   ♯ (workerF, hst) = workerIData id hst                                  2.
   ♯ (hoursF,  hst) = hoursIData  id hst                                  3.
   ♯ (btnsF,   hst) = btnsIData   hoursF.value workerF.value
                                  projectF.value hst                      4.
   ♯ (adminF,  hst) = adminIData  btnsF.value    hst                      5.
   ♯ (workerF, hst) = workerIData (adjWorkers   adminF.value) hst         6.
   ♯ (hoursF,  hst) = hoursIData  (adjDailyWork adminF.value) hst         7.
   = mkHtml "projectadmin" [ H1 [] "Project Administration"               8.
                           ... /* not shown due to lack of space */ ] hst
```

First, the three user forms, *project*, *worker*, and *hours*, are declared (lines 1-3). As a result, they recover their possibly altered state. Then the buttons are declared (line 4). If the user has pressed one of them, then the value of `btnsF` is the associated administration update function. This function, `btnsF.value`, is then applied in the declaration function of the complete administration (line 5). Given the new administration, the *worker* and *hours* form need to be updated with the new project and workers lists (line 6-7). For this reason the latter two forms occur twice in the code. This is a typical iData Toolkit programming pattern. The program guarantees that users can only add workers to existing projects and hours to existing workers.

4 Related Work

iData components are form abstractions. A pioneer project to experiment with form-based services is Mawl [5]. The <bigwig> project [9] uses Powerforms [8]. Both 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>). Powerforms reside on the client-side of a web application.

The type system is used to filter out illegal user input. They advocate compile-time systems, just as we do, because this allows one to use type systems and other static analysis. The main differences are that in our approach *all first order user types* are admissible in iData, that iData are automatically derived from these types, and that we can use the expressiveness of the host language to obtain higher-order forms/pages.

Continuations are a natural means to structure interactive web applications. This has been done by Hughes [15], using his Arrow framework; Queinnec [22], who takes the position that continuations are at the essence of web browsers; Graunke *et al* [11], who 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 by recovering its previous state, merging the user modification, if any, and computing the proper next state that is re-rendered.

Many authors have worked on creating and manipulating HTML (XML) pages in a strongly typed setting. Early work is by Wallace and Runciman [26] 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∀skell has been investigated in UUXML [4]. Elsman and Larsen [10] have worked on typed representations of XML in ML [19]. Our *types-as-grammar* approach eliminates all syntactically incorrect programs, but we have not put effort in eradicating all semantically incorrect programs. Our research interest is in the automatic creation of forms from type specifications, and less in the definition of the HTML pages in which they reside.

5 Conclusions and Future Work

In this paper we have presented the iData Toolkit, an innovative toolkit for the construction of server side web applications. The toolkit is founded on a strongly typed, pure, functional programming language with support for generic programming. The key concept of the toolkit is the iData element. A web application is a function that computes an HTML page. Forms in this page are derived automatically by the iData Toolkit from the typed states of the declared iData elements. Each and every iData handles its initialization, state recovery, and rendering. The result is that applications can be defined in a concise and declarative style.

In this paper, we have illustrated the expressiveness of the iData Toolkit by means of several small examples, and one larger case study. To test the suitability of the iData Toolkit for constructing real world applications, we have created all kinds of applications, such as a fully functional CD-shop site. Also for these larger web applications we have observed that they can be defined in the same concise and declarative way as the examples in this paper.

We believe that the conciseness of programs, the use of a single iData concept, and the embedding in a functional programming language, are important factors to enable reasoning about these programs. We think that the iData Toolkit provides a step in the direction of formal reasoning about dynamic, type-safe, server side web applications.

Acknowledgements

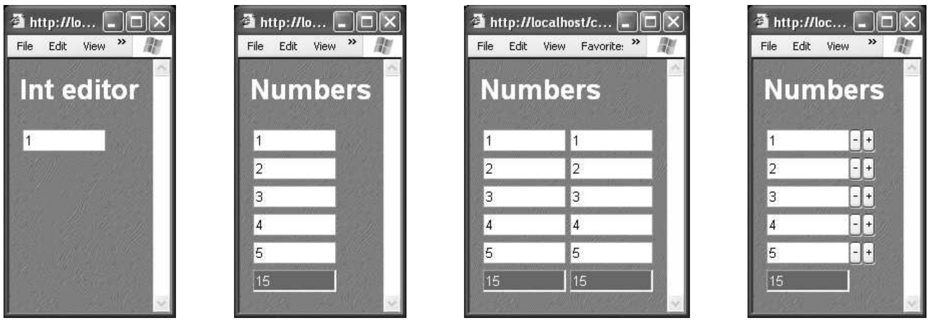
Jan Kuper coined the name iData for our editor components. Paul de Mast kindly provided us with a Clean web server application to readily test the toolkit. Javier Pomer Tendillo has been very helpful in setting up the iData Toolkit, and find out the nitty-gritty details of HTML programming. We thank Pieter Koopman and Maarten de Mol for reviewing this paper and help us improving it. Finally, we thank the referees of this paper for their useful comments.

References

1. P. Achten, M. van Eekelen, R. Plasmeijer, and A. van Weelden. GEC: a toolkit for Generic Rapid Prototyping of Type Safe Interactive Applications. In *5th International Summer School on Advanced Functional Programming (AFP 2004)*, volume 3622 of *LNCS*, pages 210–244. Springer, August 14-21 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. R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. Technical Report No: TCD-CS-2005-60.
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. A. van Weelden, S. Smetsers, and R. Plasmeijer. A Generic Approach to Syntax Tree Operations. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. Technical Report No: TCD-CS-2005-60.
26. 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.

Appendix



Int editor

1

Numbers

1
2
3
4
5
15

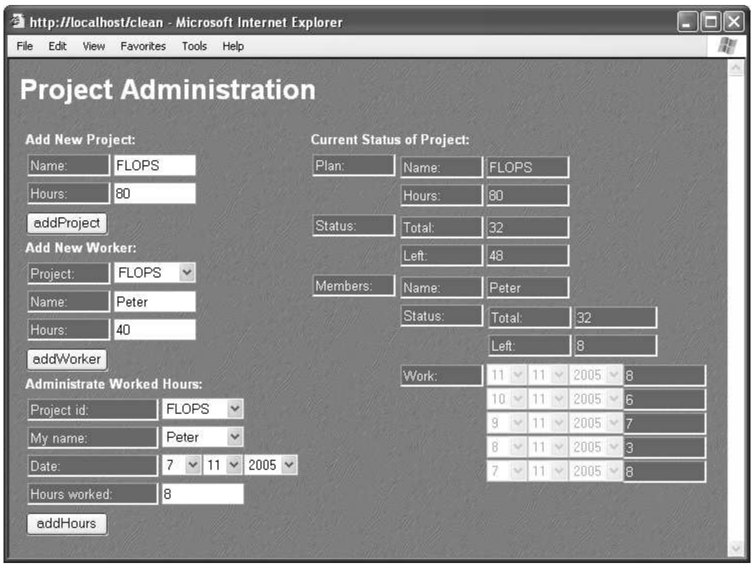
Numbers

1	1
2	2
3	3
4	4
5	5
15	15

Numbers

1 - +
2 - +
3 - +
4 - +
5 - +
15

Fig. 1. Screen shots of the initial state of the toy examples in Sect. 2. **(a)** A simple integer iData. **(b)** Summing the value of iData. **(c)** Sharing iData. **(d)** Model-View separation of iData.


http://localhost/clean - Microsoft Internet Explorer
File Edit View Favorites Tools Help
addProject
addWorker
addHours

Project Administration

Add New Project:

Name:	FLOPS
Hours:	80

Add New Worker:

Project:	FLOPS
Name:	Peter
Hours:	40

Administrate Worked Hours:

Project id:	FLOPS
My name:	Peter
Date:	7 11 2005
Hours worked:	8

Current Status of Project:

Plan:	Name:	FLOPS		
	Hours:	80		
Status:	Total:	32		
	Left:	48		
Members:	Name:	Peter		
	Status:	Total:	32	
		Left:	8	
	Work:	11 11 2005	8	
		10 11 2005	6	
		9 11 2005	7	
		8 11 2005	3	
		7 11 2005	8	

Fig. 2. Screen shot of the project administration case study in Sect. 3.