A Conference Management System based on the iData Toolkit

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 purely functional toolkit for the Clean programming language to create highly dynamic, interactive, thin client web applications on a high level of abstraction. Its main building block is the iData element. With this element the programming effort of the application programmer is reduced significantly because it takes care of state handling, rendering, user interaction, and storage management automatically. In this paper we show that it can be used for even more tasks: handle *destructively shared model data*, perform *version management*, and *state consistency management*. This can be done entirely on top of the iData Toolkit. The toolkit comes with a new programming paradigm. We illustrate the extended power of the toolkit and programming paradigm by a case study of a conference management system.

1 Introduction

The purely functional language Clean has a library to create highly dynamic, interactive, thin client web applications on a high level of abstraction. This library is the iData Toolkit [11, 13, 12].It is based on the language support for generic programming [2, 3]. The toolkit's main building block is the iData element, which is a versatile unit that automates a great deal of things for the programmer:

- it manages a state of arbitrary type;
- it renders an HTML form representation of its state;
- it handles user actions made with these forms in a type safe way;
- it stores its state either in the page or at the server side on disk.

Web applications are created by interconnecting an arbitrary collection of iData elements via their states and rendered forms. In the past years we have obtained experience in programming applications with iData elements, and their desktop GUI predecessors, the GEC elements of the GEC Toolkit [1]. This has resulted in a new programming paradigm. In the iData Toolkit programming paradigm the application programmer *models* the application as an *information system*, by identifying the entities and entity-relations and specify them as pure functional data structures and pure functions. The generic power of the toolkit is used subsequently to handle as much as possible automatically. Human intervention is still required, but the power of generic programming is that it allows application programmers to specialize the generic scheme where needed.

When constructing programs with the programming paradigm, it turns out that the 'classic' version of the toolkit has a number of limitations:

- Model types are pure functional data structures. Although functional languages can define and handle *shared* data structures, they cannot handle *destructively shared* data because this destroys referential transparency. However, in information systems destructive sharing is a natural phenomenon, because data should not be stored redundantly. Hence, an iData Toolkit application programmer can not model destructive sharing directly, but instead has to program this on top of the functional data structures and for each and every edit operation. This is cumbersome, error-prone, and an example of boilerplate code that should be automated once and for all.
- It is important in multi-user web applications with several persistent shared states to manage *versions* of these states correctly. Again, the programmer might be able to program this, but it should be dealt with once and for all.
- The final limitation concerns the *consistency of states*. The iData Toolkit is *edit driven*, i.e.: it reacts to (type safe) edit operations of the application user who can alter a part of the state of one of the iData elements. In general, it may well be the case that during a sequence of edit operations, the set of states is *inconsistent*. In that case, the application should not commit this configuration of states to disk, but rather work on a local version.

In this paper we show that the above concerns can be handled automatically by the iData elements, on top of the 'classic' iData Toolkit. We believe that this provides further evidence to the fact that iData elements form a powerful abstraction mechanism to create highly interactive and dynamic web applications with. We illustrate the use of the new techniques by studying the case of a *conference management system*. Conference management systems are software systems that support conference managers, programme committee members, and authors with a number of tasks, such as the electronic paper submission process, paper distribution and reviewing process, deadline management, and the paper discussion process. They serve as a good example of the domain of web applications that suffer from the limitations that have been presented above. We show that the resulting system widens the application domain of the toolkit while still adhering to its programming paradigm.

This paper is structured as follows: we first briefly present the iData Toolkit in Sect. 2. Next, in Sect. 3, we discuss the case study of a conference management system. Implementation details are presented in Sect. 4. Finally, related work is discussed in Sect. 5, and we conclude in Sect. 6.

2   The iData Toolkit

In this section we present the 'classic' iData Toolkit, i.e. the toolkit without the extensions that are discussed in the next sections. First, we give an informal explanation of iData elements, which are the building blocks of the iData Toolkit (Sect. 2.1). Second, we present the programming paradigm (Sect. 2.2).

2.1 iData elements

iData elements are the fundamental building blocks of the iData Toolkit. An iData element is a typed unit that provides the application user with a GUI (an HTML form) that allows him to edit values of that given type only. The GUI is derived automatically from the type and value using the generic programming facilities of Clean. In this paper, we use one toolkit function to create iData elements:

```
class iData d | gForm{|*|}, gUpd{|*|}, gPrint{|*|}, gParse{|*|} d


mkEditForm :: (InIDataId d) → HStIO d | iData d
:: HStIO d :== *HSt → (Form d,*HSt)
```

The function `mkEditForm` uses four generic cornerstone functions that are collected in the type class `iData`. The (`InIDataId d`) argument of `mkEditForm` describes the type and value of the iData element that is to be created:

```
:: InIDataId d :== (Init,FormId d)
:: Init          = Const | Init | Set
:: FormId     d  = { id::String, ival::d, lifespan::Lifespan, mode::Mode }
:: Lifespan      = Persistent^p | PersistentRO^r | Session^s | Page^n | Temp^t
:: Mode          = Edit | Display^d | NoForm^x
```

Here it suffices to state that it is a pair of an `Init` value that specifies the use of the `ival::d` value inside the (`FormId d`) record. The `lifespan` and `mode` fields control the lifespan and rendering mode of the iData element. An iData element can be stored persistently (`Persistent(RO)`) on the server side on disk, or locally in the page (`Session`, `Page`, `Temp`). Although the default mode of an iData element is `Edit`, it can also be used to display its state (`Display`), or even without any rendering at all (`NoForm`). For each of these variants, a `FormId` constructor function `{p,r,s,n,t}[d,x]FormId :: String d → FormId d` has been defined. `*HSt` is an opaque environment that contains the internal administration that is required to create HTML pages and form handling. It can be updated destructively, hence the *uniqueness type attribute* `*`. (Please consult [13] for details.)

When evaluated, `mkEditForm` basically performs the following actions: it first checks whether an earlier incarnation of the iData element (identified by the `id::String`[1] label) exists. If this is not the case, or the `Init` value is `Set`, then the `ival` value of the `FormId` argument is used. If it already existed, then it contains a possibly user-edited value. This value is used subsequently. Hence, the final iData element is up-to-date. This is recorded in the (`Form d`) record:

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

The `changed` field records the fact if the application user has edited the value of the iData element; the `value` is the up-to-date value, and `form` is the HTML rendering of this iData element that can be used within an arbitrary HTML page.

As an example, the following code snippet creates an iData element for `Int` values that, initially, looks as [42_____]:

---

[1] We are aware that the use of strings for form identification can be a source of (hard to locate) errors, but we have yet to find a better system of equal expressiveness.

```
♯ (intF,hst) = mkEditForm (Init,nFormId "My first iData!" 42) hst
```

If included in a web page, the application user can only create integer values with this iData element. A web application is any function that computes an HTML page, using an *HSt environment. Hence, its type is (HStIO Html). The wrapper function doHtmlServer transforms it into a real Clean interactive function:

```
doHtmlServer :: (HStIO Html) *World → *World
```

As an example, the following, complete code, creates a web application that allows users to edit integer values (Fig. 1(a)):

```
Start world = doHtmlServer tiny world
where tiny :: (HStIO Html)
      tiny hst
        ♯ (intF,hst) = mkEditForm (Init,nFormId "My first iData!" 42) hst
        = mkHtml "Simple Example" intF.form hst
```
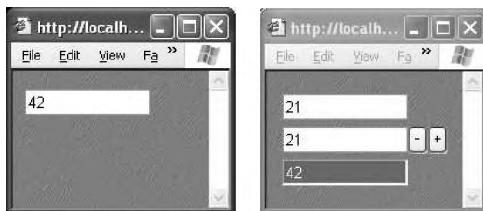

Fig. 1. (a) A single integer editor. (b) Display the sum of two integer input fields.

2.2 The iData Toolkit programming paradigm

The iData Toolkit programming paradigm advocates the use of pure data types and pure functions to model the UoD (Universe of Discourse) of the application that is to be constructed. From these types, the iData Toolkit derives the required forms automatically that can be used in the HTML pages of the application. The application programmer can specialize the derived GUI where needed, and in the end interconnect all iData elements that are relevant to his application. This amounts to the following four-step programming paradigm:

1. Model the UoD with pure data types and pure functions.
2. Derive iData from the data types generically.
3. Specialize iData where needed.
4. Define the logic of the application by interconnecting iData functionally.

Below, we illustrate the paradigm by constructing a small program that allows the application user to enter two integer values, and display their sum (Fig.1(b)). The same technique can be used to construct real-world applications such as a CD shop and a work administration [12].

1. Modelling the UoD In this step the application programmer *models* the entities and their relations by means of pure data types and pure functions over these data domains. For the sake of the example, the second integer editor is modeled distinctively as `IntCounter`, accompanied by two conversion functions:

```
::  IntCounter = IntCounter Int

instance toInt    IntCounter where toInt (IntCounter i) = i
instance fromInt IntCounter where fromInt i = IntCounter i
```

2. Deriving iData In this step the application programmer unleashes the generative power of the toolkit, and automatically derives instances for the four cornerstone functions of the toolkit. In the example the model types are either the basic `Int` type for which instances are already defined, or the integer counter type, that is specialized below.

3. Specializing iData In general, the created GUI of an iData element displays the structure of the type. For many types, this is sufficient. However, this is not always the case, and the generically derived GUI needs to be overruled by the application programmer. Overruling a generic recipe is known as *specialization*. Specialization is a delicate task, and hence the iData Toolkit provides a function that aids the application programmer with this:

```
specialize :: ((InIDataId a) → HStIO (Form a))
                 (InIDataId a) → HStIO (Form a) | gUpd{|*|} a
```

As an example, assume that there is a function

```
counterIData :: (InIDataId Int) → HStIO (Form Int)
```

that renders `Int` iData elements as [0 ___ - +] (in [12] we show how such a function is implemented). If we decide that from now on all `Int` iData elements should be rendered in this way, then this is enforced by:

```
gForm{|Int|} iDataId hst = specialize counterIData iDataId hst
```

In the example, however, we want to *model* the integer counter with the model type `IntCounter`. This amounts to calling `counterIData`, except that the `Int` values need to be converted to `IntCounter` values and vice versa. This is done with:

```
gForm{|IntCounter|} iDataId hst
  = specialize (coerceWith (toInt,fromInt) counterIData) iDataId hst
```

The `coerceWith` function is just a higher-order wrapper function that applies the conversion functions just before and immediately after the core function.

```
coerceWith :: (a → b,c → d) ((InIDataId b) → HStIO (Form c))
                               (InIDataId a) → HStIO (Form d)
coerceWith (f_ab,f_cd) f (init,formId=:{ival}) hst
  # ({changed,value,form},hst) = f (init,{formId & ival=f_ab ival}) hst
  = ({changed=changed,value=f_cd value,form=form},hst)
```

4. Interconnecting iData The final step is to *interconnect* iData elements. Interconnecting means that we define a functional dependency relation between the iData elements. The application programmer can exploit two important aspects of iData elements. First, the behavior of iData elements (discussed in Sect. 2.1) implies that they can be *shared*, i.e.: referring to the same iData element within the interconnection relation refers to the same iData element. In this way *cyclic dependency relationships* can be defined. Second, every iData element has a rendering that can be used subsequently arbitrarily many times, or even not at all. Each rendering refers to the same iData element. In the example, interconnecting the iData elements is straightforward:

```
Start world = doHtmlServer add world                                          1.
where add :: (HStIO Html)                                                     2.
      add hst                                                                  3.
        ♯ (i1F,hst) = mkEditForm (Init,nFormId "i1"   0)              hst 4.
        ♯ (i2F,hst) = mkEditForm (Init,nFormId "i2"   (IntCounter 0)) hst 5.
        ♯ (i3F,hst) = mkEditForm (Set, ndFormId "sum"                      6.
                                  (toInt i1F.value + toInt i2F.value)) hst 7.
        = mkHtml "Sum" [STable [] [i1F.form,i2F.form,i3F.form]]       hst 8.
```

The input elements are activated in lines 4–5. Their values are used by the sum display in line 7. Their forms are displayed in a single column in line 8. The resulting HTML page is displayed in Fig. 1(b).

3   A Conference Management System

In this section we show how to design a conference management system with the 'classic' iData Toolkit, its new tools, and the programming paradigm. The new tools are: **(i)** modeling *destructively shared data* with *reference types*; **(ii)** automatically *guard the consistency* of database and reference type values; **(iii)** automatic *version management*. Fig. 2(a) shows the initial screen of the system.

3.1   Modelling a conference management system

Examples of the logical entities of a conference management system are *members, accounts, papers,* and *discussions.* Appendix A.1 is a self-explanatory and self-contained subset of the collection of data types that have been defined. The types `PasswordBox`, `HtmlDate`, and `HtmlTime` are 'classic' iData Toolkit data types that have been specialized to model standard GUI elements. The types `Account` and `Login` are generally useful types for login handling. Examples of the entity-relations are determining the status of a member, determine the reviews of a programme committee member, and setting conflicts of interest. Appendix A.2 gives a subset of the functional relations between these entities.

   In the modeling step, the application programmer can use two of the above mentioned features **(i)** and **(ii)** of the 'non-classic' iData Toolkit.
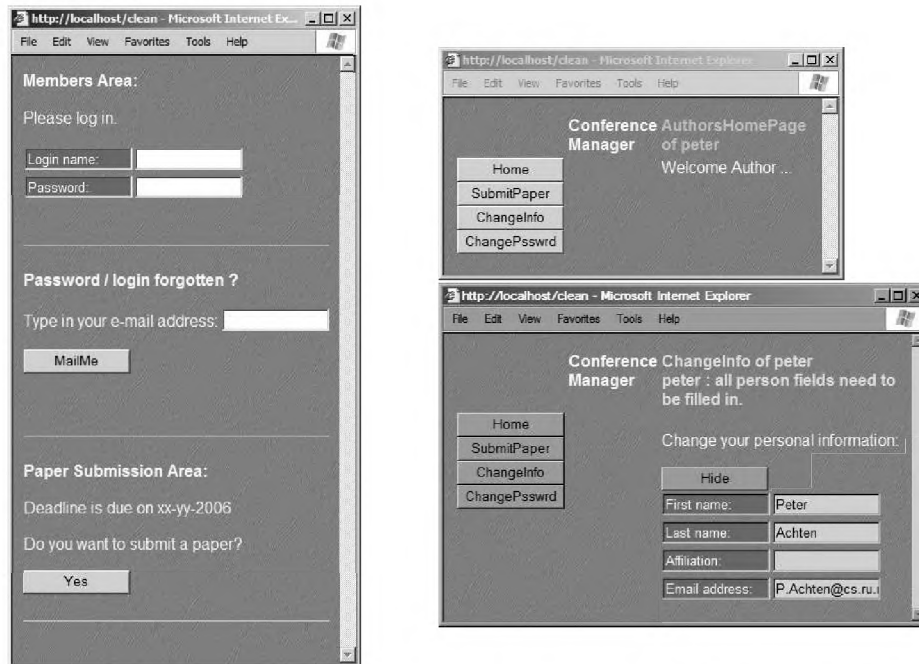
Fig. 2. (a) The initial system look. (b) The initial author page look. (c) An exception in the author page look when editing personal information.

Modelling destructively shared data In Sect. 1, we have argued that *destructive sharing* of entities is a natural phenomenon when modeling information systems. In case of the conference management system, the programmer wants to model the fact that *members*, *papers*, *reports*, and *discussions* are destructively shared. As a result, whenever the application user alters a destructively shared (sub)value in any iData element, then this (sub)value should be altered everywhere where it appears in a destructively shared context. Clearly, destructive sharing cannot be handled directly with pure data types in pure functional languages. For this purpose *reference types* have been introduced.

A reference type (`Ref2 d`) is a *phantom* type that creates a reference to a value of type `d`. Using the same reference value in a collection of data values results in a destructively shared occurrence of that value. (Sect. 4.2 discusses the implementation.) Briefly, a reference type 'connects' a type with an identifier:

```
:: Ref2 a = Ref2 String
```

Just as iData identifiers, this identifier is required to be unique. Hence, the application programmer needs to set up an additional name space for reference type identifiers. In the conference management system, this has been done by the function `setInvariantAccounts :: ConfAccounts → ConfAccounts` that traverses the complete administration for reference type occurrences, and assigns unique identifiers in such a way that the same entity obtains the desired destruc-

tive sharing structure. For reasons of space, we omit its code. It implements the following rules: persons are identified by their unique login name; refereed reports are identified by the identifier of the referee and their unique paper number; discussions by the author identifier and unique paper number; papers by author name and paper number.

Guarding consistency of iData In Sect. 1, we have argued that due to the edit-driven evaluation mechanism of the iData Toolkit, the *consistency* of the iData states cannot be guaranteed. We have included a mechanism to *judge* the consistency of destructively shared and persistent data. A `Judgement` is either `Ok` (`Nothing`), or raises an issue (`Just (id,issue)`) where `id` is the identifier of the judging entity, and `issue` a text that describes the issue. The value of an entity is committed to disk only if the corresponding judgement is `Ok`. Judgements can be rather syntactic. For instance, for `Person`, `Paper`, and `Login`, the judgements basically state that every field has to have a non-empty value:

```
invariantPerson :: String Person → Judgement
invariantPerson id {firstName,lastName,affiliation,emailAddress}
  | any ((==) "")  [firstName,lastName,affiliation,emailAddress]
             = Just (id,"all person fields need to be filled in.")
  | otherwise = Ok
```

A more challenging example of a judgement is given below.

```
invariantConfAccounts :: String ConfAccounts → Judgement             1.
invariantConfAccounts id accs                                        2.
| any ((≥) 0) papers  = Just (id,"paper number must be positive")    3.
| not (noDups papers) = Just (id,"paper number in use")              4.
| not uniqConflicts   = Just (id,"conflict already assigned to referee")  5.
| not uniqAssigns     = Just (id,"paper already assigned to referee")     6.
| conflicting         = Just (id,"assigned paper in conflict")       7.
| not (allMembers reports papers)                                    8.
                      = Just (id,"non-existing assigned paper")      9.
| not (allMembers conflicts papers)                                  10.
                      = Just (id,"non-existing assigned conflict")   11.
| otherwise           = Ok                                           12.
where                                                                13.
  papers       =            [nr  \\ (nr,_) ← getRefPapers    accs]   14.
  conflicts    = flatten    [nrs \\ (_,nrs) ← getConflicts    accs]  15.
  reports      = flatten    [nrs \\ (_,nrs) ← getAssignments accs]   16.
  uniqConflicts = and [noDups nrs \\ (_,nrs) ← getConflicts    accs] 17.
  uniqAssigns  = and [noDups nrs \\ (_,nrs) ← getAssignments accs]   18.
  conflicting  = or  [isAnyMember cNrs aNrs                          19.
                      \\ (_,cNrs,aNrs) ← getConflictsAssign accs]    20.
```

This is a judgement over the complete content of the conference management system's database. Paper numbers should be positive (line 3) and uniquely identify a paper (line 4). The list of conflicts and assigned papers should contain no duplicates (lines 5–6). Referees should not review papers for which they have a

conflicting interest (line 7). Finally, the set of reports and conflicts should be a subset of the set of papers (line 8 and 10).

Judgements are connected with reference type values and database values by the following two new functions that have been built on top of the iData Toolkit:

```
universalRefEditor
            :: (InIDataId (Ref2 a))    (a → Judgement) → HStIO (Form a)
                                                                    | iData a
universalDB :: (Init,a,String) (String a → Judgement) → HStIO a | iData a
```

Their implementation is discussed in Sect. 4. Applications of universalRefEditor are all alike, and proceed as in the case of persons:

```
editorRefPerson :: (InIDataId RefPerson) → HStIO (Form Person)
editorRefPerson (init,formid=:{ival=RefPerson refp=:(Ref2 name)})
  = universalRefEditor (init,{formid & ival=refp}) (invariantPerson name)
```

As an example of universalDB, we create the main conference database:

```
AccountsDB :: Init ConfAccounts → HStIO ConfAccounts
AccountsDB init accounts
  = universalDB (init,setInvariantAccounts accounts,uniqueDBname)
                invariantConfAccounts
```

3.2 Deriving iData

We can be very brief about this step, as this simply involves enumerating all instances to derive of almost all types for the generic cornerstone functions gForm, gUpd, gPrint, and gParse. The exceptions are that gForm needs to be specialized for the four reference types, the four custom types Reports, Conflicts, Co_authors, and Discussion, and the standard list type (display only the elements, not the list data constructors). In total, derivation concerns 27 data types, hence there are 99 derived instances and 9 specialized gForm instances.

3.3 Specializing iData

Reference types are specialized in boilerplate style, as illustrated with RefPerson (editorRefPerson is given above; invokeRefEditor is discussed in Sect. 4):

```
gForm{|RefPerson|} iDataId hst
  = specialize (invokeRefEditor editorRefPerson) iDataId hst
```

The four model types that need to be specialized are Reports, Conflicts, Co_authors, and Discussion. The first three are all basically list structures, but the application designer wants to display them in a column. They all proceed as given here for the case of Co_authors:

```
gForm{|Co_authors|} inIDataId hst
  = specialize (coerceWith (toList,fromList) vertlistFormButs) inIDataId hst
where toList (Co_authors authors) = authors
      fromList authors = Co_authors authors
```

Discussions are displayed in a table:

```
gForm{Discussion} inIDataId hst = specialize discussion inIDataId hst
where
    discussion (init,formid=:{ival=Discussion d}) hst
    = ({changed=False,form=flatten (map htmlOf d),value=formid.ival},hst)
    where
      htmlOf {messageFrom,date,time,message}
        = [ mkTable [ [ Txt "date: ", toHtml date, Txt "time: ", toHtml time]
                    , [ Txt "from: ", B [] messageFrom ] ]
            , Txt "message:", Txt message, Hr [] ]
```

3.4 Interconnecting iData

The conference management system basically proceeds along the following steps: it reads in the current accounts database, and then attempts to establish the identity of the application user. If this is a known user, then the application needs to present the current application page that the application user was visiting. This is determined by means of a *conference portal*, that determines and produces the correct page. If the user is unknown, then he is a guest, and should attempt to login to the system.

As shown in Sect. 2, the main entry of every iData Toolkit application is a function of type (HStIO Html):

```
Start world              = doHtmlServer mainEntrance world            1.

mainEntrance :: (HStIO Html)                                          2.
mainEntrance hst                                                      3.
  # (body,hst)           = loginhandling hst                         4.
  = mkHtml "Conference Manager" body hst                             5.

loginhandling :: (HStIO [BodyTag])                                   6.
loginhandling hst                                                    7.
  # (accounts,hst)       = AccountsDB Init                           8.
                           [initManagerAccount initManagerLogin] hst  9.
  = case loginHandlingPage accounts hst of                          10.
      (Left account,hst) = doConfPortal account accounts hst        11.
      (Right body,  hst) = (body,hst)                               12.
```

The loginhandling function checks the current user account. In order to do so, first the main accounts database needs to be accessed (lines 8–9). This is done with the function AccountsDB that was presented in Sect. 3.1. Initially, the accounts database contains a single entry for the conference manager. Later on, it contains all current member accounts. Hence, at this stage the application has the complete current accounts information. Second, the application needs to know the current user and his status (conference manager, program committee member, author, or guest) in order to generate the correct HTML page. The function loginHandlingPage :: ConfAccounts → HStIO (Either ConfAccount [BodyTag]) either yields the valid account of the current user or the HTML code body of the

login page that is displayed in Fig. 2(a). For conciseness, we omit its code. In case of an unknown user, body is displayed (line 12); otherwise the application uses account to switch to the proper page (line 11) using the function doConfPortal:

```
doConfPortal :: ConfAccount ConfAccounts → HStIO [BodyTag]            1.
doConfPortal account accounts hst                                     2.
  ♯ (navButtons,hst) = navigationButtons account.state hst            3.
  ♯ (currPage,  hst) = currPageStore     (homePage account.state)     4.
                                          navButtons.value hst         5.
  ♯ (navBody,   hst) = handleCurrPage currPage.value account accounts hst  6.
  ♯ (exception, hst) = eStore id hst                                   7.
  = ( [ mkSTable2 [[EmptyBody,B [] "Conference" <.||.> B [] "Manager " 8.
                   ,oops exception currPage.value]                     9.
                  ,[mkColForm navButtons.form,EmptyBody,BodyTag navBody] 10.
                  ]                                                     11.
      ], hst )                                                         12.
```

This function creates a page that consists of four areas (lines 8–12): **1.** a set of navigation buttons (navButtons.form) that depend dynamically on the user status; **2.** the static "Conference Manager" title; **3.** the user status and page in case of no issues, and the issue otherwise (line 9); **4.** the actual page content that the user is visiting (navBody). Fig. 2(b) shows the initial look in case of an author; Fig. 2(c) shows a failing judgement (empty entry in the person data). The navigation buttons are created by navigationButtons simply by enumerating the buttons, that, when pressed, yield the corresponding page that should be displayed. If no button is pressed, then its function result is the identity function. Here we only show the code for the author case:

```
navigationButtons :: Member → HStIO (Form (CurrPage → CurrPage))
navigationButtons member
  = ListFuncBut (Init, sFormId "navigation" (navButtons member))
where navButtons :: Member → [(Button,a → CurrPage)]
      navButtons (Authors _)
        = [ (LButton defpixel "Home",       const AuthorsHomePage)
          , (LButton defpixel "SubmitPaper", const SubmitPaper)
          , (LButton defpixel "ChangeInfo",  const ChangeInfo)
          , (LButton defpixel "ChangePsswrd",const ChangePassword) ]
      navButtons ...
```

CurrPage enumerates the possible pages that can be visited:

```
:: CurrPage = RootHomePage    | AssignPapers | ModifyStates   // root pages
            | AuthorsHomePage | SubmitPaper                   // authors
            | ChangePassword  | ChangeInfo                    // common
            | ListPapers      | ListReports  | DiscussPapers  // referees + root
                              | ShowPapersStatus | RefereeForm
            | RefereeHomePage                                 // referees
            | GuestHomePage                                   // guests
```

The current page is stored in an iData element. The function currPageStore uses the iData Toolkit library function mkStoreForm for this purpose, which extends

the library function `mkEditForm` with a function argument that is applied to the current value. Hence, when combined with the function result of the navigation buttons, this is the page that should be displayed.

```
currPageStore :: CurrPage → (CurrPage → CurrPage) → HStIO (Form CurrPage)
currPageStore currpage = mkStoreForm (Init, sFormId "cf_currPage" currpage)
```

The functions `homePage` and `handleCurrPage` enumerate the default starting pages and page content creation functions:

```
homePage :: Member → CurrPage
homePage (ConfManager _)    = RootHomePage
homePage (Referee    _)     = RefereeHomePage
homePage (Authors    _)     = AuthorsHomePage
homePage (Guest      _)     = GuestHomePage


handleCurrPage :: CurrPage → ConfAccount → ConfAccounts → HStIO [BodyTag]
handleCurrPage RootHomePage  = rootHomePage

                                  :

handleCurrPage ChangeInfo    = changeInfo
```

These functions define the final content of the HTML pages. As an example, here is the function that computes the page that is displayed in Fig. 2(c) in which members can modify their personal information:

```
changeInfo :: ConfAccount ConfAccounts → HStIO [BodyTag]
changeInfo {state} _ hst
  ♯ ({form},hst) = mkEditForm (Init,nFormId "sh_changeInfo"
                                     (fromJust (getRefPerson state))) hst
  = ([Br, Txt "Change your personal information:", Br, Br] ++ form,hst)
```

As this example demonstrates, some of these page generating functions are very short, and basically use one iData element; others can be rather extensive. The 14 page generating functions consume 232 lines of code, three of which consume the largest part: `guestHomePage` (55 loc), `assignPapersConflictsPage` (52 loc), and `discussPapersPage` (44 loc). Hence, this amounts to an average of 6–7 loc for the remaining 11 functions.

3.5 Summary

In the above sections we have given an impression of working with the iData Toolkit and its programming paradigm. We would like to emphasize the fact that the data types and functions that are created in the modeling step of the programming paradigm belong to the programming repertoire of any novice functional programmer. We also note that the relatively largest programming effort is in the interconnection step of the paradigm. The application logic is guided using local stores of application state to determine the proper status of the application. The current version of the conference management system consumes 945 loc.

4 Implementation

In this section we present the implementation of the new iData Toolkit tools. In Sect. 3 we enumerated them as **(i)** *reference types*; **(ii)** *guarding consistency*; **(iii)** *version management*. We first explain how to implement **(ii)** and **(iii)** for database values in Sect. 4.1. Due to their complexity, *reference types* earn a separate discussion in Sect. 4.2.

4.1 Universal database

In Sect. 2, we have seen that an iData can store its value persistently by using the `Persistent(RO)` Lifespan value. This implies that we can readily use iData elements as primitive databases:

```
universalDB₁ :: (Init,a,String) → HStIO a | iData a                1.
universalDB₁ (init,v,file) hst                                      2.
  ♯ (dbf,hst)           = mkEditForm (init,pxFormId file v) hst     3.
  = (dbf.value,hst)                                                 4.
```

It is rather easy to add version handling to this scheme:

```
universalDB₂ :: (Init,a,String) → HStIO a | iData a                 1.
universalDB₂ (init,v,file) hst                                      2.
  ♯ (dbf,hst)           = mkEditForm (Init,rxFormId file (0,v)) hst 3.
  ♯ (dbversion,dbvalue)= dbf.value                                  4.
  ♯ (versionf,hst)      = versionNr Init dbversion hst              5.
  | init == Init || dbversion ≠ versionf.value                      6.
                        = (dbvalue,snd (versionNr Set dbversion hst)) 7.
  ♯ (versionf,hst)      = versionNr Set (dbversion+1) hst           8.
  = (v,snd (mkEditForm (Set, pxFormId file (versionf.value,v)) hst)) 9.
where                                                               10.
  versionNr init c = mkEditForm (init,txFormId ("vrs_db_"+++file) c) 11.
```

Instead of storing a single value, we store a pair of the version number and the value (line 3). In addition, we maintain a version counter per database that keeps track of the correct version. The version counter is accessible by all applications that refer to this database. This storage is defined in line 11, and read in line 5. If we are only reading the database, or in case of a conflicting version (line 6), we always adhere to the database version and store its version number (line 7). In any other case, we increase the version number, and store it in both the global version counter (line 8) and the database (line 9).

The final addition is consistency handling:

```
universalDB₃ :: (Init,a,String) (String a → Judgement) → HStIO a   1.
              | iData a                                             2.
universalDB₃ (init,v,file) invariant hst                           3.
  ♯ (dbf,hst)      = mkEditForm (Init,rxFormId file (0,v)) hst     4.
  ♯ (dbversion,dbvalue)                                            5.
                   = dbf.value                                     6.
  ♯ (versionf,hst) = versionNr Init dbversion hst                 7.
```

```
| init == Init                                                              8.
  = (dbvalue,snd (versionNr Set dbversion hst))                            9.
| dbversion ≠ versionf.value                                              10.
  ♯ (_,hst)      = versionNr Set dbversion hst                            11.
  ♯ (_,hst)      = eStore ((+) (Just (file,"Screen out of date."))) hst  12.
  = (dbvalue,hst)                                                         13.
♯ exception       = invariant file v                                     14.
| isJust exception                                                       15.
  = (v,snd (eStore ((+) exception) hst))                                 16.
♯ (versionf,hst) = versionNr Set (dbversion+1) hst                       17.
= (v,snd (mkEditForm (Set, pxFormId file (versionf.value,v)) hst))       18.
where                                                                    19.
  versionNr init c = mkEditForm (init,txFormId ("vrs_db_"+++file) c)     20.
```

The second argument in line 1 and 3 of universalDB is the consistency check of the database data. It is checked in line 14 just before updating the database. If it raises an exception (line 15), then the new value is not stored in the database, but instead the exception is passed on to a global exception store (line 16). This exception store is again a simple storage iData element:

```
eStore :: (Judgement → Judgement) → HStIO Judgement
eStore f hst
  ♯ ({value},hst) = mkStoreForm (Init,{txFormId "handle_exception" Ok}) f hst
  = ( value, hst)
```

Exceptions are also stored here in case of conflicting version numbers (line 10–13). Any exception thus raised is reported to the application user as explained in Sect. 3.4 (doConfPortal).

4.2   Reference types

Reference types are used by application programmers to model *destructively shared data*. Recall that a reference type is defined as :: Ref2 a = Ref2 String. Suppose we want to destructively share values of some type A. In Sections 3.1 and 3.4 we have shown what needs to be programmed. Recapitulating:

```
:: RefA = RefA (Ref2 A)                                                   1.

gForm{|RefA|} id hst = specialize (invokeRefEditor editorRefA) id hst     2.

editorRefA :: (InIDataId RefA) → HStIO (Form A)                           3.
editorRefA (init,formid=:{ival=(RefA ref=:(Ref2 name))})                  4.
  = universalRefEditor (init,{formid & ival=ref}) (invariantA name)       5.
```

The new type RefA serves as a reference to A values (line 1). For each RefA value within a model data type, the user wants an iData element of A values. Clearly, this requires specialization (line 2). The new library function universalRefEditor handles this 'dereferencing'. It is provided with the appropriate consistency checking function invariantA (line 3–5).

The function invokeRefEditor evaluates its higher-order argument, and substitutes the given value parameter in the resulting iData:

```
invokeRefEditor :: ((InIDataId b) → HStIO (Form d))
                    (InIDataId b) → HStIO (Form b)
invokeRefEditor f (init,{ival}) = coerceWith (id,const ival) f
```

The function universalRefEditor puts everything together.

```
universalRefEditor :: (InIDataId (Ref2 a)) → HStIO (Form a) | iData a        1.
universalRefEditor (init,ref2Id={ival=Ref2 filename}) hst                    2.
  ♯ ({value},hst) = mkEditForm (Init,databaseId createDefault) hst           3.
  ♯ (valueF, hst) = mkEditForm (Init,copyId     value)         hst           4.
  ♯ (_,      hst) = mkEditForm (Set, databaseId valueF.value)  hst           5.
  = ({valueF & changed = True},hst)                                          6.
where                                                                        7.
  databaseId v   = {pxFormId "" v  &  id = filename}                         8.
  copyId     v   = {ref2Id & ival = v,id = "copy_r_"+++filename}             9.
```

The clue of the implementation is that a *persistent* iData element databaseId
is created that is identified by the reference label filename (line 8). For this
iData a default value is generated using the function createDefault (line 3). The
application user never edits this iData, but instead is offered an iData on a copy
iData, that is identified by copyId (line 9). The altered value is written back to the
database (line 5), and the altered user iData is returned by universalRefEditor
(line 6).

   We can extend universalRefEditor with version and consistency handling as
described in Sect. 4.1. For reasons of space, we omit these steps.

4.3   Summary

We have shown how the new tools can be implemented on top of the 'classic'
iData Toolkit. We use existing toolkit capabilities: elementary storages that can
be destructively shared, specialization, and the flexibility of name space man-
agement. It allows us to manipulate persistent storage in a way that cannot be
done directly in a functional language without special structures such as heaps
or mutable variables. The toolkit itself has not been changed, but applications
can now use the new tools for their purposes.

5   Related Work

In the realm of functional programming, many solutions have been proposed
to program web applications. We mention just a few of them in a number of
languages: the HaskellCGI library by Meijer [10]; the Curry approach by Hanus
[6] (the *CurryWeb* application [7] shares a number of application concerns as
the conference management system described in this paper); writing XML appli-
cations [5] by Elsman and Larsen in *SMLserver* [4]. One sophisticated system
is Thiemann's WASH/CGI [14], based on Haskell. Here, HTML is produced as
an effect of the CGI monad whereas we consider HTML as first-class citizens,
using data types. Instead of storing state, WASH/CGI logs all user responses

and I/O operations. These are replayed when needed to bring the application to its desired, most recent state. Forms are programmed explicitly in HTML, and their elements may, or may not, contain values. In the iData Toolkit, forms are generated from arbitrary data types, and always have value. Interconnecting forms in WASH/CGI is done by adding callback actions to submit fields, whereas the iData Toolkit uses a functional dependency relation. The above systems have proven to be highly inspiring. Our contribution is the identification of a single versatile unit, the iData element, that provides an integrated handling of all of their concerns while maintaining a high level of abstraction. In addition, we have shown that the programming paradigm advocates "classic" style functional programming.

The popular framework Rails [15] is based on the object oriented programming language Ruby [8]. With Rails, database front-end applications can be quickly developed. The application programmer is provided with scripts to configure directories and initial class files. The database tabling structure is used as a data structure specification language. A Rails application is a set of classes structured as a classic model-controller-view [9] application. Browser-server communication is based on urls that adhere to the configured directory structure, and the Ruby controller classes that are supposed to reside there. Server-database communication is realized by the model classes. These reflect on the database table structure, and generate the appropriate methods for the class scripts. Views are created via HTML templates that contain Ruby code to manipulate its content. This is similar to WASH/CGI in which HTML code is defined as an effect of the CGI monad. Rails shares with the iData Toolkit the goal of generating as much as possible from data structures. In Rails these are limited to table structures of basic types. The iData Toolkit can handle arbitrary, recursive, higher-order data structures. Rails applications are extremely vulnerable to configuration changes, in contrast with iData Toolkit applications. These are single executables that maintain their own state. (One can even remove all persistent files on-the-fly. Any iData Toolkit application recreates them in their initial state.)

6   Conclusions

In this paper we have presented the programming paradigm of the iData Toolkit. This four-step paradigm advocates the use of traditional, well-known, functional programming techniques to model information systems, and uses the generative power of the toolkit to automatically create interactive applications from these models. When modeling information systems, programmers need tools to model destructively shared data structures, deal with versions in a transparent way, and guard the consistency of the data. We have shown how these tools can be added on top of the 'classic' iData Toolkit, thus demonstrating its expressive power. As a representative example, we have developed a prototype conference management system, using the programming paradigm. The system is a single, compact (1kloc), application.

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. M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.
5. 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.
6. 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.
7. M. Hanus and F. Huch. An Open System to Support Web-based Learning. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, Valencia (Spain), 2003.
8. A. Hunt and D. Thomas. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Professional, 1st edition, Oct. 2000.
9. 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.
10. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
11. R. Plasmeijer and P. Achten. Generic Editors for the World Wide Web. In *Central-European Functional Programming School*, Eötvös Loránd University, Budapest, Hungary, Jul 4-16 2005.
12. R. Plasmeijer and P. Achten. iData For The World Wide Web - Programming Interconnected Web Forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *LNCS*, Fuji Susono, Japan, Apr 24-26 2006. Springer Verlag.
13. 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 - Revised Selected Papers, 17th International Workshop, IFL05*, LNCS 4015, Department of Computer Science, Trinity College, University of Dublin, September 19-21 2006.
14. 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.
15. D. Thomas and D. Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, 1st edition, Aug. 2005.

A   Appendix

A.1   A sample of the UoD model types specified as pure data types

```
:: ConfAccounts    :== [ConfAccount]
:: ConfAccount     :== Account Member
:: Account s        = { login           :: Login,   state    :: s            }
:: Login            = { loginName        :: String, password :: PasswordBox }
:: Member           = ConfManager       ManagerInfo | Authors   PaperInfo
                    | Referee           RefereeInfo | Guest     Person
:: ManagerInfo      = { person           :: RefPerson                        }
:: PaperInfo        = { person           :: RefPerson,       nr     :: PaperNr
                    , discussion        :: RefDiscussion, paper :: RefPaper
                    , status            :: PaperStatus                     }
:: PaperNr         :== Int
:: PaperStatus      = Accepted | CondAccepted | Rejected | Submitted
                    | UnderDiscussion DiscussionStatus
:: DiscussionStatus = ProposeAccept | ProposeCondAccept | ProposeReject
                    | DoDiscuss
:: RefereeInfo      = { person           :: RefPerson, reports    :: Reports
                    , conflicts         :: Conflicts                      }
:: Reports          = Reports           [(PaperNr, RefReport)]
:: Conflicts        = Conflicts         [PaperNr]
:: Person           = { firstName        :: String, lastName      :: String
                    , affiliation       :: String, emailAddress :: String  }
:: Discussion       = Discussion [Message]
:: Message          = { messageFrom      :: String, date :: HtmlDate
                    , message           :: String, time :: HtmlTime        }
:: Paper            = { title            :: String, first_author :: Person
                    , abstract          :: String, co_authors    :: Co_authors
                    , pdf               :: String                         }
:: Co_authors       = Co_authors [Person]
:: Report           = { recommendation:: Recommendation
                    , familiarity      :: Familiarity                      }
:: Recommendation   = StrongAccept | Accept | WeakAccept | Discuss
                    | StrongReject | Reject | WeakReject
:: Familiarity      = Expert | Knowledgeable | Low
:: RefPerson        = RefPerson         (Ref2 Person)
:: RefPaper         = RefPaper          (Ref2 Paper)
:: RefReport        = RefReport         (Ref2 (Maybe Report))
:: RefDiscussion    = RefDiscussion     (Ref2 Discussion)
```

A.2   A sample of the entity-relations specified as functions

```
getRefPapers      :: ConfAccounts → [(PaperNr,RefPaper)]
getConflicts      :: ConfAccounts → [(RefPerson,[PaperNr])]
getAssignments    :: ConfAccounts → [(RefPerson,[PaperNr])]
getConflictsAssign :: ConfAccounts → [(RefPerson,[PaperNr],[PaperNr])]
```