iTasks: Executable Specifications of Interactive Work Flow Systems for the Web

Rinus Plasmeijer Peter Achten Pieter Koopman

Software Technology Department, Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525ED Nijmegen, Netherlands

{rinus,P.Achten,pieter}@cs.ru.nl

Abstract

In this paper we introduce the iTask system: a set of combinators to specify *work flows* in a pure functional language at a very high level of abstraction. Work flow systems are automated systems in which *tasks* are coordinated that have to be executed by humans and computers. The combinators that we propose support work flow patterns commonly found in commercial work flow systems. Compared with most of these commercial systems, the iTask system offers several advantages: tasks are statically typed, tasks can be higher order, the combinators are fully compositional, dynamic and recursive work flows can be specified, and last but not least, the specification is used to generate an executable web-based multi-user work flow application. With the iTask system, useful work flows can be defined which cannot be expressed in other systems: work can be interrupted *and* subsequently directed to other workers for further processing.

The implementation is special as well. It is based on the Clean iData toolkit which makes it possible to create fully dynamic, interactive, thin client web applications. Thanks to the generic programming techniques used in the iData toolkit, the programming effort is reduced significantly: state handling, form rendering, user interaction, and storage management is handled automatically. The iTask system allows a task to be regarded as a special kind of persistent redex being reduced by the application user via task completion. The combinators control the order in which these redexes are made available to the application user. The system rewrites the persistent task redexes in a similar way as functions are rewritten in lazy functional languages.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language ClassificationsApplicative (functional) languages; H.4.1 [*Information Systems Applications*]: Office Automation—Workflow management; H.5.3 [*Information Interfaces And Presentation*]: Group and Organization Interfaces—Web-based interaction

General Terms Design; Languages

Keywords Clean; iData; iTask

ICFP'07 October 1–3, 2007, Freiburg, Germany

Reprinted from ICFP'07, Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, October 1–3, 2007, Freiburg, Germany, pp. 141–152.

1. Introduction

The iData toolkit (Plasmeijer and Achten 2006a,b) is a high level library for creating interactive, thin client, web applications. It is a domain specific language embedded in the pure, lazy functional programming language Clean. In order to validate the expressiveness of the toolkit, a number of non-trivial web applications have been developed, such as a web shop, a project administration system (Plasmeijer and Achten 2006b), and a conference management system (Plasmeijer and Achten 2006c). Based on these case studies, we observe that the iData toolkit is well suited to create complex GUI forms, which can be used to create and change values of complex data types. However, the iData toolkit does not provide special support for the specification of complex control flows. An iData web application runs on a server and is started from scratch each time a page is demanded from a client. To realize a control flow, the application programmer needs to keep track of the current application state by means of data storages. This can lead to programs that are difficult to comprehend and maintain, and it does not scale well.

In order to include control flow to the iData toolkit, we have been inspired by contemporary *work flow* systems. Work flow systems are automated systems in which work flow situations are specified (typically in a graphical way) that need to be executed by humans and computers. Current commercial work flow systems (such as Business Process Manager, COSA Workflow, FLOWer, i-Flow 6.0, Staffware, Websphere MQ Workflow, and YAWL) mainly deal with control flow rather than data. Hence, it is not cut and clear whether the data flow oriented approach of the iData toolkit is suitable to support work flows.

In this paper we present the iTask system. It is a combinator library for the specification of interactive multi-user web-based *work flows*. It is built on top of the iData toolkit, and both can be used within the same program. It covers all known *work flow patterns* that are found in contemporary commercial work flow tools (van der Aalst et al. 2002), and is thus suited to describe real-world applications. Moreover, we introduce a number of novel combinators to specify dynamic, higher-order, and recursive work flows. The iTasks system makes good use of the functional host language, and extend these patterns with strong typing, higher-order functions, lazy evaluation, and a monadic style of programming. Its foundation upon the generic (Hinze 2000; Alimarine 2005) features of the iData toolkit yields compact, robust, reusable and understandable code. Work flows are defined on a very high level of abstraction. It truly is an executable specification, as much is done and generated automatically. This requires a significant amount of explanation before we show the expressive power in the examples.

We start with a short overview of the iData toolkit in Sect. 2. By making use of generic programming techniques, web forms (iData

elements or editors) are generated and handled automatically for the used types. Arbitrarily complex dependencies between iData elements can be defined in a functional style. Such editors behave in a similar way as cells in a spreadsheet: making a change in one of the editors automatically affects the contents of every editor that depends on its value. In Sect. 3 we explain why this standard behavior of an editor is not suitable to express control flows conveniently and we show how this is solved in a suitable way: data editors become task editors. The concepts and their implementation are explained in a stripped down version of the iTask system: the iTask Core System. In Sect. 4 we illustrate the power of the full system by giving a representative set of work flow examples. In Sect. 5 we reveal the most interesting aspects of the real implementation of the iTask system. We end with the discussion of related work in Sect. 6 and conclusions in Sect. 7.

2. The iData Toolkit

In this section we present the 'classic' iData toolkit. With this iData toolkit one can create dynamic interactive web applications. These web applications are geared towards the manipulation of data, typical examples are calculators and web shops. The web application at the server side performs all essential work, the browser at the client side just displays the HTML-code produced by the web application and passes the user input to the web application. This distribution of work is called a *thin client* architecture. 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 her to edit values of that given type only. This editor can be derived automatically from the (recursive) type using the generic programming facilities of Clean. If the generic behavior is not the desired one, the programmer can define a tailor made form for all values of a specific type or even for individual values. In this paper, we use a few of the available toolkit functions to create iData elements.

An iData element is created with a function of synonym type `HStIO` that transforms an opaque environment of type `*HSt` into a tuple containing its *form* and the new `*HSt`:

```
:: HStIO d :== *HSt → (Form d,*HSt)
```

A `Form` contains state information about the iData element as well as the associated HTML code for the form corresponding to the iData element. `*HSt` is an opaque environment that contains the internal administration that is required for creating HTML pages and handling forms. It can be updated destructively, hence the *uniqueness type attribute* `*`. Please consult (Plasmeijer and Achten 2006a) for details of `HSt`. Passing a unique `*HSt` around between iData elements orders the manipulations of the iData elements. This effect is similar to using the `IO` monad in Haskell, but uniquely attributed states are passed around explicitly.

The overloaded function `mkIData` creates an iData element. `mkIData` is an explicit `HSt` environment transformer function. Its signature is (in Clean, type classes are enumerated at the *end* of a type signature, after the `|` symbol):

```
mkIData :: (InIDataId d) → HStIO d | iData d

class iData d           | gForm {|*|}, iCreateAndPrint, gParse{|*|}
                        , gerda {|*|}, TC   d
class iCreateAndPrint d | iCreate, iPrint d
class iCreate d         | gUpd  {|*|}       d
class iPrint d          | gPrint{|*|}       d
```

`mkIData` uses generic functions (in this case of kind $\star$) to create iData elements (do not confuse this kind with the uniqueness attribute). These are collected in the type class `iData` which gathers the six generic cornerstone functions of the iData toolkit. They can be used for values of *any* type to automatically create an HTML form (`gForm`), to handle the effect of any edit action with the browser including the creation of default values (`gUpd`), to print or serialize any value (`gPrint`), to parse or de-serialize any value (`gParse`), to store, retrieve or update any value in a relational database (`gerda`), or to serialize and de-serialize values and functions in a `Dynamic` (using the compiler generated `TC` class).

`mkIData` is applied to an `(InIDataId d)` argument that describes the type and value of the iData element that is to be created:

```
:: InIDataId d :== (Init, FormId d)
:: Init        = Const | Init | Set

mkFormId       :: String d → FormId d
```

The function `mkFormId` creates a default `(FormId d)` value, given a unique identifier string[1] and the value of the iData element. Note that in Clean the arity of functions is denoted explicitly by white-space between the arguments, hence the arity of `mkFormId` is two. The `Init` value describes the use of that value: it is either a `Const`ant or it can be edited by the user. In case of `Init`, it concerns the initial value of the editor. Finally, it can be `Set` to a new value by the program. A `(FormId d)` value is a record that identifies and describes the *use* of the iData element:

```
:: FormId    d  = { id :: String, ival :: d
                  , lifespan :: Lifespan, mode :: Mode }
:: Lifespan     = Database | Persistent | PersistentRO
                | Session  | Page | Temp
:: Mode         = Edit | Submit | Display  | NoForm
```

The `lifespan` field controls the storage of the value of the iData element: it can be stored persistently on the server side on disk in a relational database (`Database`) or in a file (`Persistent` with `RO` read-only), it can be stored locally at the client site in the web page (`Session`, `Page` (default)), or one can decide not to store it at all (`Temp`). Storage and retrieval of data is done automatically by the system. The `mode` field controls the *rendering* of the iData value: by default it can be `Edit`ed which means that every change made in the form is communicated to the server, one can choose for the more traditional handling of forms where local changes can be made that are all communicated when the `Submit` button is pressed, but it can also be `Display`ed as a constant, or it is not rendered at all (`NoForm`).

To facilitate the creation of non-default `(FormId d)` values, the following straightforward type classes have been defined:

```
class    (<@) infixl 4 att :: (FormId d) att → FormId d
class    (@>) infixr 4 att :: att (FormId d) → FormId d
instance <@ String, Lifespan, Mode
instance @> String, Lifespan, Mode
```

For instance, `(mkFormId "answer!" 42 <@ Display)` describes an identifier for an iData element of type `Int` that has label `"answer!"`, value 42, and cannot be edited by the user.

When evaluated, `(mkIData (init, iDataId))` basically performs the following actions: it first checks whether an earlier incarnation of the iData element (identified by `iDataId.id`, i.e. the name of the iData element) exists. If this is not the case, or `init` equals `Set`, then `iDataId.ival` is used as the current value of the iData element. If it already existed, then it contains a possibly user-edited value, which is used subsequently. Hence, the final iData element is always up-to-date. This is recorded in the `(Form d)` record:

[1] The use of strings for form identification can be a source of (hard to locate) errors in the iData system. The iTask system eliminates these issues by an automated systematic identification system.

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

The changed field records the fact whether the application user has previously edited the value of the iData element; the value is the up-to-date value; form is the HTML rendering of this iData element that can be used within an arbitrary HTML page. BodyTag is an algebraic data type that maps one-to-one to the HTML-grammar.

As an example, consider the following function:

```
iDataF label = mkIData (Init, mkFormId label default)
```

The function default uses the generic gUpd function to create a value of the desired type:

```
default :: d | iCreate d
```

This generic default function plays a significant role in both the iData and iTasks toolkits: it allows us to always create a value of the correct type. The type of the context in which iDataF is applied determines what value is created and then displayed. Just by giving it type String→HStIO Int we obtain an Int editor, and by giving it some other type, such as String→HStIO Person with:

```
::  Person = { firstName   :: String,  surname :: String
             , dateOfBirth :: HtmlDate, gender :: Gender }
::  Gender = Male | Female
```

we obtain a Person editor. See Fig. 1 for screen shots.

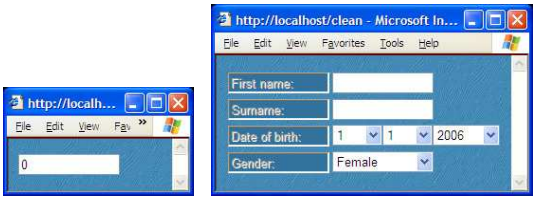

Figure 1. (a) An Int editor. **(b)** A Person editor.

A web application is any function that computes an HTML page, using an *HSt environment. Hence, its type is *HSt → (Html, *HSt). The function mkHtml, when given a string (to name a page) and HTML code (the body of the page), is such a function. The wrapper function doHtmlServer transforms it into an interactive program.

```
doHtmlServer :: (*HSt → (Html,*HSt)) *World → *World
mkHtml       :: String [BodyTag] *HSt → (Html,*HSt)
```

As an example, we show the complete code of a web application that allows users to edit integer values (see Fig. 2**(a)** for a screen shot). In Clean, ♯ is a non-recursive let-definition which scope extends downwards.

```
module singleIntegerEditor
import StdEnv, StdHtml

Start world = doHtmlServer tiny world

tiny :: *HSt → (Html,*HSt)
tiny hst
♯ (intF,hst) = mkIData (Init,mkFormId "answer!" 42) hst
= mkHtml "Simple Example" intF.form hst
```

Notice that whenever the user commits a change in one of the forms on a page with iData, the information is sent to the server which then starts the corresponding Clean application from scratch. The application automatically recovers the (possibly persistent) values of all iData elements, including those that may have been edited by the user. In the tiny application, the effect is that the previous change made by the user is noticed and shown. This is not exciting, but by connecting iData elements, we can make interesting applications.

Figure 2. (a) A single integer editor. **(b)** Display the sum of two integer input fields.

2.2 Interconnecting iData

An interactive web application programmed with iData usually consists of a collection of *interconnected* iData elements. Interconnecting means that the value of iData elements may depend on the value of others. We express this dependency as a function. 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.: multiple occurrences of the same iData identifier within the interconnection relation all refer to the same iData element. Second, the rendering of an iData element (the .form field of its Form record) is perfectly separated from its behavior. The rendering can be used arbitrarily many times, or even not at all. Each rendering refers to the same iData element. We exploit these features in the iTasks toolkit in the next sections.

As stated at the end of Sect. 2.1, the web application is restarted from scratch every time a user event is posted to it from the client side. The interconnection relation determines dynamically which iData elements recover their, possibly edited, states and also determines what HTML output should be returned to the client side. The ability to recover all of its states is a crucial feature of the iData system, because a web application is restarted on each event and hence has no notion of the previous state at that moment.

As an example of interconnecting iData elements, consider a program that creates two integer edit boxes and an integer display containing the sum of values of the two edit boxes (Fig. 2**(b)**):

```
Start world = doHtmlServer add world                                    1.

add :: *HSt → (Html,*HSt)                                               2.
add hst                                                                 3.
♯ (i1F,hst) = mkIData (Init,mkFormId "i1" 0)              hst           4.
♯ (i2F,hst) = mkIData (Init,mkFormId "i2" 0)              hst           5.
♯ (i3F,hst) = mkIData (Set, mkFormId "sum"                              6.
                      (i1F.value + i2F.value)<@Display) hst             7.
= mkHtml "Sum" [STable [] [i1F.form,i2F.form,i3F.form]] hst             8.
```

The two input elements are activated in lines 4–5. Their possibly edited values are used by the sum display in lines 6–7. Their forms are displayed in a single column in line 8.

Notice the declarative nature of interconnecting iData elements: one specifies two input boxes and the display of their sum in a third, and this is indeed what we get over and over again whenever a user changes one of the input values. In that sense, the behavior of an iData application can be compared with value changes made in a cell of a spreadsheet. The rendering and handling of input is done automatically. Notice that making of a HTML-table that contains the forms of the three iData elements is separated from the creation of the elements themselves.

3. Introduction to iTasks

The following iData exercise was given to us by Phil Wadler:

> "Suppose that you want two integer forms to appear *one after another* on the screen and *then* show the sum of them, how do you program this using iData?"

The key concept in the iData toolkit is that an interactive application is a collection of interconnected editors. From this point of view, the concept of a 'terminated' editor is not the standard behavior of an iData editor. Instead, the collection of editors stays alive after each edit operation, allowing the user to enter other data as is also common in a spreadsheet. The exercise above illustrates the need to specify the control flow between editors as well. This is technically possible since all editors are created dynamically. However, there is no specific support in the iData library to do this conveniently and in our case studies we have encountered similar situations in which control flows could be defined with iData elements, but in an ad-hoc way.

In this section we explain the principle of the iTasks toolkit. We first show in Sect. 3.1 how a standard iData editor can be changed into an iTask editor by extending it with a storage for its evaluation state and a confirmation button, which can be used by the user to confirm that the iData element is no longer required. This is defined entirely within the iData toolkit. With this technique, an ad-hoc solution to Wadler's exercise can be constructed. This solution, however, does not scale up with real programs. In Sect. 3.2 we show a solution to Wadler's exercise using an extended editor and a monadic combinator library in a way that does scale well.

3.1 Handling a Sequence of Forms in iData

We define a special function to make a `taskEditor`. It is an 'ordinary' editor extended with a `Boolean` iData state in which we record whether the editor task is finished. It is not up to an iData editor to decide whether a task is finished, but this is indicated by the user by pressing an additional button. Hence, a standard iData editor is extended with a button and a boolean storage. These elements are created by the library functions `simpleButton` and `mkStoreForm`:

```
simpleButton :: String String (d→d) →HStIO (d→d)
mkStoreForm  :: (InIDataId d) (d→d) →HStIO d | iData d
```

(`simpleButton` *l name f*) creates an iData element whose appearance is that of a push button with given *name*. It is identified with label *l*. When pressed (which is an edit operation by the user), its value is the function *f*, otherwise it is the identity function. (`mkStoreForm` *iD f*) creates an iData element that applies *f* to its current state.

With these two standard functions from the iData toolkit we can enhance any iData editor with a button and boolean storage:

```
taskEditor :: String String a *HSt → (Bool,a,[BodyTag],*HSt)          1.
           | iData a                                                  2.
taskEditor btnName label v hst                                        3.
# (btn,  hst) = simpleButton btnLabel btnName (const True)  hst       4.
# (done, hst) = mkStoreForm (Init,mkFormId storeLabel False)          5.
                                                      btn.value hst   6.
# (f, btnF)   = if done.value ((>@) Display,Br) (id,btn.form)         7.
# (idata,hst) = mkIData (Init,f (mkFormId editLabel v)) hst           8.
= (done.value,idata.value,idata.form ++ [btnF],hst)                   9.
where editLabel  = label +> "_Editor"                                 10.
      btnLabel   = label +> "_Button"                                 11.
      storeLabel = label +> "_Store"                                  12.
```

In the function `taskEditor` we create, as usual, an iData element for the value `v` (line 8). The `label` argument is used to create three additional identifiers for the value (`editLabel`), the button element (`btnLabel`), and the boolean storage element (`storeLabel`). In Clean, `String`s are arrays of unboxed `Char`s. For convenient `String` concatenation, the overloaded operators (x`+>str`) and (`str<+`x) are used which concatenate the string representation of x and `str`.

The trigger button (line 4) is a simple button that, when pressed, has the function value (`const True`), and which is the identity function `id` otherwise. The boolean storage is created as an iData storage (lines 5–6). It is interconnected with the trigger button by its value: it applies the function value of the button to its boolean value (initially `False`). Therefore, the value of the boolean storage becomes `True` only if the user presses the trigger button. If the user has indicated that the editor has terminated, then the trigger button should not appear (`Br` encodes `<br>`), and the iData element should be in `Display` mode, and otherwise the trigger button should be shown (`btn.form`) and the iData element should still be editable (line 7). In this way, the user is forced to continue with whatever user interface is created after pressing the trigger button.

By using `taskEditor` instead of a regular editor we can formulate a solution to Wadler's exercise.

```
sequenceIData :: *HSt → (Html,*HSt)
sequenceIData hst
# (done1,v1,form1,hst) = taskEditor "Done" "v1" 0 hst
# (done2,v2,form2,hst) = taskEditor "Done" "v2" 0 hst
= mkHtml "Naive solution:"
  [          BodyTag form1
  , if done1 (BodyTag form2) EmptyBody
  , if done2 (BodyTag [Txt "+",Hr [],toHtml (v1+v2)]) EmptyBody
  ] hst
```

In this solution, different HTML code is generated depending on which iData element is finished. In this case, the exercise was not hard, but the resulting code is not very declarative either. We have to explicitly analyze in which state the program is (which tasks are finished or not). Clearly, this style of programming will not scale to programs where many different control flows are possible. Additionally, we need to invent unique identifiers (`"v1"`, `"v2"`, ...) for every iData element.

However, the basic idea of extending iData elements with a boolean storage and trigger button turns out to be a valid one. We use it in the next section to create a scalable solution.

3.2 The iTask Core System

The reason why most web applications are much harder to program, read and understand when compared with desktop applications is that desktop applications can directly interact with the environment at any point in time because they are directly connected with that environment. Due to the client-server architecture, a web application cannot do this. A web application emits an HTML page and terminates. It has to store information somewhere to handle the next request from the user in an appropriate way. It has to recover the relevant states, find out what it was doing and what it has to do next. The resulting code is hard to understand.

A conceivable alternative approach is to adopt the Seaside approach (Ducasse et al. 2004). If the application can automatically remember where it was, programs become easier to write and read. Since a Clean application is compiled to native code, suspending execution, as Seaside does, involves creating core dumps of the run-time system. However, a work flow system needs to support several users that work together. The action of one user can influence the work of others. A core dump only reflects the work of one user. For this reason, we propose a simpler set-up of the system: we start the same application from scratch, as we already did, and use iData elements to remember the state for all users. For a programmer, the application still appears to behave as if it continues evaluation after an I/O request from a browser.

In this section we introduce the main principles of the iTasks system. For didactic reasons we restrain ourselves to a strongly simplified iTasks core system. This core system is single user and has limited possibilities to manipulate tasks. With the core system, we create a satisfying solution to Wadler's exercise. In Sect. 4 we extend this to a full fledged multi-user system.

3.2.1 Editors as Primitive iTask in the Core System

iTasks are defined on top of iData. An iTask is a state transition function of the following type:

```
:: Task a :== *TSt → (a,*TSt)
```

Later in this section we show how tasks can be sequentially composed. iTasks work on a unique state *TSt which extends the unique iData state *HSt with a boolean value activated to indicate the status of a task (when a task is called it tells whether it has to be activated or not, when a task has been evaluated it tells whether it is finished or not), a tasknr for the automatic generation of fresh task identifier values, and html which accumulates all HTML output. For each of these fields, we introduce corresponding update functions (set_activated, set_tasknr, and set_html).

```
:: *TSt        = { hst       :: *HSt
                 , activated :: Bool
                 , tasknr    :: TaskID
                 , html      :: [BodyTag] }
:: TaskID     :== [Int]
set_activated :: Bool      *TSt → *TSt
set_tasknr    :: TaskID    *TSt → *TSt
set_html      :: [BodyTag] *TSt → *TSt
```

We first introduce a function that lifts an extended iData element, as described in Sect. 3.1, to an iTask:

```
editTask :: String a → Task a | iData a
editTask name a = doTask editTask'
where
  editTask' tst=:{tasknr,hst,html}
  # (done,na,nhtml,hst) = taskEditor name (toString tasknr) a hst
  = (na,{tst & activated = done, hst = hst, html = html ++ nhtml})
```

editTask takes an initial value of any type and delivers an iTask of that type. When the task is activated, an extended iData element is created by calling taskEditor. Any iData element automatically remembers the state of any edit action, no matter how complicated the editor is. The HTML code produced by taskEditor is added to the accumulator of the iTask state. In the end all HTML code of all iTasks can be displayed by showing the HTML of the top-task. There can be many active iTasks, so in practice this is probably not what we want. However, for the core system this will do.

The function doTask is an internal wrapper function that is used within the iTasks toolkit for every iTask (note that o is function composition).

```
doTask :: (Task a) → Task a | iCreate a
doTask mytask          = evalTask o incTaskNr
where evalTask tst=:{activated, tasknr}
      | not activated = (default, tst)
      # (val, tst)    = mytask tst
      = (val,{tst & tasknr = tasknr})
```

doTask first ensures that the task number is incremented. In this way, each task obtains a unique number, which eliminates the shortcoming that was mentioned in Sect. 3.1. Tasks are numbered systematically, in the same way as chapters, sections and subsections are numbered in a book or in this paper: tasks on the same level are numbered subsequently with incTaskNr below, whereas a subtask j of task i is numbered i.j with subTaskNr below. Fresh subtask numbers are generated with newSubTaskNr. We use a reversed list of integers for this administration.

```
incTaskNr    tst = {tst & tasknr = case tst.tasknr of
                                     []     → [0]
                                     [i:is] → [i+1:is] }
subTaskNr  i tst = {tst & tasknr = [ i:tst.tasknr]}
newSubTaskNr tst = {tst & tasknr = [-1:tst.tasknr]}
```

The systematic numbering is important because it is also used for garbage collection of subtasks (see Sect. 5).

Next doTask checks whether the task indeed is the next task to be activated by inspecting the value of tst.activated:

- If not activated, the default value is returned. This explains the overloading context restriction of doTask. As a consequence, an iTask *always has a value*, just as an iData element.
- If activated, the task can be executed. This means that the user can select this task via the web interface, and proceed by generating an input event for this task. Task definitions are compositional, so the started tasks may consist of many subtasks of arbitrary complexity. When a task is started, it is either activated (or re-activated for further evaluation) or, in case the task has already been finished in the past, its result is stored as an iData object and is retrieved. In any of these cases, the result of a task (either finished or not yet finished) is returned to the caller of doTask and the task number is reset to its original value.

  It is assumed that any task sets activated to True if the task is finished (indicating that the next task can be activated), and to False otherwise. In the latter case the user still has to do more work on it in the newly created web page.

3.2.2 Basic Combinators of the Core System

Now we introduce iTask combinators for the sequential composition of iTasks. Thanks to uniqueness typing we can freely choose how to thread the unique iTask state *TSt: either explicitly in the Clean style or implicitly using a monadic style. In the implementation of the iTask system we have chosen for the explicit style: it gives more flexibility because we have direct access to both the unique iTask state *TSt and the unique iData state *HSt as is shown in the definition of editTask. However, to the application programmer *TSt is an opaque environment, and for her we provide a monadic interface.

```
(=>>) infix  1 :: (Task a) (a → Task b) → Task b
(#>>) infixl 1 :: (Task a)      (Task b) → Task b
return         :: a → Task a
```

It is convenient to have an alternative return_D function that also displays the returned value. Its definition is straightforward:

```
return_D :: a → Task a | gForm{|*|}, iCreateAndPrint a
return_D a = doTask (λtst → (a,{tst & html = tst.html ++ toHtml a}))
```

When a task is in progress, it is useful to provide feedback to the user what she is supposed to be doing. For this purpose two combinators are introduced. $(p \ ?\!\gg t)$ is a task that displays prompt p as long as task t is running, whereas $(p \ !\!\gg t)$ always displays prompt p as soon as task t is activated.

```
(?>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(?>>) prompt task = prompt_task
where
    prompt_task tst=:{html = ohtml,activated}
    | not activated = (default,tst)
    # (a,tst=:{activated,html = nhtml}) = task {tst & html = []}
    | activated     = (a,{tst & html = ohtml})
    | otherwise     = (a,{tst & html = ohtml ++ prompt ++ nhtml})

(!>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(!>>) prompt task = prompt_task
where
    prompt_task tst=:{html = ohtml,activated}
    | not activated          = (default,tst)
    # (a,tst=:{html = nhtml}) = task {tst & html = []}
    = (a,{tst & html = ohtml ++ prompt ++ nhtml})
```

With these definitions, the solution to Wadler's exercise, given by sequenceITask in Fig. 3, becomes surprisingly simple. Notice that the solution not only works for integers, since sequenceITask is overloaded. It works for any type on which iData and + are defined. The implementation is concise, which is completely due to the power of the underlying iData library.

```
sequenceITask :: Task a | iData, + a
sequenceITask
= editTask "Done" default ⋙= λv1 →
  editTask "Done" default ⋙= λv2 →
  [Txt "+",Hr []]
  !⋙ return_D (v1+v2)
```

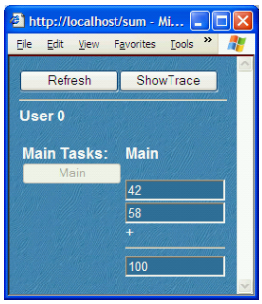

Figure 3. The sum exercise, as an iTask application.

The screen shot shows that the iTasks implementation adds a tracing option at the top of the window. For each user, the main tasks are displayed in a column. The selected main task is displayed next to this column.

3.2.3 Reflection (Part I)

The behavior of the described core system is a combination of re-evaluating the application and having the extended iData elements retrieve their previous states that are possibly updated with the latest changes done by the application user. The Clean application is still restarted from scratch when a new page is requested from the browser. However, the application will now automatically find its way back to the tasks it was working on during the previous incarnation. Any iTask editor created with editTask automatically remembers its contents and state (finished or not) while the other iTask combinators are pure functions which can be recalculated and in this way the system can determine which other tasks have to be inspected next. Tasks that are not yet activated might deliver some default value, but it is not important because it is not used anywhere yet, and the task produces no HTML code. In this way we achieve the same result as in Seaside, albeit that we reconstruct the state of the run-time system by a combination of re-evaluation from scratch and restoring of the previous edit states.

3.2.4 Work Flow Pattern Combinators of the Core System

The core system presented above is extendable. The sequential composition is covered by the combinators ⋙= and ♯⋙. In this section we introduce parallel composition, repetition and recursion. In Sect. 6 we discuss their relation with work flow patterns found in contemporary work flow tools.

The infix operator (t_1 -&&- t_2) activates subtasks t_1 and t_2 and ends when both subtasks are completed; the infix operator (t_1 -||- t_2) also activates two subtasks t_1 and t_2 but ends as soon as one of them terminates, but it is biased to the first task at the same time. In both cases, the user can work on each subtask in any desired order. A subtask, like any other task, can consist of any composition of iTasks.

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iCreate a
                                                   & iCreate b
(-&&-) taska taskb = doTask and
where
    and tst=:{tasknr}
    ♯ (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
    ♯ (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
    = ((a,b),set_activated (adone && bdone) tst)

(-||-) infixr 3 :: (Task a) (Task a) → Task a | iCreate a
(-||-) taska taskb = doTask or
where
    or tst=:{tasknr}
    ♯ (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
    ♯ (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
    = (if adone a (if bdone b default)
      ,set_activated (adone || bdone) tst
      )

mkParSubTask :: Int TaskID (Task a) → Task a
mkParSubTask i tasknr task
    = task o newSubTaskNr o set_activated True o subTaskNr i
```

The function mkParSubTask is a special wrapper function for subtasks. It is used to activate a subtask and to ensure that it gets a correct task number.

Another iTask combinator is foreverTask which repeats a task infinitely many times.

```
foreverTask :: (Task a) → Task a | iCreate a
foreverTask task
    = doTask (foreverTask task o snd o task o newSubTaskNr)
```

As an example, consider the following definition:

```
t = foreverTask (sequenceITask -||- editTask "Cancel" default)
```

In t the user can work on sequenceITask, but while doing this, she can always decide to cancel it. After completion of any of these alternatives the whole task is repeated.

More general than repetition is to allow arbitrary recursive work flows. A *crucial* combinator for recursion is newTask.

```
newTask :: (Task a) → Task a | iCreate a
newTask task = doTask (task o newSubTaskNr)
```

(newTask t) promotes any user defined task t to a proper iTask such that it can be recursively called without causing possible non-termination. It ensures that t is only called when it is its turn to be activated and that an appropriate subtask number is assigned to it. Consider the following example of a recursive work flow:

```
getPositive :: Int → Task Int
getPositive i = newTask (getPositive' i)                          1.
where                                                             2.
    getPositive' i = [Txt "Type in a positive number:"]           3.
                     ?⋙ editTask "Done" i ⋙= λni →                4.
                     if (ni > 0) (return ni) (getPositive ni)     5.
```

Function getPositive requests a positive number from the user. If this is the case the number typed in is returned, otherwise the task calls itself recursively for a new attempt. This example works fine. However, it would not terminate if getPositive' calls itself directly in line 5 instead of indirectly via a call to newTask. Remember that every editor returns a value, whether it is finished or not. If it is not yet finished, it returns default. The default value for type Int happens to be zero, and therefore by default getPositive' goes into recursion. The function newTask will prevent infinite recursion because the indicated task will not be activated when the previous task is not yet finished. Hence, one has to keep in mind to regard getPositive as a task that can be recursively activated, and not as a plain recursive function.

The combinator repeatTask repeats a given task, task, until the predicate p holds.

```
repeatTask task p = t default
where
   t v = newTask (task v) ⋙= λnv → if (p nv) (return_D nv) (t nv)
```

Using this combinator the task getPositive can be expressed as:

```
getPositive = repeatTask (λi → [Txt "Type in a positive number:"]
                               ?⋙ editTask "Done" i) (λx → x>0)
```

Note the importance of the place of newTask. If it would be moved to the recursive call, by replacing (t v) by newTask (t v), the task would always be executed immediately for a first time (i.e. without waiting for activation). This is generally not the desired behavior.

3.2.5 Reflection (Part II)

With the combinators presented above, iTasks can be composed as desired. As discussed in Sect. 3.2.4, one can imagine all kinds of additional combinators. For all well-known work flow patterns we have defined iTask combinators that mimic their behavior. In the next section we discuss the most important ones and show their usage. The actual implementation of the combinators in the iTask library is more complicated than the combinators introduced in the core system. There are additional requirements, such as:

Presentation issues: One can construct complicated tasks that have to be presented to users systematically and clearly. The system needs to prompt the user for information on the right moment, remove feedback information when it is no longer needed, and so on. A user might have to work on several tasks in any order she wants. Such tasks have to be presented clearly as well, e.g. by creating a separate web page for each task and a button to navigate between these tasks.

Multiple users: A work flow system is a multi-user system. Tasks can be assigned to different users, persistent storage and retrieval of information in a database needs to be handled, think about version control, ensure consistent behavior by ruling out possible race conditions, ensure that the correct information is communicated to each user, inform a user that she has to wait on information to be produced by someone else, and so on.

Efficiency: Real world work flow systems run for years. How can we ensure that the system will scale up and that it can reconstruct itself efficiently?

Features: One can imagine many more options one would like to have. For instance, it might be important that tasks are performed on time. A manager might want to know which tasks and/or persons are preventing the completion of other tasks.

In the next section we present the iTask combinators including support for these features. The consequences this has for the implementation of our core system is described next.

4. Overview of the iTasks System

In this section we present the main concepts of the iTasks toolkit by means of a number of examples. Please note that despite their small size, they do represent complex work flow situations that occur in the real world. Some of these situations can not be handled by contemporary work flow specification tools.

4.1 Sequence and choice: a single step coffee machine

Coffee vending machines are popular examples to illustrate sequencing and choice. We present an example of a coffee machine that offers the user either coffee or tea. After choosing, the user pays the proper amount of money and obtains the selected product. This also terminates the coffee machine:

```
Start world = doHtmlServer (singleUserTask coffeemachine) world

coffeemachine :: Task (String,Int)
coffeemachine
= [Txt "Choose product:"]
  ?>> chooseTask [(p <+ ": " <+ c, return prod)
                  \\ prod=:(p,c) <- products
                  ] =>> λprod=:(p,c) ->
  [Txt ("Chosen product: " <+ p)]
  ?>> pay prod (buttonTask "Thanks" (return prod))
where
  products     = [("Coffee",100),("Tea",50)]
  pay (p,c) t = buttonTask ("Pay " <+ c <+ " cents") t
```

The combinators that are used in this example are:

```
buttonTask ::   String (Task a) -> Task a | iCreateAndPrint a
chooseTask :: [(String, Task a)] -> Task a | iCreateAndPrint a
```

(buttonTask l t) enhances a task t with a push button labeled with l that needs to be pressed first by the user before she can do t. Choosing between alternatives of labeled actions l_i and tasks t_i is given by chooseTask $[(l_0,t_0)\ldots(l_n,t_n)]$. The resulting value is the value of the selected task t_i.

The function singeUserTask is a wrapper function that converts an iTask to an iData environment transformer function:

```
singleUserTask :: (Task a) *HSt -> (Html,*HSt) | iCreate a
```

4.2 Repetition, recursion and state: a coffee machine

The coffee machine in the previous example offers a single beverage, and terminates. It is extended to an eternal vending machine with the foreverTask combinator:

```
Start world
= doHtmlServer (singleUserTask (foreverTask coffeemachine)) world
```

The previous example abstracted from the paying task: the function (pay (p,c) t) offers a labeled action to pay the full amount of money, and then continues with task t. In a more refined model, the user is able to insert coins until the inserted amount of money exceeds the cost of the product. Moreover, she can also choose to abandon the paying task and not get the selected drink at all. This is suitably modeled with a recursive task specification:

```
getCoins :: ((Bool,Int,Int) -> Task (Bool,Int,Int))
getCoins = repeatTask get (λ(cancel,cost,_) -> cancel || cost <= 0)
where
  get (cancel,cost,paid)
        = newTask "pay" ([Txt ("To pay: " <+ cost)]
          ?>> chooseTask [(c +> " cents", return (False,c))
                          \\ c <- coins
                         ]
              -||-
              buttonTask "Cancel" (return (True,0))
          =>> λ(cancel,c) -> return (cancel,cost-c,paid+c))
  coins   = [5,10,20,50,100,200]
```

The crucial combinator in this definition is newTask which was introduced in Sect. 3.2.4 (the additional string argument is used for tracing). Clearly, we regard getCoins not as a common recursive function, but as a definition of a recursive task that has to be activated when the previous task, which might be the previous invocation of getCoins, is finished.

We can now redefine the pay function of Sect. 4.1:

```
pay (p,c) t = getCoins (False,c,0) =>> λ(cancel,_,paid) ->
              [Txt ("Product = "<+if cancel "cancelled" p
                                 <+". Returned money = "<+(paid-c))]
              ?>> t
```

It should be noted that getCoins and pay illustrate that tasks may depend on the actual values that are generated within the system. These kind of workflows are hard to model with current day work flow specification tools.

4.3 Multi-User Workflows

The solution to Phil Wadler's exercise that was given in Sect. 3, was a *single user* application. Work flow systems usually involve arbitrarily many users. This is supported by the iTask system.

```
multiUserTask :: Int (Task a) *HSt -> (Html,*HSt) | iCreate a
:: UserID :== Int
```

We identify users with index values $i \geq 0$. The wrapper function (multiUserTask n t) creates a work flow system, defined by t for

users $0 \ldots n-1$. For quick testing, it provides an additional user interface for selecting the proper user.

By default, tasks store their information on the client side. If one wants to use the system with multiple users over the net, one has to store iTask information persistently on the server side. To conveniently control this, we introduce similar operations as <@ and @> (Sect. 2.1).

```
class (<<@) infix 3 att :: (Task a) att → Task a
class (@>>) infix 3 att :: att (Task a) → Task a
instance <<@ Lifespan, Mode
instance @>> Lifespan, Mode
```

The operators can be applied to any task or task expression to set the attributes of all underlying iData elements. So, (task<<@Persistent) stores the information of all the underlying iData in files while (task<<@Database) stores the information in a relational database. Storage and retrieval is handled automatically (Sect. 2).

Assigning a task t to user i with some motivation m is done by (m,i)@:t. If there is no motivation, then one uses i@::t.

```
(@:)  infix 3 :: (String,UserID) (Task a) → Task a | iCreate a
(@::) infix 3 ::         UserID  (Task a) → Task a | iCreate a
```

Suppose that the first integer editing task in Wadler's exercise should be performed by user 1, the second by user 2, and the result is shown to user 0 (the default user). The code becomes:

```
sequenceMU :: Task a | iData, +, zero a
sequenceMU
=   ("Enter a number",1) @: editTask "Done" zero ⋙= λv1 →
    ("Enter a number",2) @: editTask "Done" zero ⋙= λv2 →
    [Txt "+",Hr []] !>> return_D (v1 + v2)

Start world
= doHtmlServer (multiUserTask 2 sequenceMU <<@ Persistent) world
```

The iTask system ensures that each user sees only tasks assigned to them. This is essentially a *filter* of the full task tree, because any task may decide to assign tasks to any other user.

4.4 Speculative tasks and multiple users: deadlines

Work flow systems need to handle time-related tasks: some task t has to be finished before a given time T or it is canceled. In this example we show how this is expressed with the iTasks toolkit. The time related combinators are the following:

```
waitForDateTask  :: HtmlDate → Task HtmlDate
waitForTimeTask  :: HtmlTime → Task HtmlTime
waitForTimerTask :: HtmlTime → Task HtmlTime
```

The algebraic types HtmlDate and HtmlTime are elements of the iData toolkit that have been specialized to show user convenient date and time editors. waitForDate(Time)Task terminates in case the given date (time of day) has passed; waitForTimerTask terminates after a given time interval.

In our example, we use the latter combinator to delegate work:

```
delegateTask who time t                                              1.
=   ("Timed Task",who)@:                                             2.
      waitForTimerTask time ⋗>> return Nothing                       3.
      -||-                                                           4.
      [Txt ("Please finish task before" <+ time)]                    5.
      ?>> (t ⋙= λv → return (Just v))                                6.
    )                                                                7.
```

(delegateTask i dt t) assigns a task t to user i that needs to be finished before dt time (line 5–6) is passed. If the user does not complete the task on time, delegation fails, and should also terminate (line 3).

The main work flow situation is modeled as follows:

```
deadline :: (Task a) → Task a | iData a                              1.
deadline t                                                           2.
= [Txt "Choose person you want to delegate work to:"]                3.
  ?>> editTask "Set" (PullDown size  (0,map toString [1..n]))        4.
                                        ⋙= λwho →                    5.
  [Txt "How long do you want to wait?"]                              6.
  ?>> editTask "SetTime" default        ⋙= λtime →                   7.
  [Txt "Cancel delegated work if you get impatient:"]                8.
  ?>> delegateTask who time t                                        9.
      -||-                                                           10.
      buttonTask "Cancel" (return Nothing) ⋙= check                  11.
where
  check result                                                       12.
  = case result of                                                   13.
      (Just value) → [Txt ("Result of task: " <+ value)]             14.
                     ?>> buttonTask "OK" (return value)              15.
      Nothing → [Txt "Task expired/canceled; do it yourself!"]       16.
                     ?>> buttonTask "OK" t                           17.
```

The main task consists of selecting a user to whom a task t should be delegated (lines 3–5), deciding how much time this user is given for this exercise (lines 6–7), and then delegating the task (line 9). We also model the situation that the current user gets impatient, and decides to abandon the delegated task (line 11). Either way, we know whether the task has succeeded and display the result and terminate (lines 14–15), or the current user has to do it herself (lines 16–17).

The work flow described by (deadline t) defines a single delegation. It can be transformed into an iteration with the foreverTask combinator that we have also used in Sect. 4.2. We are obviously creating a multi-user system, and hence use the multiUserTask wrapper function for some constant $n > 0$. As example task we reuse the task getPositive from Sect. 3.2.4. This finalizes the example:

```
Start world
= doHtmlServer
    (multiUserTask n
       (foreverTask (deadline getPositive) <<@ Database)) world
```

4.5 Parameterized tasks: a reviewing process

In this example we show that iTasks and iData cooperate in close harmony. We present a reviewing process in which the product of a user is judged by a reviewer who can either approve, reject, or demand rework of the product. The latter is described with an algebraic data type:

```
:: Review = Approved | Rejected | NeedsRework TextArea
```

TextArea is an algebraic data type that is specialized by the iData toolkit as a multi-line text edit box that can be used by the reviewer to enter comments.

A reviewer inspects the product v that needs to judged, and makes a decision. This is defined concisely as:

```
review :: a → Task Review | iData a
review v
= [toHtml v]
  ?>> chooseTask
  [("Rework",   editTask "Done" (NeedsRework default) <<@ Submit)
  ,("Approved", return Approved)
  ,("Reject",   return Rejected)
  ]
```

Any task result that can be displayed can also be subject to reviewing, hence the restriction to the generic iData class.

The main task is to produce a product v according to some task t that can be judged by a reviewer u. If the reviewer demands rework of v, the task should be restarted with that particular v, because the user would have to completely recreate a new product otherwise. Therefore, the product and the task to produce it are given as a pair

(a, a→Task a), and the result of the main task is to return a product and its review (a,Review). This is done as follows:

```
taskToReview :: UserID (a,a→Task a)→Task (a,Review) | iData a 1.
taskToReview reviewer (v,task)                               2.
= newTask "taskToReview"                                     3.
  ( task v                  ⋙= λnv →                         4.
    reviewer @:: review nv ⋙= λr →                           5.
    [Txt ("Reviewer " <+ reviewer <+ " says "),toHtml r]     6.
    ?>> buttonTask "OK"                                      7.
    case r of                                                8.
      (NeedsRework _) →taskToReview reviewer (nv,task)       9.
      else            →return (nv,r)                         10.
  )
```

The task is performed to return a product (line 4), which is reviewed by the given reviewer (line 5). Her decision is reported (line 6), and only in case of a demanded rework, this has to be repeated (line 9).

For the example, we select a two-user system (multiUserTask 2) in which user 0 creates the product, and user 1 reviews it:

```
Start world
= doHtmlServer (multiUserTask 2 (taskToReview 1 (default,t))) world

t v = [Txt "Fill in Form:"] ?>> editTask "TaskDone" v <<@ Submit
```

Note the high degree of parameterization and therefore reusability of the code: taskToReview handles *any* task, and by providing *only* a type signature to t above, we get a form task for values of that type for free. For example, when t is of type Person (Sect. 2.1) an editor for this type is created automatically.

4.6 Higher order tasks: shifting work

A distinctive feature of the iTask system is that tasks can be higher order: data can be communicated but also (partially evaluated) tasks can. One can create task closures, i.e. tasks that already have been partially evaluated by someone and ship them to some other user who can continue to work on it.

```
:: TClosure a = TClosure (Task a)

(-!>) infix 4 :: (Task s) (Task a) →Task (Maybe s, TClosure a)
                                   | iCreateAndPrint s & iCreateAndPrint a
```

The proper generic functions have been specialized for type TClosure such that it acts as a container of tasks. Any task can be put in a value of this type, but we want to be able to put a partially evaluated task in it. Therefore we need a way to interrupt a task that is being evaluated. *(stop -!> t)* (the implementation of -!> is discussed in Sect. 5.4) is a variant of an or-task which takes two tasks: whenever *stop* is done, t is interrupted and this possibly partially evaluated task is delivered as result. However, t can also finish normally, and the fully completed task is delivered. The result of *stop*, therefore, is only returned when it finishes before t.

```
delegate :: (Task a) [UserID] →Task a | iData a                1.
delegate task set = newTask "delegate" doDelegate              2.
where                                                          3.
 doDelegate                                                    4.
 = findSomeone ⋙= λwho →                                       5.
   who @:: stopTask -!> task ⋙= λ(stopped,TClosure task) →     6.
   if (isJust stopped) (delegate task set) task                7.

 findSomeone                                                   8.
 = orTasks [("Wait " <+ who                                    9.
           , who @:: buttonTask "I will do it" (return who))   10.
            \\ who ←set]                                       11.

 stopTask = buttonTask "Stop" (return True)                    12.
```

The function delegate first tries to findSomeone to perform the task (line 5). All persons in set are asked whether they want the task. The first user who accepts the task obtains it and she can work on it. However, the work can be interrupted by completion of stopTask which ends when the user has pushed the Stop button. If this is the case, all persons are asked again to volunteer for the job. The one who accepts, obtains the task in the state as it has been left by the previous worker and she can continue to work on it. The whole recursively defined process finally ends when the delegated task is fully completed by someone.

The conditions for stopping a task can be arbitrarily complex. For instance, by using stop2 not only the user herself can stop the task, but someone else can do it for her as well (e.g. the user who delegated the task in the first place), or it can be timed out.

```
stop2 user time = stopTask -||- (user @:: stopTask) -||- timer time
timer time      = waitForTimerTask time ⋗>> return True
```

5. Implementation

As mentioned earlier in Sect. 3.2.5, the actual implementation is more complicated because it needs to support more features. We discuss the most interesting aspects by building on the core system.

5.1 Handling Multiple Users

On each event every iTask application is (re)started for all its users. All tasks are recalculated from scratch, but only for one user the tasks are shown. By default, tasks are assigned to user 0. As presented in Sect. 4.3, users can be assigned to tasks with the operators @: and @::. We give the HTML accumulator within the TSt environment (Sect. 3.2.1) a tree structure instead of a list structure, and we keep track of the user to whom a task is assigned, as well as the identification of the application user:

```
:: *TSt      = { ...
                 , myId      :: UserID      // id of task user
                 , userId    :: UserID      // id of application user
                 , html      :: HtmlTree }  // accumulator for html code
:: HtmlTree = BT [BodyTag]
            | (@@:) infix 0 (UserID,String) HtmlTree
            | (-@:) infix 0  UserID         HtmlTree
            | (+-+) infixl 1 HtmlTree       HtmlTree
            | (+|+) infixl 1 HtmlTree       HtmlTree
defaultUser = 0
```

(BT *out*) represents HTML output; ((*u*,*name*)@@:*t*) assigns the html tree t to user u where *name* is the button with which the user can select this task; (*u*-@:*t*) also assigns the html tree t to user u, but now t should not be displayed. The remaining constructors (t_1+-+t_2) (and (t_1+|+t_2)) place output t_1 left (above) of output t_2.

In a single-user application, the only user is defaultUser; in a multi-user application, the current user can be selected with a menu at the top of the browser window. This feature is added for testing, for the final application one needs of course to add a decent login procedure. Initially, myId is defaultUser, userId is the selected user, and the accumulator html is empty (BT []). After evaluation of a task, the accumulator contains all HTML output of each and every activated iTask. It is not hard to define a filtering function that extracts all tasks for the current user from the output tree.

Version management is important as well for a multi-user web enabled system. Back buttons of browsers and cloning of browser windows might destroy the correct behavior of an application. For every user a version number is stored and only requests matching the latest version are granted. An error message is given otherwise after which the browser window is updated showing the most recent version. Since we only have one application running on the server side, storage and retrieval of any information is guaranteed to be indivisible such that problems in this area cannot occur.

Another aspect to think about is that the completion of one task by one user, e.g. a Cancel action, may remove tasks others are

working on (see e.g. the deadlines example in Section 4.4). This effects the implementation of all choice combinators: one has to remember which task was chosen to avoid race conditions.

5.2 Optimizing the Reconstruction of the Task Tree

An iTask application reconstructs itself over and over each time a client browser is manipulated by someone. The more progress made in the application, the more tasks are created. Hence, the evaluation tree increases in size and it takes longer to reconstruct it. For a real world work flow application this is unacceptable.

We optimize the reconstruction process similar to the normal rewriting that takes place in the implementation of functional languages such as Clean and Haskell. When a closure is evaluated, the function call is replaced by its result. Similar, when a task is finished, it can be replaced by its result. We have to store such a result persistently, for which we can of course again use an iData element. However, it is not necessary to optimize each result in order to avoid the creation of too many iData storages. We can freely choose between recalculation (saving space) or storing (saving time). In the iTask toolkit we have decided to optimize "big" tasks only. Combinators such as repeatTask produce only intermediate results and can be replaced by the next call to itself. For these kinds of combinators the task tree will not grow at all. However, user defined tasks that are created with newTask are likely being used to abstract from such "big" tasks.

Here is what the actual newTask combinator does, as opposed to the core version of Sect. 3.2.4.

```
newTask :: (Task a) → Task a | iData a                                  1.
newTask t = doTask (λtst=:{tasknr,hst}                                  2.
 ♯ (taskval,hst) = mkStoreForm (Init,storeId) id hst                    3.
 ♯ (done,v)      = taskval.value                                        4.
 | done          = (v,{tst & hst = hst})                                5.
 ♯ (v,tst=:{activated = done,hst})                                      6.
                 = t {tst & tasknr = [-1:tasknr],hst = hst}             7.
 | not done      = (v,{tst & tasknr = tasknr})                          8.
 ♯ (_,hst)       = mkStoreForm (Init,storeId) (const (True,v)) hst      9.
 = (v,{tst & tasknr = tasknr, hst = hst})                               10.
)                                                                       11.
where storeId  = mkFormId (tasknr +> "_New")                            12.
                          (False,default) <@ Session                    13.
```

A storage is associated with task t (line 3) that initially has a default value (lines 12–13). If the task was finished in the past, it is not re-evaluated. Instead, its value is retrieved from the storage (line 4 and 5), otherwise it needs to be evaluated (lines 6–7). If the user actions have not terminated task t, then it has not produced a final value yet, thus the storage need not be updated (line 8). If the user has terminated the task, then the storage is updated with the final value (line 9), and a boolean mark to prevent re-evaluation of this particular "redex".

5.3 Garbage Collection of iData Objects

The optimization described above will prevent the task evaluation tree from growing, but all persistent iData objects created in previous runs are not garbage collected automatically. Although certain results are not needed for the computation of the task tree anymore, one nevertheless might want to keep them for other reasons. Consider the gathering of statistical information such as "who has performed a certain task in the past?" and "which tasks have taken a long time to complete?", or one wants to remember a result of a task, but not of any of its subtasks. For this reason, we have provided an option that can be switched on and off to automatically take care of the garbage collection of tasks and their subtasks, no matter where they are stored. The numbering discipline plays a crucial role in identifying which subtasks belong to a given task, such that any choice of garbage collection strategy can be implemented.

5.4 Higher-Order Tasks

A distinctive feature of the iTask toolkit is the ability to communicate higher-order tasks that have been partially evaluated (Sect. 4.6). In the real world it is obvious that work that has been done partially can be handed over to other persons who finish the work. This is not one of the standard work flow patterns that can be found in contemporary work flow tools (see (van der Aalst et al. 2002)). We show that the iTask toolkit does support this work flow pattern, and that it does so in a concise way. The complete realization of the $(p\text{-}!\triangleright t)$ is as follows:

```
(-!>) infix 4 :: (Task s) (Task a) → Task (Maybe s,TClosure a)          1.
              | iCreateAndPrint s & iCreateAndPrint a                   2.
(-!>) p t = doTask (λtst=:{tasknr,html}                                 3.
 ♯ (v,tst=:{activated = done,html = task})                              4.
           = t {set (BT []) True tst & tasknr = taskId}                 5.
 ♯ (s,tst=:{activated = halt,html = stop})                              6.
           = p {set (BT []) True tst & tasknr = stopId}                 7.
 | halt    = return (Just s, TClosure (close t))                        8.
                    (set  html                    True  tst)          9.
 | done    = return (Nothing,TClosure (return v))                       10.
                    (set (html +|+ task)          True  tst)          11.
 | otherwise = return (Nothing,TClosure (return v))                     12.
                    (set (html +|+ task +|+ stop) False tst)            13.
)                                                                       14.
where close t       = t o (set_tasknr taskId)                           15.
      set html done = (set_html html) o (set_activated done)            16.
      stopId        = [-1,0:tasknr]                                     17.
      taskId        = [-1,1:tasknr]                                     18.
```

Both the suspendable task t and the terminator task p are evaluated (lines 4–5 and 6–7). Their current renderings are task and stop respectively, and they both contain the most recent user edit operations. The most exciting spot is line 8: if p is finished (condition halt is true), then the task t *as far as it has been evaluated* has to be returned. However one has to realize that a task t is only a recipe that is executed by applying it to its state. When a task is executed, it *always* returns a result and a state, even if the task is not yet finished. This also holds for task t when it is activated in line 5. There actually are no partially evaluated task closures in this system, there are only tasks and when they are applied they return their result. The crucial issue is how to return a partially evaluated task if none exist? The answer is given in line 15! Remember that an iTask application can reconstruct itself completely from scratch. This property also holds for any iTask expression in the application. The only thing we need is the task recipe and the state of a task, and in particular, the task number stored in this state. Given a task number and a task we can reconstruct the work done so far! So by passing the task function and the task number to somebody else, the work can be reconstructed and the person can continue the work. Line 15 assures that an interrupted task is reapplied on the original task number when it is restarted.

6. Related Work

In the realm of functional programming, many solutions that have been inspiring for our work have been proposed to program web applications. We mention just a few of them in a number of languages: the HaskellCGI library (Meijer 2000); the Curry approach (Hanus 2001); writing XML applications (Elsman and Larsen 2004) in *SMLserver* (Elsman and Hallenberg 2003). One sophisticated system is WASH/CGI by Thiemann (2002), based on Haskell. Here, HTML is produced as an effect of the CGI monad whereas we consider HTML as a first-class citizen, 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. In iTasks, we replay the program once instead of the session, and restore the state of the program on-the-

fly using the storage capabilities of the underlying iData. Forms are programmed explicitly in HTML, and their elements may, or may not, contain values. In the iTask toolkit, forms and tasks are generated from arbitrary data types, and always have a value. Interconnecting forms in WASH/CGI is done by adding callback actions to submit fields, whereas the iData toolkit uses a functional dependency relation.

Two more recent approaches that are also based on functional languages are Links (Cooper et al. 2006) and Hop (Serrano et al. 2006). Both languages aim to deal with web programming within a single framework, just as the iData and iTask approach do. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. Notable differences between Links and iData and iTasks are that the latter has a more refined control over the location of state storage, and even the presence of state, which needs to be mimicked in Links with recursive functions. Compiling to JavaScript gives Links programs more expressive and computational power at the client side: in particular Links offers thread-creation and message-passing communication, and finally, the client side code can call server side logic and vice versa. The particular focus of Hop is on rendering graphically attractive applications, like desktop GUI applications can. Hop implements a strict separation between programming the user interface and the logic of an application. The main computation runs on the server, and the GUI runs on the client(s). Annotations decide where a computation is performed. Computations can communicate with each other, which gives it similar expressiveness as Links. The main difference between these systems and iTasks (and iData) is that the latter are restricted to thin-client web applications, and provide a high degree of automation using the generic foundation.

iData components that reside in iTasks are abstractions of forms. A pioneer project to experiment with form-based services is Mawl (Atkins et al. 1997). It has been improved upon by means of Powerforms (Brabrand et al. 2000), used in the <bigwig> project (Brabrand et al. 2002). These projects provide *templates* which, roughly speaking, are HTML pages with *holes* in which scalar data as well as lists can be plugged in (Mawl), but also other *templates* (<bigwig>). They advocate compile-time systems, because this allows one to use type systems and other static analysis. Powerforms reside on the client-side of a web application. The type system is used to filter out illegal user input. The use of the type system is what they have in common with our approach. Because iData are encoded by ADTs, we get higher-order forms for free. Moreover, we provide higher-order tasks that can be suspended and migrated.

Web applications can be structured with *continuations*. This has been done by Hughes (2000), in his arrow framework. Queinnec (2000) states that "A browser is a device that can invoke continuations multiply/simultaneously". Graunke et al. (2001) have explored continuations as one of three functional compilation techniques to transform sequential interactive programs to CGI programs. The Seaside (Ducasse et al. 2004) system offers an API for programming web pages using a Smalltalk interpreter. When waiting for new information from the browser, a Seaside application is suspended and continues evaluation as soon as input is available. To make this possible, the whole state of the interpreter's run-time system is stored after a page has been produced and this state is recovered when the next user event is posted such that the application can resume execution. In contrast to iTask, Seaside has to be a single user system by construction.

Our approach is simpler yet more powerful: every page has a complete (set of) model value(s) that can be stored and recovered generically. An application is resurrected by restarting the very same program, which recovers its previous state on-the-fly.

Our combinator library has been inspired by the comprehensive analysis of work flow patterns by van der Aalst et al. (2002) of over more than 30 contemporary commercial work flow systems. These patterns are typically based on a Petri-net style, which implies that patterns for *distributing* work (also called *splitting*) and *merging* (*joining*) work are distinct and can be combined more or less arbitrarily. In the setting of a strongly typed combinator style such as iTasks, it is more natural to define combinator functions that pair splitting and merging patterns. For instance, the two combinators -&&- and -||- that were introduced in Sect. 3.2.4 pair the *and split – and join* and *or split – synchronizing merge* patterns. Conceptually, the Petri-net based approach is more fine-grained, and should allow the work flow designer greater flexibility. However, we believe that we have captured the essential combinators of these systems. We plan to study the relationship between the typical functional approach and the classic Petri-net based approach in the near future.

Contemporary commercial work flow tools use a graphical formalism to specify work flow cases. We believe that a textual specification, based on a state-of-the-art functional language, provides more expressive power. The system is strongly typed, and guarantees all user input to be type safe as well. In commercial systems, the connection between the specification of the work flow and the (type of the) concrete information being processed, is not always well typed. Our system is fully dynamic, depending on the values of the concrete information. For instance, recursive work flows can easily be defined. In a graphical system the flows are much more static. Our system is higher order: tasks can communicate tasks. Work can be interrupted and conditionally moved to other users for further completion. Last but not least: we generate a complete working multi-user web application out of the specification. Database storage and retrieval of the information, version management control, type driven generation of web forms, handling of web forms, it is all done automatically such that the programmer only needs to focus on the flow specification itself.

7. Conclusions

The iTask system is a domain specific language for the specification of work flows, embedded in Clean. The specification is used to generate a multi-user interactive web-based work flow management system.

We hope to have convinced the reader that the notation we offer is concise as well as intuitive. For functional programmers the monadic style of programming should look familiar. Users of commercial work flow systems, who design work flows, typically use a graphical formalism for this purpose. For this group of potential users a text based approach is likely to be harder to understand. It should be investigated in what way a mapping from a graphical approach to the textual approach can be constructed.

The iTask toolkit covers all standard work flow patterns in a combinatorial style. Moreover, it adds further expressive power in terms of a strongly typed system, dynamic run-time behavior, and higher-order tasks that can be suspended, passed on to other users, and continued. At the same time it generates a multi-user interactive web-based application that automatically handles sessions, state and state storage, HTML rendering, and more.

This latter feature is due to building the iTask toolkit on top of the iData toolkit. This project provides further evidence that the iData concept is a versatile, elementary unit to create interactive web applications. One particular helpful design decision was to separate handling values and constructing the rendering of the application in the iData toolkit. This allows the iTask toolkit to separately handle the flow of information and the filtering of the correct HTML code for the end user. The iData enabled us to do "task rewriting" in a similar way as expressions are rewritten in languages such as Clean and Haskell. Finally, iTasks profit from

these advantages, and strengthen them by extending the expressive power by defining work flow system on a sophisticated high level of abstraction.

Future work will be the investigation of more "unusual" useful work flow patterns. We will investigate how the system will scale up in terms of performance when applications run longer and have more users. Also we are working on a new option for the evaluation of tasks on the client side using Ajax technology in combination with an efficient interpreter for functional languages (Jansen et al. 2006).

Acknowledgments

The authors would like to thank Phil Wadler for his inspiring exercise, Erik Zuurbier for the many discussions on the state-of-art of contemporary work flow systems and as a source of many examples, Wil van der Aalst for commenting on the difference between the combinator approach and contemporary work flow specification languages, Maarten de Mol and Arjen van Weelden for reading the draft version of the paper, and the anonymous reviewers for their constructive comments.

References

Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, University of Nijmegen, The Netherlands, 2005. ISBN 3-540-67658-9.

David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a Domain Specific Language for Form-based Services. In *Usenix Conference on Domain Specific Languages*, October 1997.

Claus Brabrand, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, 2000.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> Project. In *ACM Transactions on Internet Technology (TOIT)*, 2002.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, CWI, Amsterdam, The Netherlands, 7 - 10 November 2006. Springer-Verlag. to appear.

Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside - A Multiple Control Flow Web Application Framework. In Stéphane Ducasse, editor, *Proceedings ESUG 2004 International Conference – Research Track*, volume Technical Report IAM-04-008, pages 231–254. Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, November 7 2004.

Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.

Martin Elsman and Ken Friis 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.

Paul Graunke, Shriram Krishnamurthi, Robert Bruce Findler, and Matthias 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, September 2001.

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.

Ralf 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. URL urlciteseer.nj.nec.com/hinze99new.html.

John Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.

Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006*, pages 157–172, Nottingham, UK, The University of Nottingham, April 19-21 2006.

Erik Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.

Rinus Plasmeijer and Peter 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 2006a.

Rinus Plasmeijer and Peter 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 2006b. Springer Verlag.

Rinus Plasmeijer and Peter Achten. A Conference Management System based on iData. In Z. Horvath and V. Zsok, editors, *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL'06*, Budapest, Hungary, Eotvos Lorand University, Faculty of Informatics, Department of Programming Languages and Compilers, Sept 4–6 2006c. To appear in Springer LNCS.

Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings Fifth International Conference on Functional Programming (ICFP'00)*, September 2000.

Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Proceedings ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 975 – 985, Portland, Oregon, USA, October 22-26 2006.

Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In S. Krishnamurthi and C.R. 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.

W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.