# An Introduction to iTasks: Defining Interactive Work Flows for the Web

Rinus Plasmeijer, Peter Achten, and Pieter Koopman

Radboud University Nijmegen, Netherlands
{rinus,P.Achten,pieter}@cs.ru.nl

**Abstract.** In these lecture notes we present 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 either humans or computers. The combinators that we propose support work flow patterns commonly found in commercial work flow systems. In addition, we introduce novel work flow patterns that capture real world requirements, but that can not be dealt with by current systems. Compared with most of these commercial systems, the iTask system offers several further 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: a work can be interrupted *and* subsequently directed to other workers for further processing. The iTask system has been constructed in the programming language Clean, making use of its generic programming facilities, and its iData toolkit with which interactive, thin-client, form-based web applications can be created. In all, iTasks are an excellent case of the expressive power of functional and generic programming.

## 1 Introduction

Work flow systems are automated systems that coordinate *tasks*. Parts of these tasks need to be performed by humans, other parts by computers. Automation of tasks in this way can increase the quality of the process, as the system keeps track of tasks, who is performing them, and in what order they should be performed. For this reason, there are many commercial work flow systems (such as Business Process Manager, COSA Workflow, FLOWer, i-Flow 6.0, Staffware, Websphere MQ Workflow, and YAWL) that are used in industry. If we investigate contemporary work flow systems from the perspective of a modern functional programming language such as Clean and Haskell, then there are a number of salient features that functional programmers are accustomed to that appear to be missing in work flow systems:

– Work flow situations are typically specified in a graphical language, instead of a textual language as typically used in programming languages. Functional programmers are keen on abstraction using higher order functions, generic programming techniques, rich type systems, and so on. Although experiments have been conducted to express these key features graphically (Vital [11], Eros [7]), functional programs are typically specified textually.
– Work flow systems mainly deal with control flow rather than data flow as in functional languages. As a result, they have focussed less on expressive type systems and analysis as has been done in functional language research.
– Within work flow systems, the data typically is globally known and accessible, and resides in databases. In functional languages, data is passed around between function arguments and results, and is therefore much more localized.

Given the above observations, we have posed the question if, and which, functional programming techniques can contribute to the expressiveness of work flow systems. In these lecture notes we show how web-applications with complex control flows can be constructed by presenting the iTask system: a set of combinators 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. The library covers all known *work flow patterns* that are found in contemporary commercial work flow tools [24]. The iTask toolkit extends these patterns with strong typing, higher-order functions and tasks, lazy evaluation, and a monadic style of programming. Its foundation upon the generic [1, 13] 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.

The iData toolkit [18, 19] is a high level library for creating interactive, thin client, web applications. For this reason it is well suited as an implementation platform for iTasks, because work flow systems are typically multi-user applications. As web browsers are ubiquitously available, it makes sense to implement a work flow system with web technology. The iData toolkit 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 [18], and a conference management system [17]. 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 is less suited for the specification of programs that require explicit *control flows*. 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.

A small, yet illustrative, exercise to handle work flow situations 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 programme this using iData?"

The key idea of an iData program is that it really is a collection of editors. From this point of view, the concept of a 'terminated' editor is not very natural. 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. These issues are tackled within the iTask system.

In these lecture notes, we assume that the reader is familiar with the functional programming language Clean [1] that is used in this paper.

The major part of this tutorial is devoted to presenting the iTask toolkit by means of a range of examples that demonstrate its major concepts in Sect. 2. We briefly discuss its implementation in Sect. 3. We end with related work in Sect. 4 and conclusions in Sect. 5. Appendix A gives the complete *api* of the iTask toolkit.

## 2 Overview of the iTask System

In this section we present the main concepts of the iTasks toolkit by means of a number of examples.

### 2.1 A Simple Example

With the iTask system, the work flow engineer specifies a work flow situation using combinators. This specification is interpreted by the iTask system. It presents to the work flow user a web browser interface that implements the given task. As a starter, we give the complete code of an extremely simple work flow, viz. that of a single, elemental, task in which the user is requested to fill in an integer form (see also Fig. 1):

```
module example

import StdEnv, iTasks

Start :: *World → *World
Start world = doHtmlServer (singleUserTask 0 True simple) world

simple :: Task Int
simple = editTask "Done" createDefault
```

[1] See http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf for the main differences between Clean and Haskell.

In line 3, the necessary modules are imported. `StdEnv` contains the standard functions, data structures, and type classes of Clean. `iTasks` imports the iTask system. The expression to be reduced as the main function is always given by the `Start` function. Because it has an effect on the external world, it is a function of type `*World` → `*World`. In Clean, effects on an environment of some type `T` are usually modeled with environment transformer functions of type (... `*T` → (... ,`*T`)). The *uniqueness attribute* `*` indicates that the environment is to be passed along in a single threaded way. This effect is similar to using the `IO` monad in Haskell, but uniquely attributed states are passed around explicitly. Violations against single threading are captured by the type system. In the iTask toolkit, tasks that produce values of some type `a` have type `Task a`:

```
:: Task a :== *TSt → (a,*TSt)
```

Here, `*TSt` is the unique and opaque environment that is passed along all tasks.

The iTasks library function `doHtmlServer` is a wrapper function that takes a function that generates an HTML page, and turns it into a real Clean application. The library function `singleUserTask` takes a work flow specification (here `simple`), provides it with a single user infrastructure, and computes the corresponding HTML page that reflects the current state of the work flow system. In Sect. 2.7 we encounter the `multiUserTask` function that dresses up multi-user work flow specifications. The infrastructure is a tracing option at the top of the window. It displays for each user her main tasks in a column. The selected main task is displayed next to this column.

The example work flow is given by `simple` (lines 8–9). It creates a single task with the library function `editTask` which has the following type:

```
editTask :: String a² → Task a |³ iData a
```

Its first argument is the label of the push button that the user can press to tell the system that this task is finished. Its second argument is the initial value that the task will display. When the user is done editing, hence after pressing the push button, the edited value is emitted by `editTask`. The type of `editTask` is overloaded. The type class `iData` collects all generic functions that are required for the iTask library to derive the proper instances.

```
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
```

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

[2] Note that in Clean the arity of functions is denoted explicitly by white-space between the arguments, hence the arity of `editTask` is two.

[3] Type class restrictions always occur at the end of a type signature, after a | symbol. The equivalent Haskell definition reads `editTask :: (iData a) => String -> a -> Task a`.

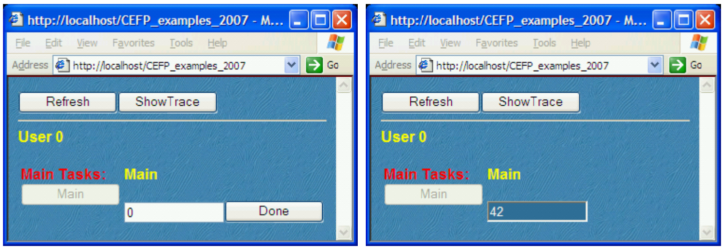

http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
0
Done
http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
42

**Fig. 1.** An elemental `Int` iTask when started (left) and finished (right)

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).

Note that the type of `simple` is more restrictive than that of `editTask`. This is because it uses the `createDefault` function which has signature:

```
createDefault :: d | gUpd{|⋆|} d
```

This function can generate a value for any type for which an instance of the generic `gUpd` function has been derived. Consequently, the most general type of `simple` is:

```
simple :: Task a | iData a
```

which is an overloaded type. Using this type makes the type of `Start` also overloaded, which is not allowed in Clean. There are basically two ways to deal with this: the first way is to replace `createDefault` with a concrete integer value, say 0:

```
simple = editTask "Done" 0
```

In that case, its type is `:: Task Int`. However, this is not very flexible: `simple` is now restricted to being an integer editing task. The second way, which was used in the original solution, is much more general: by only modifying the type signature of `simple`, but not its implementation, we can alter its editing task.

In the remainder of this tutorial, we skip the first three overhead lines of the examples, and show only the `Start` function.

## Exercises

**1.** *Getting started*
Download Clean for free at
    `http://clean.cs.ru.nl/`.
Install the Clean system. Also download the iTask system, which is available at
    `http://www.cs.ru.nl/~rinus/iTaskIntro.html`.
Follow the installation instructions *"iTasks - Do Read This Read Me.doc"* file that can be found in the `iTasks Examples` folder.

When done, start the Clean IDE. Create a new Clean implementation module, named *"exercise1.icl"*, and save it in a new directory of your choice. Create a new project, and confirm the suggested name and location by the Clean IDE (i.e. *"exercise1.prj"* in the newly created directory). Set the Environment to *"iTasks and iData and Util"*; otherwise the Clean compiler will complain about a plethora of missing files. Create, within the newly created directory, a subdirectory with the *same name*, and copy the file *"back35.jpg"* into it. This file can be found in any of the `Examples\iTasks Examples\` example directories of the iTask system. Use for each of the exercises a separate directory, to allow the system to create databases in such a way that they do not cause conflicts of name and type.

Enter in *"exercise1.icl"* the complete code that has been displayed in Sect. 2.1. Compile and run the application. If everything has gone well, you should see a console window that asks you to open your favorite browser and direct it to the given address. Follow this instruction, and you should be presented with your first iTask application that should be similar to Fig. 1.

## 2.2 Playing with Types

In this example we exploit the general purpose code of the previous example. The only modification we make is in line 8:

```
simple :: Task (Int,Real)                                        8.
```

Compiling and running this example results in a simple task for filling in a form of a pair of an `Int` and `Real` input field (see Fig. 2).

Now suppose that we want to do the same for a simple person administration form: we introduce a suitable record type, `Person`, defined as:

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

`HtmlDate` is a predefined algebraic data type for which an editor is created that allows the user to manipulate dates with separate editors for the year, month, and day. The only thing we need to do is to change the signature of `simple` into:

```
simple :: Task Person                                            8.
```

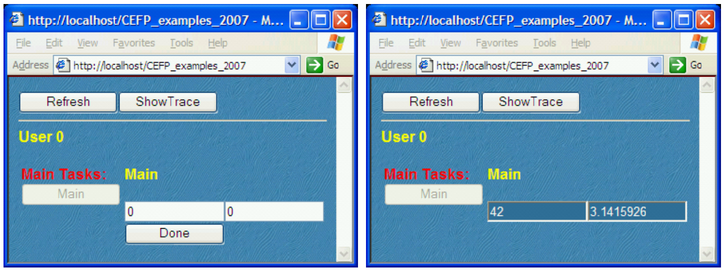
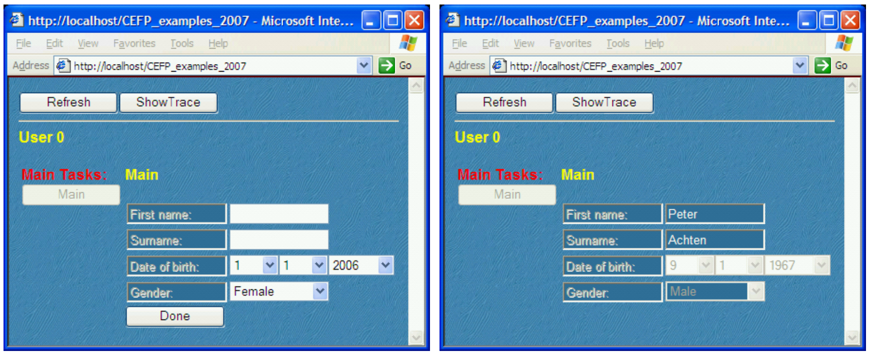

http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007 Go
Refresh ShowTrace
User 0
Main Tasks: Main
Main
0 0
Done
http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007 Go
Refresh ShowTrace
User 0
Main Tasks: Main
Main
42 3.1415926

**Fig. 2.** An (`Int`,`Real`) iTask when started (left) and finished (right)


http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
Done
http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main

| | |
|---|---|
| First name: | |
| Surname: | |
| Date of birth: | 1 1 2006 |
| Gender: | Female |

| | |
|---|---|
| First name: | Peter |
| Surname: | Achten |
| Date of birth: | 9 1 1967 |
| Gender: | Male |

**Fig. 3.** A Person iTask when started (left) and finished (right)

We intend to obtain an application such as the one displayed in Fig. 3.

Unfortunately, this does not compile successfully. A range of error messages is generated that complain that there are no instances of type `Person` for the generic functions that belong to the `iData` class. The reason that the (`Int`,`Real`) example does compile, and the `Person` example does not, is that for all basic types and basic type constructors such as (,), instances for these generic functions have already been asked to be derived. To allow this for `Person` and `Gender` values as well, we only need to be polite and ask for them:

```
derive gForm  Person, Gender
derive gUpd   Person, Gender
derive gPrint Person, Gender
derive gParse Person, Gender
derive gerda  Person, Gender
```

This example demonstrates that the code is very general purpose, and can be customized by introducing the desired type definitions, and politely asking the generic system to derive instance functions for the new types.

## Exercises

**2.** *Playing with a type of your own*
Create a new directory and subdirectory with the same name. Copy the *"exercise1.icl"* file into the new directory, and rename it to *"exercise2.icl"*. Copy the *"back35.jpg"* file into the subdirectory. Within the Clean IDE, open *"exercise2.icl"* and create a new project. Set the Environment to *"iTasks and iData and Util"*.

Define a new (set of) type(s), such as the `Person` and `Gender` given in Sect. 2.2, and create a `simple` editing task for it.

## 2.3 Playing with Attributes

In the previous examples an extremely simple, single-user, work flow was created. Even for such simple systems, we need to decide were to store the state of the application, and whether it should respond to every user editing action or only after an explicit *submit* action of the user. These aspects are *attributes* of tasks, and they can be set with the overloaded infix operator `<<@`:

```
class    (<<@) infixl 3 b :: (Task a) b → Task a
instance <<@  Lifespan        // default: Session
           ,  Mode            // default: Edit
           ,  GarbageCollect  // default: Collect
           ,  StorageFormat   // default: PlainString

:: Lifespan       = Session | Page    | Database | TxtFile | TxtFileRO | Temp
:: Mode           = Edit    | Submit | Display  | NoForm
:: GarbageCollect = Collect | NoCollect
:: StorageFormat  = PlainString | StaticDynamic
```

The `Lifespan` attribute controls the storage of the value of the iTasks: it can be stored persistently on the server side on disk in a relational database (`Database`) or in a file (`TxtFile` with `RO` read-only), it can be stored locally at the client side 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` attribute controls the *rendering* of the iTask: 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`). The `GarbageCollect` attribute controls whether the task tree should be garbage collected. This issue is described in more detail in Sect. 3.6. Finally, the `StorageFormat` attribute determines the way data is stored: either as a string (`PlainString`) or as a dynamic (`StaticDynamic`).

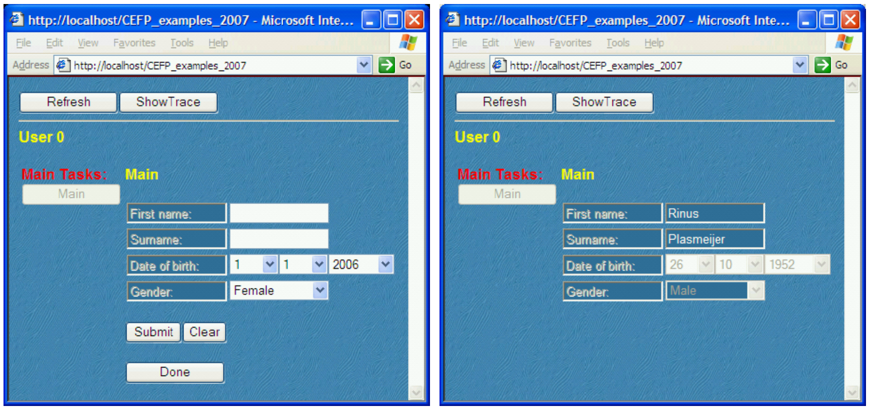
As an example, consider attributing the `simple` function of Sect. 2.1 in the following way (see Fig. 4):

```
simple :: Task Person                                            8.
simple = editTask "Done" createDefault <<@ Submit <<@ TxtFile    9.
```

### Exercises

**3.** *A persistent type of your own*
Create a new project for *"exercise3.icl"* as instructed in exercise 2.

Modify the code in such a way that it creates an application in which the most recently entered data is displayed, regardless whether the browser has been closed or not.


http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address
Go
Refresh
ShowTrace
Main Tasks:
Main
Main
Submit
Clear
Done
http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address
Go
Refresh
ShowTrace
Main Tasks:
Main
Main

http://localhost/CEFP_examples_2007

User 0

First name:
Surname:
Date of birth: 1 1 2006
Gender: Female

http://localhost/CEFP_examples_2007

User 0

First name: Rinus
Surname: Plasmeijer
Date of birth: 26 10 1952
Gender: Male

**Fig. 4.** A Person iTask attributed to be a 'classic' form editor

With these attributes, the application only responds to user actions after she has pressed the "Submit" button, and the value is stored in a text based database.

## 2.4 Sequencing with Monads: Wadler's Exercise

In the previous examples, the work flow consisted of a single task. One obvious combination of work flows is *sequential composition*. This has been realized within the iTask toolkit by providing it with appropriate instances of the *monadic* combinator functions:

```
(=>>) infix  1 :: (Task a) (a → Task b) → Task b | iCreateAndPrint b
(#>>) infixl 1 :: (Task a)      (Task b) → Task b
return_V       :: b                      → Task b | iCreateAndPrint b
```

where =>> is the *bind* combinator, and return_V the *return* combinator. Hence, ($m$ =>> $\lambda x \rightarrow n$) performs task $m$ if it should be activated, and passes its result value to $n$, which is only activated when required. The only task of (return_V $v$) is to emit value $v$. As usual, the shorthand combinator #>> that is defined immediately in terms of =>> ($m$ #>> $n \equiv m$ =>> $\lambda$ _ $\rightarrow n$) is provided as well. It is convenient to have a few alternative *return*-like combinators:

```
return_VF :: b [BodyTag] → Task b | iCreateAndPrint b
return_D  :: b           → Task b | iCreateAndPrint, gForm{|*|} b
```

With (return_VF $v$ *info*), customized information *info* given as HTML is shown to the application user. The algebraic type BodyTag maps one-to-one to the HTML-grammar. With (return_D $v$) the standard generic output of $v$ is used instead. It should be noted that unlike return_V these combinators are not true *return* combinators, as they do have an effect. Hence, the monad law $m$ =>> $\lambda v \rightarrow$ *return* $v = m$ is invalid when *return* is constructed with either return_VF or return_D.

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.

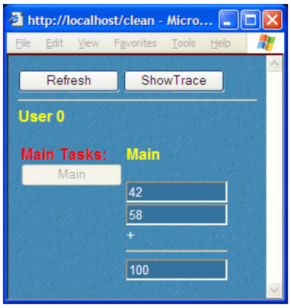
$(p \;?\!\gg t)$ is a task that displays prompt $p$ while task $t$ is running, whereas $(p \;!\!\gg t)$ displays prompt $p$ from the moment task $t$ is activated. Hence, a message displayed with !>> stays displayed once it has appeared, and a message displayed with ?>> disappears as soon as its argument task has finished.

```
(?>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(!>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
```

The prompt is defined as a piece of HTML.

With these definitions, the solution to Wadler's exercise becomes surprisingly simple.

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


http://localhost/clean - Micro...
File Edit View Favorites Tools Help
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main

42
58
+
100

## Exercises

**4.** *Hello!*
Create a work flow that first asks the name of a user, and then replies with "Hello" and the name of the user.

**5.** *To !>> or to ?>>*
Create a new project with the code of `sequenceITask`, and modify the !>> combinator into ?>>. What is the difference with the !>> combinator?

**6.** *Enter a prime number*
Create a work flow that uses the `<|` combinator (see Appendix A) to force the user to enter a prime number. A prime number $p$ is a positive integral number that can be divided only by 1 and $p$.

**7.** *Tearing `Person` apart*
In Sect. 2.2, a `Person` editing task was created with which the user edits complete `Person` values. Create a new work flow in which the user has to enter values for the fields one by one, i.e. starting with first name, and subsequently asking the surname, date of birth, and gender. Finally, the work flow should return the corresponding `Person` value.

**8.** *Adding numbers*
Create a work flow that first asks the user a positive (but not too great) integer number $n$, and subsequently have him enter $n$ values of type `Real` (use the `seqTasks`

combinator for this purpose – see Appendix A). When done, the work flow should display the sum of these values.

## 2.5 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. This is a single user task. The `Start` function is standard:

```
Start world = doHtmlServer (singleUserTask 0 True coffeemachine) world
```

The coffee machine is specified by the function `coffeemachine`. Before we give its definition, we first introduce a number of functions. 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*. Two iTask combinators will be used in `coffeemachine`:

```
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 choice buttons are aligned horizontally.

We are now ready to give the definition of `coffeemachine`:

```
coffeemachine :: Task (String,Int)                                              1.
coffeemachine                                                                   2.
= [Txt "Choose product:"]                                                       3.
  ?>> chooseTask [(p <+ ": " <+ c, return_V prod) \\ prod=:(p,c) ← products]    4.
  =>> λprod →                                                                   5.
  [Txt ("Chosen product: " <+ fst prod)]                                        6.
  ?>> pay prod (buttonTask "Thanks" (return_V prod))                            7.
where                                                                           8.
  products    = [("Coffee",100),("Tea",50)]                                     9.
  pay (p,c) t = buttonTask ("Pay " <+ c <+ " cents") t                          10.
```

First, the user is presented with a choice between coffee and tea (lines 3-4). Having chosen a product, the user is supposed to pay in a single step (line 7). In Sect. 2.6, we extend this to specifying a sub work flow for inserting coins in the coffee machine.

Besides `chooseTask`, the iTask toolkit offers a number of related task selection combinators:

```
chooseTaskV     :: [(String,Task a)] → Task  a  | iCreateAndPrint a
chooseTask_pdm  :: [(String,Task a)] → Task  a  | iCreateAndPrint a
mchoiceTasks    :: [(String,Task a)] → Task [a] | iCreateAndPrint a
```

`chooseTaskV` is the same as `chooseTask`, except that the choice buttons are aligned vertically. The same holds for `chooseTask_pdm`, except that it offers a pull down menu to select the desired task. Finally, a multiple choice of tasks is provided with `mchoiceTasks`.

**Exercises**

**9.** *Calculating on numbers*
In this exercise you extend the work flow in exercise 8 with the option to *add* (+), *subtract* (0), *multiply* (*), or *divide* (/) all numbers. Hence, if the input consists of numbers $x_1 \ldots x_n$, and the operator $\odot$, then the result should be computed as $(\ldots(x_1 \odot x_2) \odot \ldots x_{n-1}) \odot x_n$.

### 2.6 Repetition, Recursion and State: A Coffee Machine

The coffee machine in the previous example offers a single beverage, and terminates. In order to get more profit out of this machine, we extend it to a beverage vending machine that runs forever with the `foreverTask` combinator:

```
Start world = doHtmlServer (singleUserTask 0 True (foreverTask coffeemachine)) world
```

The signature of `foreverTask` is not surprising:

```
foreverTask :: (Task a) → Task a | iData a
```

It repeats its argument task infinitely many times.

The previous example abstracted from the paying task: the function call (`pay` $(p,c)$ $t$) offers a labeled action to pay the full amount of money $c$ for the chosen product $p$, 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 beverage at all. This is suitably modeled with a recursive task specification:

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

The iteration of inserting coins is modeled with the `repeatTask_Std` combinator:

```
repeatTask_Std :: (a → Task a) (a → Bool) a → Task a | iCreateAndPrint a
```

(`repeatTask_Std` $t\ p\ v_0$) executes a sequence of tasks $t\ v_0, t\ v_1, \ldots t\ v_n$ along a progressing sequence of values $v_0, v_1, \ldots v_n$. Here, $v_i$ is the result value of task ($t\ v_{i-1}$). The final result value, $v_n$, is also the result value of (`repeatTask_Std` $t\ p\ v_0$). For each $i < n$, we have $\neg(p\ v_i)$, and ($p\ v_n$). Hence, it works in a way similar to a *repeat t until p* control structure in imperative languages. The combinator `-||-` allows evaluation of two tasks in any order, and is finished as soon as either one task is finished. This is different from the behaviour of the task selection combinators that were discussed above in Sect. 2.5: they allow the user to select one task, which is then evaluated to the end. A similar combinator to `-||-` is `-&&-` which allows evaluation of two tasks in any order, but that finishes only if both tasks have finished.

The crucial combinator in this example is `newTask` (the implementation of `newTask` is discussed in Sect. 3.6). (`newTask` $l\ t$) promotes any user defined task $t$ to a proper iTask such that $t$ is only called when it is its turn to be activated. This is to prevent unwanted non-termination: although a *task description* is allowed to be defined recursively, at any stage of its execution, a workflow system is in some well defined state. 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. 2.5:

```
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 other current day work flow specification tools.

## Exercises

**10.** *A mini calculator*
Create a work flow that repeatedly offers the user the choice between:

– *First* enter a `Real` number $r$ and *next* choose an operator $\odot$ (as in exercise 9) and that returns $c \odot r$, with $c$ the current value; $c \odot r$ becomes the new current value.
– Return the current value $c$.

### 2.7 Multi-user Workflows

The solution to Phil Wadler's exercise that was given in Sect. 2.4, was a *single user* application. Work flow systems usually involve arbitrarily many users. This is supported by the iTask system.

```
multiUserTask :: !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
:: UserID :== Int
```

We identify users (using type synonym `UserID`) with integer index values $i \geq 0$. The wrapper function `multiUserTask` $n$ *trace* $t$ 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 of the HTML interface. 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 use the attribute setting operator `<<@` that was introduced in Sect. 2.3.

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 True 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. It should be noted that users have access to data only via the editor tasks. Because every task is always assigned to exactly one user, there is no danger of having multiple users attempting to update the same data item.

#### Exercises

**11.** *orTasks versus andTasks*
Create a work flow that first asks the user to enter a positive integral value $n$, and that subsequently creates $n$ tasks with `orTasks` and `andTasks`. The tasks are simple `buttonTasks`. Study the different behavior of `orTasks` and `andTasks`.

**12.** *Number guessing*
Create a 2-person work flow in which person 1 enters an integer value $1 \leq N \leq 100$, and who has person 2 guess this number. At every guess, the work flow should give feedback to person 2 whether the number guessed is too low, too high, or just right. In the latter case, the work flow returns `Just`$N$. Person 2 can also give up, in which case the work flow should return `Nothing`.
**Optional:** Person 1 is given the result of person 2, and has a chance to respond with a 'personal' message.

**13.** *Tic-tac-toe*
Create a 2-person work flow for playing the classic 'tic-tac-toe' game. The tic-tac-toe game consists of a $3 \times 3$ matrix. Player 1 places $\times$ marks in this matrix, and player 2 places $\circ$ marks. The first person to create a (horizontal, vertical, or diagonal) line of three identical marks wins. The work flow has to ensure that players enter marks only when it is their turn to do so.

---

## 2.8 Speculative Tasks and Multiple Users: Deadlines

Work flow systems need to handle time-related tasks: for instance, 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_V Nothing)            3.
             -||-                                               4.
        ([Txt ("Please finish task within" <+ time)]            5.
         ?>> (t =>> λv → return_V (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])) =>> λwho →        4.
  [Txt "How long do you want to wait?"]                                        5.
  ?>> editTask "SetTime" createDefault                          =>> λtime →    6.
  [Txt "Cancel delegated work if you get impatient:"]                          7.
  ?>> delegateTask who time t                                                  8.
      -||-                                                                     9.
      buttonTask "Cancel" (return_V Nothing) =>> check                         10.

  check (Just v)                                                               11.
  = [Txt ("Result of task: " <+ v)] ?>> buttonTask "OK" (return_V v)           12.
  check Nothing                                                                13.
  = [Txt "Task expired/canceled; do it yourself!"] ?>> buttonTask "OK" t       14.
```

The main task consists of selecting a user to whom a task $t$ should be delegated (lines 3–4), deciding how much time this user is given for this exercise (lines 5–6), and then delegating the task (line 8). We also model the situation that the current user gets impatient, and decides to abandon the delegated task (line 10). Either way, we know whether the task has succeeded and display the result and terminate (lines 11–12), or the current user has to do it herself (lines 13–14).

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. 2.6. 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 `simple` task from Sect. 2.1 with a concrete, non-overloaded type. This finalizes the example:

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

## Exercises

**14.** *Delayed task*
Create a work flow in which first an integral value $n$ is asked, and that subsequently waits $n$ seconds before it is finished. Use the `waitForTimerTask` combinator for this purpose.

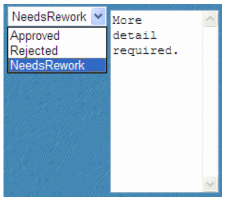
**15.** *Number guessing with deadline*
Use the delegation example of Sect. 2.8 in such a way that the number guessing game of exercise 12 can be created with it.

**16.** *Tic-tac-toe with deadline*
Use the delegation example of Sect. 2.8 in such a way that the tic-tac-toe game of exercise 13 can be created with it.

## 2.9 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, as shown above.

A reviewer inspects the product *v* that needs to be 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 createDefault) <<@ Submit)
               ,("Approved",return_V Approved)
               ,("Reject",   return_V Rejected)
               ]
```

Any task result that can be displayed, can also be subject to reviewing, hence the restriction to the generic iData class. The rendering is done with the iData toolkit function toHtml, which has signature:

```
toHtml :: a → BodyTag | gForm{|*|} a
```

Hence, (review *v*) displays *v* in the browser. The reviewer subsequently has to choose whether *v* should be reworked, and can comment on her decision, or *v* can be approved or rejected.

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_V (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 True (foreverTask reviewtask <<@ TxtFile)) world

reviewtask :: Task (Person,Review)
reviewtask = taskToReview 1 (createDefault, t)

t :: a → Task a | iData a
t v = [Txt "Fill in Form:"] ?>> editTask "TaskDone" v <<@ Submit
```

Note the high degree of parameterization and therefore re-useability of the code: `taskToReview` handles *any* task, and by providing *only* a type signature to `reviewtask` above, we get a form task for values of that type for free. Above, we have chosen the `Person` type. This is similar to the `simple` example that we started with in Sect. 2.1.

## 2.10 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. a task $t$ that already has been partially evaluated by someone can be shipped to some other user as (`TCl` $t$) who can continue to work on $t$.

```
:: TCl a      = TCl (Task a)
```

The proper generic functions have been specialized for type `TCl` 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.

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

*(stop* `-!>` *t)* 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 $t$. Note that, because `stop` is a type variable, any task can be used as the *stop* task.

As an example of using `-!>`, we present a highly dynamic case in which a worker pool of people can work on a given task. At any time, a worker can decide to stop working on that task, which should then be *continued* to work on by somebody else. Of course, the next person should not restart the task, but work with the partially evaluated task. The code of this example is given by delegate:

```
delegate :: (Task a) HtmlTime → Task a | iData a                            1.
delegate t time                                                             2.
=   [Txt "Choose persons you want to delegate work to:"]                    3.
    ?>> determineSet []          =>> λpeople →                              4.
    delegateToSomeone t people =>> λresult →                                5.
    return_D result                                                         6.
where                                                                       7.
    delegateToSomeone :: (Task a) [UserID] → Task a | iData a               8.
    delegateToSomeone t people = newTask "delegateToSet" doDelegate         9.
    where                                                                   10.
        doDelegate                                                          11.
         =  orTasks [ ( "Waiting for " <+ who                               12.
                      , who @:: buttonTask "I Will Do It" (return_V who)    13.
                      )                                                     14.
                    \\ who ← people                                         15.
                    ] =>> λwho →                                            16.
            who @:: stopTask -!> t =>> λ(stopped,TCl t) →                   17.
            if (isJust stopped) (delegateToSomeone t people) t              18.

        stopTask        = buttonTask "Stop" (return_V True)                 19.
```

The function delegate first creates a worker pool of people to choose from (line 3–4). All people are asked whether they want the task (line 5 and lines 8–18). 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 -||- (0 @:: stopTask) -||- timer time
timer time      = waitForTimerTask time #>> return_V True
```

Finally, creating the worker pool is a recursive work flow in which the user can select from candidates 1 upto $n$.

```
determineSet :: [UserID] → Task [UserID]                                    1.
determineSet people = newTask "determineSet" pool                           2.
where                                                                       3.
    pool   = [Txt ("Current set:" <+ people)]                               4.
             ?>> chooseTask                                                 5.
```

```
                [("Add Person", cancelTask person)
                ,("Finished",   return_V Nothing)
                ] =>> λresult →
            case result of
              (Just new) → determineSet (sort (removeDup [new:people]))
              Nothing    → return_V people
    person =  editTask "Set" (PullDown size (0,map toString [1..npersons]))
             =>> λwhomPD → return_V (Just (toInt (toString whomPD)))

cancelTask task = task -||- buttonTask "Cancel" (return_V createDefault)
```

**Exercises**

**17.** *Number guessing in a group*
In this exercise you extend the number guessing game of exercises 12 and 15 to a fixed set of persons $1 \ldots N$ in which user 0 determines who of $1 \ldots N$ is the next person to try to guess the number.

### 2.11 Summary

In this section we have given a range of examples to illustrate the expressive power of the iTask toolkit. We have not covered all of the available combinators. They can be found in Appendix A.

## 3 The iTasks Core System

The examples that have been given in Sect. 2 illustrate that iTask applications are multi-user applications that use mainly forms to communicate with end users, have various options to store data (client side and server side), and are highly dynamic. In general, implementing such kind of web applications is quite a challenge, especially when compared with desktop applications. One reason for this complication 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, web applications 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 is to adopt the Seaside approach [6]. 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 implementation principles of the iTasks system. For didactic reasons we restrain ourselves to a strongly simplified iTask *core system*. This core system is single user and has limited possibilities to manipulate tasks. The core system is already sufficient to create the solution to Wadler's exercise that was shown in Sect. 2.4. The full iTask toolkit that has been shown in Sect. 2 is built according to these principles.

## 3.1 iData as Primitive iTask in the Core System

In this section we describe how to lift iData elements to become iTasks. The iData library function `mkIData` creates an iData element. `mkIData` is an explicit `*HSt` environment transformer function. Its signature is:

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

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

`*HSt` contains the internal administration that is required for creating HTML pages and handling forms. Please consult [19] for details. `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[4] and the value of the iData element. 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 }
```

The `Lifespan` and `Mode` types were introduced in Sect. 2.3. They have the same meaning in the context of iData. 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
```

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

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 kept track of in the (`Form d`) record:

```
:: 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.

If we want to lift iData elements to the iTask domain, we need to include a concept of termination because this is absent in the iData framework: an iData application behaves as a set of iData elements that can be edited over and over again by the application user without predetermining some evaluation order. We ‘enhance’ iData elements with a concept of termination. We define a special function to make such a `taskEditor`. It is an ‘ordinary’ editor extended with a `Bool`ean 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 functions `simpleButton` and `mkStoreForm`:

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

(`simpleButton` *name l f*) creates an iData element whose appearance is that of a push button labeled *l*. It is identified with *name*. 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) | iData a          1.
taskEditor btnName label v hst                                                  2.
♯ (btn,  hst) = simpleButton btnLabel btnName (const True)  hst                 3.
♯ (done, hst) = mkStoreForm (Init,mkFormId storeLabel False) btn.value hst      4.
♯ (f, btnF)   = if done.value ((>@) Display,Br) (id,btn.form)                   5.
♯ (idata,hst) = mkIData (Init,f (mkFormId editLabel v)) hst                     6.
= (done.value,idata.value,idata.form ++ [btnF],hst)                             7.
where editLabel  = label +> "_Editor"                                           8.
      btnLabel   = label +> "_Button"                                           9.
      storeLabel = label +> "_Store"                                            10.
```

In the function `taskEditor` we create, as usual, an iData element for the value `v` (line 6). The `label` argument is used to create three additional identifiers for the value (editLabel), the button element (btnLabel), and the boolean storage element (storeLabel).

The trigger button (line 3) 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 (line 4). 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, and the iData element should be in Display mode, and otherwise the trigger button should be shown and the iData element should still be editable (line 5). In this way, the user is forced to continue with whatever user interface is created after pressing the trigger button.

The definition of taskEditor suggests that we need to extend the *HSt with some administration to keep track of the generated HTML, and identification labels for the editors that are lifted. This is what *TSt is for. It extends the *HSt environment 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
```

With this administration in place, we can use taskEditor to lift iData elements to elemental iTasks, viz. ones that allow the user to edit data and indicate termination of this elemental task. Recall that Task a was defined as (Sect. 2.1) *TSt → (a,*TSt):

```
editTask :: String a → Task a | iData a
editTask label a = doTask editTask`
where
  editTask` tst=:{tasknr,hst,html}
  # (done,na,nhtml,hst) = taskEditor label (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. A unique identifier is generated by this system (function doTask, which is explained below), which eliminates the shortcoming that was mentioned above. 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.

```
doTask :: (Task a) → Task a | iCreate a
doTask mytask         = doTask‘ o incTaskNr
where doTask‘ tst=:{activated, tasknr}
      | not activated = (createDefault, 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. 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 represent the numbering with an integer list, in reversed order.

```
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. 3.6).

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 `createDefault` 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 fully compositional, so the started tasks may actually consist of many subtasks of arbitrary complexity. When a task is started, it is either activated (or re-activated for further evaluation) or, 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 Basic Combinators of the Core System

As we have discussed in Sect. 2.4, sequential composition within the iTask toolkit is based on monads. Thanks to uniqueness typing we can freely choose how to thread the unique iTask state `*TSt`: either in explicit environment passing style or in implicit 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` should be opaque, and for her we provide a monadic interface. In the core system, their implementation is simply that of a state transformer function. Therefore, we do not include their code.

The implementation of the alternative `return_D` function is straightforward:

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

The implementation of the prompting combinators `?>>` and `!>>` is also not very difficult:

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

## 3.3 Reflection (Part I)

The behavior of the described core system is a combination of re-evaluating the application and having the enhanced 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 now automatically finds 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.

### 3.4 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.

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 createDefault)
          , 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" createDefault)
```

In t the user can work on sequenceITask (Sect. 2.4), 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. As we have stated in Sect. 2.6, 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 `createDefault`. 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`, until the predicate `p` holds.

```
repeatTask task p = t createDefault
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 the `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.

## 3.5 Reflection (Part II)

With the combinators presented above, iTasks can be composed as desired. As discussed in Sect. 3.4, one can imagine all kinds of additional combinators. For all well-known work flow patterns we have defined iTask combinators that mimic their behavior. They have been discussed in Sect. 2. 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 the user 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. Users should be able to work on several tasks in any order they want. 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.

The consequences for the implementation of the core system are described next.

### 3.6 The Actual iTask Implementation

In this section we discuss the most interesting aspects of the actual implementation by building on the core system.

**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. 2.7, users can be assigned to tasks with the operators `@:` and `@::`. We give the HTML accumulator within the `TSt` environment (Sect. 3.1) 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 label of 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. These two alternatives are used to distinguish between output for a given user, and other output. 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 2.8). This effects the implementation of all choice combinators: one has to remember which task was chosen to avoid race conditions.

**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. In a naive implementation, this would lead to a linear increase in time per user action on the work flow, which is clearly 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. 3.4.

```
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") (False,createDefault) <@ Session    12.
```

A storage is associated with task $t$ (line 3) that initially has a default value (line 12). 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 "redex".

**Garbage Collection of iData Objects.** The optimization described above prevents 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?". Another reason is that one wants to remember a result of a task, but not of any of its subtasks. We have therefore included variants of certain combinators in the iTask library, such as `repeatTaskGC` and `newTaskGC` which automatically take care of the garbage collection of 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.

**Higher-Order Tasks.** A distinctive feature of the iTask toolkit is the ability to communicate higher-order tasks that have been partially evaluated (Sect. 2.10). 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 [24]). 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 ($p$-!>$t$) 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.

## 4 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 Haskell CGI library [16]; the Curry approach [12]; writing XML applications [9] in *SMLserver* [8]. One sophisticated system is WASH/CGI by [23], 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 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 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 [5] and Hop [22]. 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 [2]. It has been improved upon by means of Powerforms [3], used in the <bigwig> project [4]. 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. Their and our approach make good use of the type system. 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, in his arrow framework [14]. Queinnec states that "A browser is a device that can invoke continuations multiply/simultaneously" [21]. Graunke *et al* [10] have explored continuations as one of three functional compilation techniques to transform sequential interactive programs to CGI programs. The Seaside [6] 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 by construction a single user system.

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.

Workflow systems are distributed software systems, and as such can also be implemented using a programming language with support for distributed computing such as D-Clean [25], GdH [20], Erlang, and Java. iTasks, on the other hand, makes effective use of the distributed nature of the web: web browsers act as distributed rendering resources, and the server controls what gets displayed where and when. Furthermore, the interactive components are created in a type-directed way, which makes the code concise. There is no need to program the data flow between the participating users, again reducing the code size.

Our combinator library has been inspired by the comprehensive analysis of work flow patterns of over more than 30 contemporary commercial work flow systems [24]. 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 combinatorial approach such as the 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. 2.6 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.

## 5 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.

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 (see Appendix A). 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 extended 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. 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 [15].

## Acknowledgements

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, and Wil van der Aalst for commenting on the difference between the combinator approach and contemporary work flow specification languages.

## References

1. Alimarine, A.: Generic Functional Programming - Conceptual Design, Implementation and Applications. PhD thesis, University of Nijmegen, The Netherlands (2005) ISBN 3-540-67658-9
2. Atkins, D., Ball, T., Benedikt, M., Bruns, G., Cox, K., Mataga, P., Rehor, K.: Experience with a Domain Specific Language for Form-based Services. In: Usenix Conference on Domain Specific Languages (October 1997)
3. Brabrand, C., Møller, A., Ricky, M., Schwartzbach, M.: Powerforms: Declarative client-side form field validation. World Wide Web Journal 3(4), 205–314 (2000)
4. Brabrand, C., Møller, A., Schwartzbach, M.: The <bigwig> Project. ACM Transactions on Internet Technology (TOIT) (2002)
5. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709. Springer, Heidelberg (2007)
6. Ducasse, S., Lienhard, A., Renggli, L.: Seaside - A Multiple Control Flow Web Application Framework. In: Ducasse, S. (ed.), Proceedings ESUG 2004 International Conference – Research Track, volume Technical Report IAM-04-008, pp. 231–254. Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, November 7 (2004)
7. Elliot, C.: Tangible Functional Programming. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), Freiburg, Germany, October 1–3, pp. 59–70. ACM, New York (2007)
8. Elsman, M., Hallenberg, N.: Web programming with SMLserver. In: Dahl, V., Wadler, P. (eds.) PADL 2003. Springer, Heidelberg (2003)
9. Elsman, M., Larsen, K.F.: Typing XHTML Web applications in ML. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 224–238. Springer, Heidelberg (2004)
10. Graunke, P., Krishnamurthi, S., Findler, R.B., Felleisen, M.: Automatically Restructuring Programs for the Web. In: Feather, M., Goedicke, M. (eds.) Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE 2001). IEEE CS Press, Los Alamitos (2001)
11. Hanna, K.: A Document-Centered Environment for Haskell. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015. Springer, Heidelberg (2006)
12. Hanus, M.: High-Level Server Side Web Scripting in Curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
13. Hinze, R.: A new approach to generic functional programming. In: The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Massachusetts, Boston, pp. 119–132 (January 2000)
14. Hughes, J.: Generalising Monads to Arrows. Science of Computer Programming 37, 67–111 (2000)
15. Jansen, J., Koopman, P., Plasmeijer, R.: Efficient Interpretation by Transforming Data Types and Patterns to Functions. In: Nilsson, H. (ed.) Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, April 19-21, 2006, pp. 157–172. The University of Nottingham (2006)
16. Meijer, E.: Server Side Web Scripting in Haskell. Journal of Functional Programming 10(1), 1–18 (2000)
17. Plasmeijer, R., Achten, P.: A Conference Management System based on the iData Toolkit. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 108–125. Springer, Heidelberg (2007)

18. Plasmeijer, R., Achten, P.: iData For The World Wide Web - Programming Interconnected Web Forms. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945. Springer, Heidelberg (2006)
19. Plasmeijer, R., Achten, P.: The Implementation of iData - A Case Study in Generic Programming. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 106–123. Springer, Heidelberg (2006)
20. Pointon, R., Trinder, P., Loidl, H.: The Design and Implementation of Glasgow distributed Haskell. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011. Springer, Heidelberg (2001)
21. Queinnec, C.: The influence of browsers on evaluators or, continuations to program web servers. In: Proceedings Fifth International Conference on Functional Programming (ICFP 2000) (September 2000)
22. Serrano, M., Gallesio, E., Loitsch, F.: 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), Portland, Oregon, USA, pp. 975–985, October 22-26 (2006)
23. Thiemann, P.: WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Form. In: Krishnamurthi, S., Ramakrishnan, C. (eds.) PADL 2002. LNCS, vol. 2257, pp. 192–208. Springer, Heidelberg (2002)
24. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane (2002)
25. Zsók, V., Hernyák, Z., Horváth, Z.: Distributed Pattern Design in D-Clean. In: Central-European Functional Programming School, CEFP 2005, oldal, vol. 33 (2005), `http://plc.inf.elte.hu/cefp/download/dclean_dbox_lecturenotes.pdf`

## A iTask Toolkit

This is the complete *api* of the iTask toolkit.

```
definition module iTasks

// iTasks library for defining interactive multi-user workflow tasks (iTask) for the web
// defined on top of the iData library

// ©iTask & iData Concept and Implementation by Rinus Plasmeijer, 2006,2007 - MJP
// Version 1.0 - april 2007 - MJP
// This library is still under construction - MJP

import iDataSettings, iDataButtons

derive gForm   Void
derive gUpd    Void, TCl
derive gPrint  Void, TCl
derive gParse  Void
derive gerda   Void
```

```
:: *TSt                         // task state
:: Task a       :== St *TSt a   // an interactive task
:: Void         = Void          // for tasks returning non interesting results,
                                // won't show up in editors either

/* Initiating the iTask library: to be used with an iData server wrapper!
startTask       :: start iTasks beginning with user with given id, True if trace allowed
                     id < 0  : for login purposes.
startNewTask    :: same, lifted to iTask domain, use it after a login ritual
singleUserTask  :: start wrapper function for single user
multiUserTask   :: start wrapper function for user with indicated id with option to switch
                     between [0..users - 1]
multiUserTask2  :: same, but forces an automatic update request every (n minutes, m seconds)
*/
startTask       ::                 !Int !Bool !(Task a) !*HSt → (a,[BodyTag],!*HSt) | iCreate a
startNewTask    ::                 !Int !Bool !(Task a)       → Task a       | iCreateAndPrint a

singleUserTask  ::                 !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
multiUserTask   ::                 !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
multiUserTask2  :: !(!Int,!Int) !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a

/* Setting options for any collection of iTask workflows
(<<@)           :: set iData attribute globally for indicated (composition of) iTasks
*/
class (<<@) infix 3 b :: (Task a) b → Task a
:: GarbageCollect = Collect | NoCollect

instance <<@        Lifespan              // default: Session
                ,   StorageFormat         // default: PlainString
                ,   Mode                  // default: Edit
                ,   GarbageCollect        // deafult: Collect

defaultUser     :== 0                     // default id of user

// Here follow the iTask combinators:

/* promote any iData editor to the iTask domain
editTask        :: create a task editor to edit a value of given type,
                     and add a button with given name to finish the task
*/
editTask        :: String a                   → Task a   | iData a

/* standard monadic combinators on iTask
(=>>)           :: for sequencing: bind
(#>>)           :: for sequencing: bind, but no argument passed
return_V        :: lift a value to the iTask domain and return it
*/
(=>>) infix  1  :: (Task a) (a → Task b)      → Task b   | iCreateAndPrint b
(#>>) infixl 1  :: (Task a) (Task b)          → Task b
return_V        :: a                          → Task a   | iCreateAndPrint a
```

```
/* prompting variants
(?>>)           :: prompt as long as task is active but not finished
(!>>)           :: prompt when task is activated
(<|)            :: repeat task as long as predicate does not hold, give error otherwise
return_VF       :: return the value and show the HTML  code specified
return_D        :: return the value and show it in iData  display format
*/
(?>>) infix 5   :: [BodyTag] (Task a)           → Task a   | iCreate a
(!>>) infix 5   :: [BodyTag] (Task a)           → Task a   | iCreate a
(<|)  infix 6   :: (Task a) (a → .Bool, a → [BodyTag])
                                                → Task a   | iCreate a
return_VF       :: a [BodyTag]                  → Task a   | iCreateAndPrint a
return_D        :: a                            → Task a   | gForm {|*|}, iCreateAndPrint a

/* Assign tasks to user with indicated id
(@:)            :: will prompt who is waiting for task with give name
(@::)           :: same, default task name given
*/
(@:)  infix 3   :: !(!String,!Int) (Task a)     → Task a   | iCreateAndPrint a
(@::) infix 3   ::            !Int (Task a)     → Task a   | iCreate a

/* Handling recursion and loops
newTask         :: use the to promote a (recursively) defined user function to as task
foreverTask     :: infinitely repeating Task
repeatTask      :: repeat Task until predict is valid
*/
newTask         :: !String (Task a)             → Task a   | iData a
foreverTask     ::         (Task a)             → Task a   | iData a
repeatTask_Std  :: (a → Task a) (a → Bool) → a → Task a   | iCreateAndPrint a

/*  Sequencing Tasks:
seqTasks        :: do all iTasks  one after another, task completed when all done
*/
seqTasks        :: [(String,Task a)]            → Task [a] | iCreateAndPrint a

/* Choose Tasks
buttonTask      :: Choose the iTask  when button pressed
chooseTask      :: Select one iTask  with button, buttons horizontally displayed
chooseTaskV     :: Select one iTask  with button, buttons vertically displayed
chooseTask_pdm  :: Select one iTask  with pull down menu
mchoiceTask     :: Select several iTasks  with marked check boxes
*/
buttonTask      ::  String (Task a)     → Task a          | iCreateAndPrint a
chooseTask      :: [(String,Task a)]    → Task a          | iCreateAndPrint a
chooseTaskV     :: [(String,Task a)]    → Task a          | iCreateAndPrint a
chooseTask_pdm  :: [(String,Task a)]    → Task a          | iCreateAndPrint a
mchoiceTasks    :: [(String,Task a)]    → Task [a]        | iCreateAndPrint a

/* Do m Tasks parallel / interleaved and FINISH as soon as SOME Task completes:
orTask          :: both iTasks  in any order, completion when first done
(-||-)          :: same, now as infix combinator
orTask2         :: both iTasks  in any order, completion when first done
```

```
orTasks          :: all iTasks  in any order, completion when first done
*/
orTask           :: (Task a,  Task a)     → Task a              | iCreateAndPrint a
(-||-) infixr 3  :: (Task a) (Task a)     → Task a              | iCreateAndPrint a
orTask2          :: (Task a,  Task b)     → Task (EITHER a b)   | iCreateAndPrint a
                                                                & iCreateAndPrint b
orTasks          :: [(String, Task a)]    → Task a              | iData a

/* Do Tasks parallel / interleaved and FINISH when ALL Tasks done:
andTask          :: both iTasks  in any order, completion when both done
(-&&-)           :: same, now as infix combinator
andTasks         :: all iTasks  in any order, completion when all done
andTasks_mu      :: assign task to indicated users, task completed when all done
*/
andTask          :: (Task a,  Task b)     → Task (a,b)          | iCreateAndPrint a
                                                                & iCreateAndPrint b
(-&&-) infixr 4  :: (Task a) (Task b)     → Task (a,b)          | iCreateAndPrint a
                                                                & iCreateAndPrint b
andTasks         :: [(String,Task a)]     → Task [a]            | iCreateAndPrint a
andTasks_mu      :: String [(Int,Task a)] → Task [a]            | iData a

/* Time and Date management:
waitForTimeTask :: Task is done when time has come
waitForTimerTask:: Task is done when specified amount of time has passed
waitForDateTask :: Task is done when date has come
*/
waitForTimeTask :: HtmlTime               → Task HtmlTime
waitForTimerTask:: HtmlTime               → Task HtmlTime
waitForDateTask :: HtmlDate               → Task HtmlDate

/* Experimental department
   Will not work when the tasks are garbage collected to soon !!

-!>             :: a task, either finished or interrupted (by completion of the first task)
                   is returned in the closure if interrupted, the work done so far is
                   returned(!) which can be continued somewhere else
channel         :: splits a task in respectively a sender task closure and receiver task
                   closure; when the sender is evaluated, the original task is evaluated as
                   usual; when the receiver task is evaluated, it will wait upon completion
                   of the sender and then gets its result;
                   Important:
                      Notice that a receiver will never finish if you don't activate the
                   corresponding receiver somewhere.
closureTask     :: The task is executed as usual, but a receiver closure is returned
                   immediately. When the closure is evaluated somewhere, one has to wait
                   until the task is finished. Handy for passing a result to several
                   interested parties.
closureLzTask   :: Same, but now the original task will not be done unless someone is asking
                   for the result somewhere.

*/
:: TCl a          = TCl (Task a)
```

```
(-!>) infix 4   :: (Task stop) (Task a) → Task (Maybe stop,TCl a) | iCreateAndPrint stop
                                                                  & iCreateAndPrint a
channel         :: String (Task a)      → Task (TCl a,TCl a)      | iCreateAndPrint a
closureTask     :: String (Task a)      → Task (TCl a)            | iCreateAndPrint a
closureLzTask   :: String (Task a)      → Task (TCl a)            | iCreateAndPrint a

/* Operations on Task state
taskId          :: id assigned to task
userId          :: id of application user
addHtml         :: add HTML  code
*/
taskId          :: TSt → (Int,TSt)
userId          :: TSt → (Int,TSt)
addHtml         :: [BodyTag] TSt → TSt

/* Lifting to iTask  domain
(*>>)           :: lift functions of type (TSt→ (a,TSt)) to iTask  domain
(@>>)           :: lift functions of (TSt→ TSt) to iTask  domain
appIData        :: lift iData  editors to iTask  domain
appHSt          :: lift HSt domain to TSt domain, will be executed only once
appHSt2         :: lift HSt domain to TSt domain, will be executed on each invocation

*/
(*>>) infix 4   :: (TSt → (a,TSt)) (a → Task b) → Task b
(@>>) infix 4   :: (TSt → TSt) (Task a)         → Task a
appIData        :: (IDataFun a)                 → Task a      | iData a
appHSt          :: (HSt → (a,HSt))              → Task a      | iData a
appHSt2         :: (HSt → (a,HSt))              → Task a      | iData a

/* Controlling side effects
Once           :;  task will be done only once, the value of the task will be remembered
*/
Once           :: (Task a)                    → Task a      | iData a
```