Specifying Interactive Work Flows for the Web

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort
Radboud University Nijmegen, Netherlands
{rinus,P.Achten,pieter,b.lijnse,thomas}@cs.ru.nl

Abstract. In these lecture notes we present the iTask system: a set of combinators to specify workflows in a pure functional language at a very high level of abstraction. Workflow 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 workflow patterns commonly found in commercial workflow systems. In addition, we introduce novel workflow 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 workflows can be specified, and last but not least, the specification is used to generate an executable web-based multi-user workflow application. With the iTask system, useful workflows 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

Workflow 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 workflow 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 workflow 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 workflow systems:

- Workflow 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.

- Workflow 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 workflow 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 workflow 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 workflows. It is built on top of the iData toolkit, and both can be used within the same program. The library covers all known workflow patterns that are found in contemporary commercial workflow tools [21]. 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 [13, 1] features of the iData toolkit yields compact, robust, reusable and understandable code. Workflows are defined on a very high level of abstraction. It truly is an executable specification, as much is done and generated automatically.

As a running example, we will study the architecture of a conference management (CM) systems, and implement a small prototype. CM is a good case study of a workflow because it controls the activities of people with various roles, such as program chairs and program committee members. It is also challenging because many of these activities run in parallel, and the system should not hamper the activities of the workers of the 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 and exercises 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 Programming Workflows with iTasks

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

```
module example
import StdEnv, StdiTasks
Start :: *World → *World
Start world = single0UserTask [] simple world
simple :: Task Int
simple = editTask "Done" createDefault
```

![Fig. 1. An elementary Int iTask when started.](image)

In line 3, the required modules are imported. StdEnv contains the standard func-
tions, data structures, and type classes of Clean. StdiTasks imports the iTask sys-
tem. 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 usu-
ally 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 library function `singleUserTask` takes a workflow specification (here `simple`), provides it with a single worker infrastructure, and computes the corresponding HTML page that reflects the current state of the workflow system. In Sect. 2.8 we encounter the `multiUserTask` function that dresses up multi-user workflow 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 workflow 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 a | gForm { | }, iCreateAndPrint, iParse, iSpecialStore a
class iCreateAndPrint a | iCreate, iPrint a
class iCreate a | gUpd { | } a
class iPrint a | gPrint { | } a
class iParse a | gParse { | } a
class iSpecialStore a | TC a
```

They can be used for values of any type to automatically create an HTML form (`gForm`), to handle the effect of any edit action with the browser including the creation of default values (`gUpd`), to print or serialize any value (`gPrint`), to parse or de-serialize any value (`gParse`), 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 :: a | gUpd[*] a
```

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:

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

\[\text{Type class restrictions always occur at the end of a type signature, after a `|` symbol. The equivalent Haskell definition reads `editTask :: (iData a) => String → a → Task a`.}\]
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
To get started quickly we have compiled a convenient distribution package which contains the latest Clean system for windows, all the additional iTask libraries and the examples and exercises for AFP2008.

Download this distribution at:

http://clean.cs.ru.nl/download/clean22/
windows/Clean2.2-iTasks-AFP2008.zip

Unpack this zip archive and follow the instructions in the “iTasks - Do Read This Read Me.doc” file that can be found in the root folder of the archive.

When done, start the Clean IDE. Open the project file of the CM system case study, CM.prj. The project window should now be filled with all module names that the CM system uses. Compile and run the application. If everything is well, you should see a console window that asks you to open your browser and direct it to the given address. Follow this instruction, and you should be presented with the login screen of the CM system.

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:

```haskell
simple :: Task (Int,Real)
```

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).

In the CM case study, users are populated with program chairs (Chair) and program members (PC). We can define a record type, User, defined as:

```haskell
:: User = { login :: Login,
             email :: TextInput,
             role :: Role
         }
:: Role = Chair | PC
```
Login is a predefined algebraic data type for which an editor is created that allows the user to use a standard login input box for entering the account name and hidden password entry box. In order to use it, you need to include iTaskUtil to the import list at line 3. TextInput is also a predefined type for entering basic data (integers, reals, strings), and give the input box a desired width. In order to obtain an editor for User values we need to change the signature of simple into:

\[
\text{simple :: Task User}
\]

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 User, Role, and Login for the generic iData class functions. The reason that the (Int,Real) example does compile, and the User 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 User, Role, and Login values as well, we only need to be polite and ask for them:

```plaintext
derive gForm User, Role, Login
derive gUpd User, Role, Login
derive gPrint User, Role, Login
derive gParse User, Role, Login
```

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. Copy the “exercise1.icl” file into the new directory, and rename it to “exercise2.icl”. Within the Clean IDE, open “exercise2.icl” and create a new project. Set the Environment to “iTasks”.

Define a new (set of) type(s), such as the User and Role 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, workflow was created. Even for such simple systems, we need to decide where 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 `<<@`:

```plaintext
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 | DataFile
  | Client | 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). A novel attribute is to enforce client side evaluation, with the Client attribute. 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 Edited 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 Displayed 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 User

simple = editTask "Done" createDefault <<< Submit <<< TxtFile
```

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.

![Fig. 4. A User iTask attributed to be a ‘classic’ form editor.](image)

Editor tasks created with editTask allow the worker to enter any value, provided it is of the corresponding type of the editor. For many cases, this is sufficient. However, sometimes you wish to impose constraints on the edited values that cannot be expressed via the type system of Clean. Examples are editor tasks for even Int values, User values in which sensible values have been entered, and so on. For this purpose a predicate of type \( \mathbb{a} \rightarrow (\text{Bool}, \text{HtmlCode}) \) can be used to test the value of type \( \mathbb{a} \) that is produced by the worker. If the value is correct,
then the predicate returns True, otherwise it returns False and some explanation in the form of HtmlCode. The function editTaskPred does this:

\[
\text{editTaskPred} :: !a !\{(a \rightarrow (\text{Bool}, \text{HtmlCode}))\} \rightarrow \text{Task a} | \text{iData a}
\]

The worker can edit values as usual, but these are checked with the predicate when the user submits the value. If the predicate does not hold, then the error message is displayed. Only if it holds, then then the editor task is finished, and the new value is propagated. Consider the following example from the CM case study:

simple :: Task User
simple = editTaskPred \{createDefault & User.email = emptyTextInput\} checkUser

checkUser :: User \rightarrow (\text{Bool}, \text{HtmlCode})
checkUser \{User | login \rightarrow \{\text{loginName, password, email}\}\}
| loginName == "" = (False, [Txt "You need to enter a login name"])
| password == PasswordBox "" = (False, [Txt "You need to enter a password"])
| fromTextInput email == "" = (False, [Txt "You need to enter an email address"])
| otherwise = (True, [])

In this example, the predicate check checks a few simple properties of User values. Fig. 5 shows this editor task in action.

![Fig. 5. A User iTask, now validating entered values.](image)

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.

2.4 Simple database access

In the previous section we have shown that the programmer can decide where the state of a task editor is stored. This feature of editors can also be used to create a module for simple database access, which is named \texttt{iTasksDB.dcl}. We summarize the key ingredients of this module:

\textbf{definition module} \texttt{iTasksDB}\linebreak
\hspace{1em}:: DBid a

\hspace{1em}mkDBid :: !String !Lifespan \to DBid a
\hspace{1em}readDB :: !(DBid a) \to Task a \mid iData a
\hspace{1em}writeDB :: !(DBid a) !a \to Task a \mid iData a

\texttt{(mkDBid name Database)} returns a database identifier only if \texttt{name} is a proper file name, because the read and write operations will be performed on disk. \texttt{(mkDBid name lifespan)} (with \texttt{lifespan \neq Database}) accepts any \texttt{name}. \texttt{(readDB name)} reads the current content of the identified database, and returns \texttt{createDefault} otherwise. \texttt{(writeDB name v)} sets the current content of the identified database to \texttt{v} and returns that value as well.

Suppose we wish to set up a User administration in the CM case study. We can introduce the following functions for that purpose (these are very similar to those in module \texttt{CMDatabase.icl}):

\hspace{1em}usersId :: DBid \texttt{[User]}
\hspace{1em}usersId = mkDBid "Users" TxtFile

\hspace{1em}readUsersDB :: Task \texttt{[User]}
\hspace{1em}readUsersDB = readDB usersId

\hspace{1em}writeUsersDB :: ([User] \to Task [User])
\hspace{1em}writeUsersDB = writeDB usersId

We use them in the following section.

2.5 Sequencing with monads

In the previous examples, the workflow consisted of a single task. One obvious combination of workflows is \textit{sequential composition}. This has been realized within the iTask toolkit by providing it with appropriate instances of the \textit{monadic} combinator functions:

\texttt{(<\Rightarrow>) infix 1 :: (Task a) (a \to Task b) \to Task b \mid iCreateAndPrint b}
\texttt{(\Rightarrow\Leftarrow) infixl 1 :: (Task a) (Task b) \to Task b}
\texttt{return_V :: b \to Task b \mid iCreateAndPrint b}
where \( \Rightarrow \) is the bind combinator, and \( \text{return}_V \) the return combinator. Hence, \((m \Rightarrow \lambda x \rightarrow n)\) 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 \((\text{return}_V v)\) is to emit value \(v\). As usual, the shorthand combinator \(\Rightarrow\) that is defined immediately in terms of \((m \Rightarrow n \equiv m \Rightarrow \lambda \_ \rightarrow n)\) is provided as well.

As an example, we can extend the User administration that was given in Sect. 2.4 with a function to prepend a single user to the administration:

```haskell
addUserDB :: User → Task [User]
addUserDB user = readUsersDB ≫> λ users → writeUsersDB [user:users]
```

It is convenient to have a few alternative return-like combinators:

```haskell
return_VF :: b [BodyTag] → Task b | iCreateAndPrint b
return_D :: b → Task b | iCreateAndPrint, gForm ![ ⋆ ] b
```

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

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 ?\(\gg\) stays displayed once it has appeared, and a message displayed with !\(\gg\) disappears as soon as its argument task has finished.

```haskell
(??) 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.

The example at the end of Sect. 2.3 defined a User editor task with the predicate checkUser. With a minor change, it also checks whether the entered user value has a fresh name:

```haskell
checkUser :: [User] → (Bool, HtmlCode)
checkUser users {User | login={loginName,password},email}
| loginName == "" = (False, [Txt "You need to enter a login name"])
| password == PasswordBox "" = (False, [Txt "You need to enter a password"])
| fromTextInput email == "" = (False, [Txt "You need to enter an email address"])
| isMember loginName userNames = (False, [Txt "This login name already exists"]) // new
| otherwise = (True,[])
where
    userNames = [n \ \ {User | login={loginName=n}} ← users]
```

With this predicate we can create a User editor task that tests for existing user names:
addUserForm :: Task User
addUserForm = readUsersDB ⇒> users →
              msg ?⇒ editTaskPred {createDefault & User.email = emptyTextInput}
                               (checkUser users)
    where msg = [Txt "Please enter a username and password for the new user:"]

A sensible task for the program chair is to add users to the CM system. This can be expressed as:

addUser :: Task Void
addUser = addUserForm ⇒> user →
           addUserDB user ?⇒
           endMsg ?⇒ button "Ok"
    where endMsg = [Txt "A new user has been added"]

Exercises

4. **Hello!**
Create a workflow that first asks the name of a user, and then replies with “Hello” and the name of the user.

5. **To ∈ or to ⇒**
Open the CM system project file, and find the function addUser (in the main module CM.icl). Alter the ∈⇒ combinator into ⇒. Compile and re-run the application. What is the effect of this change?

6. **Enter a prime number**
Create a workflow 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 User apart**
In Sect. 2.2, a User editor task was created with which complete User values can be edited. Create a new workflow in which the user has to enter values for the fields one by one, i.e. starting with the login name, and subsequently asking the password, email and role. Finally, the workflow should return the corresponding User value.

8. **Adding users**
Create a workflow that first asks the user a positive (but not too great) integer number n, and subsequently have him enter n values of type User (use the seqTasks combinator for this purpose – see Appendix A). When done, the workflow should display the names of these users.
2.6 Sequence and choice: breakable work

The monadic combinators presented in the previous section are useful for sequential composition. Obviously, realistic workflows also require choice, and this is provided by the iTask system with the following basic combinator:

\[ (\sim \sim) \text{ infixr 3 : : } \text{!(Task a)} \text{ !(Task a)} \rightarrow \text{Task a | iData a} \]

\((t_1 \sim \sim t_2)\) is a task that terminates as soon as either \(t_1\) or \(t_2\) has terminated, or both.

The combination of monadic composition and choice leads to a number of useful derived combinators. Some of them have been defined in module CMCombinators.icl in the case study. Here we discuss some of them.

Tasks constructed with the monadic combinators rigidly force the worker to perform the given tasks in the prescribed order, and terminate only when the very last task has been performed. For real world cases, this is sometimes too restrictive: we want to model the fact that a worker can choose to abort her work. The break combinator models this behavior:

\[ \text{break :: (Task a)} \rightarrow \text{Task (Maybe a | iData a)} \]

\(\text{break taska} = (\text{taska} =\gg> \text{return}_V \circ \text{Just})\)

\(\sim \sim (\text{cancel} =\gg> \text{return}_V \text{Nothing})\)

\((\text{break} t)\) is a task that performs \(t\), and if that has terminated and yielded a value \(v\) yields \((\text{Just} v)\). However, at any time before finishing \(t\), the worker also has the choice to perform the cancel task, and return \(\text{Nothing}\) instead.

Together with \text{button}, \text{ok}, and \text{void}, \text{cancel} forms another group of tiny, but useful combinators:

\[ \text{button :: String } \rightarrow \text{Task Void} \]

\(\text{button label} = \text{editTask label Void}\)

\[ \text{ok :: Task Void} \]

\(\text{ok} = \text{button "Ok"}\)

\[ \text{cancel :: Task Void} \]

\(\text{cancel} = \text{button "Cancel" Void}\)

\[ \text{void :: Task Void} \]

\(\text{void} = \text{return}_V \text{Void}\)

\(\text{Void}\) is similar to Haskell’s () value, and is defined as :: \(\text{Void} = \text{Void}\).

The use of \text{Maybe} values, as done by \text{break}, is a common functional programming idiom. Because many tasks yield these values, it is useful to define an alternative \(\gg\gg\) combinator:

\[ \text{try :: (Task (Maybe a)) (a } \rightarrow \text{Task b) (Task b) } \rightarrow \text{Task b | iData b} \]

\(\text{try taska taskfa taskb} = \text{taska} =\gg\gg \lambda x \rightarrow\)

\(\text{case x of}\)

\(\text{Nothing } \rightarrow \text{taskb}\)

\(\text{Just} x' \rightarrow \text{taskfa} x'\)
(try t succeed fail) is a task that first performs t. If t succeeds, and yields (Just v), then the task proceeds as (succeed v). If t fails, and yields Nothing, then the task proceeds as fail. Another useful alternative >>= combinator is breakable:

breakable :: (Task a) (a → Task Void) → Task Void | iData a
breakable taska taskfa = try (break taska)

(taskfa void)

(breakable t succeed) is a task that first performs t, while at the same time allowing the worker to abort t. If the worker chooses to finish t and yield a value v, then the task proceeds as (succeed v). If the worker chooses to abort t at any stage, the whole task returns Void.

As an example of this combinator, we can turn the addUser task for the program chair (defined at the end of Sect. 2.5) into a task that can be aborted:

```
addUser :: Task Void
addUser = breakable addUserForm
  (λuser → addUserDB user >>=
   endMsg >>= ok)
```

where endMsg = [Txt "A new user has been added"]

### 2.7 Recursive tasks

So far we have introduced sequential, monadic, composition and choice. The next key ingredient is to allow recursive workflow specifications. Recursion is fundamental to define computations that may run arbitrarily long. First we start with a useful combinator that can be found in the iTask API, foreverTask:

```
main :: User → Task Void
main user =:
  {User | login = {loginName}, role}
  = welcomeMsg >>= foreverTask (chooseTask homeMsg userTasks)
```

where

- welcomeMsg = [H1 [] ("Welcome " ++ loginName), Br]
- homeMsg = [Txt "Choose one of the tasks below or select a task that has been ", Txt "assigned to you from the list on the left", Br, Br]
- userTasks = case role of
  Chair = [("Show users", showUsers),
    ("Add user", addUser),
    ("Show papers", showPapers),
    ("Assign reviewers", assignReviewers),
    ("Judge papers", judgePapers)]
  PC = [("Show papers", showPapers),
    ("Mark papers", markPapers user)]

( foreverTask t ) repeats task t infinitely many times in sequence. It is used in this code fragment of the CM system to define the main part of the possible actions.
of a user, once he has successfully logged in. Because we do not know how long the user will keep logged in, she is offered a choice between several tasks infinitely many times. The userTasks function defines the possible tasks, depending on the role of the particular user.

2.8 Multi-User Workflows

So far the examples that have been shown are single user applications. Workflow systems usually involve arbitrarily many users. This is supported by the iTask system. The simplest way is to use the multiUserTask function, which has exactly the same type as the function singleUserTask that we have used so far. You can try this on any of your previous exercises and study the difference. However, most applications require some login ritual to allow only known users access to the application. Of course, the CM system is an example of such an application. For this purpose, a more elaborate function has been provided:

\[
\text{workFlowTask :: } \forall \text{[StartUpOptions]} \rightarrow \forall \text{[(Task ((Bool,UserId),a))]} \rightarrow \forall \text{[(UserId a → LabeledTask b)]} \rightarrow \forall \text{[!(HSt → ([!Bool,Html,HSt] | iData b)]).}
\]

The second argument of workFlowTask is to determine whether the person who is attempting to log in is a known user, and return a True boolean value if so (as well as the user’s UserId which is an integer value, and the initial data that that user requires). The third argument is the actual task that the user can continue to work on once successfully logged in. In the CM system case study, you can find this function right at the top at the Start function. The action that determines whether the user is known is called public, and the action that the user can continue with is called main, which we have already encountered in Sect. 2.7.

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 \( i@:(m,t) \). If there is no motivation, then one uses \( i@::t \).

\[
(\text{@::}) \quad \text{infix 3 :: !UserId !([LabeledTask a] → Task a | iData a)}
\]

\[
(\text{@::}) \quad \text{infix 3 :: !UserId !([Task a] → Task a | iData a)}
\]

Exercises

9. orTasks versus andTasks

Create a workflow 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 button tasks. Study the different behavior of orTasks and andTasks.

10. **Number guessing**
Create a 2-person workflow 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 workflow should give feedback to person 2 whether the number guessed is too low, too high, or just right. In the latter case, the workflow returns Just $N$. Person 2 can also give up, in which case the workflow should return Nothing.
**Optional:** Person 1 is given the result of person 2, and has a chance to respond with a ‘personal’ message.

11. **Tic-tac-toe**
Create a 2-person workflow 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 workflow has to ensure that players enter marks only when it is their turn to do so.

2.9 **Speculative tasks and multiple users: deadlines**
Workflow 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:

```haskell
waitForDateTask :: HtmlDate \rightarrow Task HtmlDate
waitForTimeTask :: HtmlTime \rightarrow Task HtmlTime
waitForTimerTask :: HtmlTime \rightarrow 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; `waitForTimer`-Task terminates after a given time interval.

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

```haskell
delegateTask who time t
= ("Timed Task",who)@:
  0:( (waitForTimeTask time >>= return_V Nothing)
    -||-
      (Txt ("Please finish task within" <+ time))
    >>= (t >>= \v \rightarrow return_V (Just v)))
```

`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 workflow situation is modeled as follows:
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 workflow described by \( \text{deadline} \ t \) defines a single delegation. It can be transformed into an iteration with the \texttt{foreverTask} combinator that we have also used in Sect. 2.7. We are obviously creating a multi-user system, and hence use the \texttt{multiUserTask} wrapper function for some constant \( n > 0 \). As example task we reuse the \texttt{simple} task from Sect. 2.1 with a concrete, non-overloaded type. This finalizes the example:

\[
\texttt{Start world = doHtmlServer (multiUserTask n True (foreverTask (deadline simple) @ Database)) world}
\]
Exercises

12. *Delayed task*
Create a workflow 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.

13. *Number guessing with deadline*
Use the delegation example of Sect. 2.9 in such a way that the number guessing game of exercise 10 can be created with it.

14. *Tic-tac-toe with deadline*
Use the delegation example of Sect. 2.9 in such a way that the tic-tac-toe game of exercise 11 can be created with it.

2.10 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:

\[
\text{Review} = \text{Approved} \mid \text{Rejected} \mid \text{NeedsRework TextArea}
\]

`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:

```hs
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, (\texttt{review } v) \text{ displays } v \text{ in the browser. The reviewer subsequently has to choose whether } v \text{ should be reworked, and can comment on her decision, or } v \text{ can be approved or rejected.}

The main task is to produce a product } v \text{ according to some task } t \text{ that can be judged by a reviewer } u. \text{ If the reviewer demands rework of } v, \text{ the task should be restarted with that particular } v, \text{ 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 \rightarrow \text{Task } a), \text{ and the result of the main task is to return a product and its review } (a, \text{Review}).\text{ This is done as follows:}

\begin{verbatim}
  taskToReview :: UserID (a,a \rightarrow Task a) \rightarrow Task (a,Review) \mid iData a
  taskToReview reviewer (v,task)
  = newTask "taskToReview"
    ( task v
      \Rightarrow \lambda nv \rightarrow
        reviewer @:: review nv \Rightarrow \lambda r \rightarrow
          \begin{cases}
            \text{NeedsRework }_1 & \rightarrow \text{taskToReview reviewer } (nv,task) \\
            \text{else} & \rightarrow \text{return } V (nv,r)
          \end{cases}
    )
\end{verbatim}

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 (\texttt{multiUserTask \_2}) in which user 0 creates the product, and user 1 reviews it:

\begin{verbatim}
Start world
= doHtmlServer (multiUserTask \_2 True (foreverTask reviewtask \llf TxtFile)) world

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

  t :: a \rightarrow Task a \mid iData a
  t v = [Txt "Fill in Form:"] \gg editTask "TaskDone" v \llf Submit
\end{verbatim}

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

2.11 Spawning tasks and controlling them

A novel feature of the iTask toolkit is the ability to \texttt{spawn} and \texttt{delete} arbitrarily complex new tasks. Existing tasks can use a number of functions to check or wait for completion of such a spawned task. Tasks can get suspended and activated again, and tasks can suspend or delete themselves. These functions can be found in the module \texttt{iTasksProcessHandling.dcl}. We show the main definitions here:
A spawned task \( t \) of type \((\text{Task } a)\) is identified and manipulated by means of an identification value of type \((\text{Wid } a)\). Now \((\text{spawnWorkflow } \text{uid } \text{active } \text{label}, t)\) creates a new task \( t \) that runs in parallel to the currently existing tasks. This new task \( t \) is handled by user \( \text{uid} \), and if \( \text{active} \) holds, it will be an active task the user can engage in immediately. If \( \text{active} \) does not hold, then the task is initially suspended. \text{spawnWorkflow} returns the handle \( \text{ht} \) to the spawned task.

It should be noted that the behavior described above is very similar to the use of \( \& \) and \( \&: \) combinators that have described in Sect. 2.8. However, because we now have a handle to such a spawned task, we can create more complicated, and more realistic, workflow cases. Consider for instance the need of the program chair in the case study to assign reviewing tasks to program members. Only after every review task has been finished, the program chair can proceed to collect the information and make a decision on the papers. This can be expressed as a single \text{andTasks} expression, sequentially followed by the task to make the decision. Unfortunately, real life is usually less structured: for a subset of papers it becomes rapidly clear that they should be accepted, and another subset gets rejected; some papers require additional reviewing; and some reviewers may fail to deliver before the deadline. Hence, it makes more sense to structure this workflow as a set of spawned tasks. In the case study, this is done in the function \text{assignReviewersForm}.

The functions mentioned above are fairly self-explanatory. One interesting function is \text{changeWorkflowUser}, which, when given a user identification, transfers the indicated task to the given user. This is of course a useful construct that occurs many times in the real world: workers may get ill, change jobs, have holidays, but the work remains to be done. For these cases new resources need to found and work has to be reallocated.
15. **Number guessing in a group**
In this exercise you extend the number guessing game of exercises 10 and 13 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.

### 2.12 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 workflow 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. 2.5. The full iTask toolkit that has been shown in Sect. 2 is built according to these principles.
3.1 iData as Primitive iTasks 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
```

`mkIData` contains the internal administration that is required for creating HTML pages and handling forms. Please consult [17] 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:

```
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` 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
```

When evaluated, `(mkIData (init, InIDataId))` basically performs the following actions: it first checks whether an earlier incarnation of the iData element (identified by `InIDataId.id`, i.e. the name of the iData element) exists. If this is not the case, or `init` equals `Set`, then `InIDataId.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] }
```

\(^4\) 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.
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 Boolean `iData` state in which we record whether the editor task is finished. It is not up to an `iData` editor to decide whether a task is finished, but this is indicated by the user by pressing an additional button. Hence, a standard `iData` editor is extended with a button and a boolean storage. These elements are created by the functions `simpleButton` and `mkStoreForm`:

```hs
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:

```hs
taskEditor :: String String a *HSt → (tBool, a, [BodyTag], *HSt) | iData a
```

1. 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):

\[
\begin{align*}
*: & *TSt = \{ \text{hst} :: *HSt, \text{activated} :: \text{Bool}, \text{tasknr} :: \text{TaskID}, \text{html} :: [\text{BodyTag}] \} \\
*: & *TSt = [\text{Int}] \\
\text{set_activated} :: & \text{Bool} *TSt \rightarrow *TSt \\
\text{set_tasknr} :: & \text{TaskID} *TSt \rightarrow *TSt \\
\text{set_html} :: & [\text{BodyTag}] *TSt \rightarrow *TSt \\
\end{align*}
\]

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 \rightarrow (a, *TSt):

\[
\begin{align*}
\text{editTask} :: & \text{String} a \rightarrow \text{Task} a \mid \text{iData} a \\
\text{editTask} \ \text{label} \ a & = \text{doTask} \ \text{editTask}' \\
\text{where} \\
\text{editTask}' \ \text{tst} & =: \{ \text{tasknr, hst, html} \} \\
\text{♯} \ (\text{done, na, nhtml, hst}) & = \text{taskEditor} \ \text{label} \ \text{toString} \ \text{tasknr} \ a \ \text{hst} \\
& = (\text{na}, \{\text{tst & activated} = \text{done, hst} = \text{hst, html} = \text{html} + + \text{nhtml}\}) \\
\text{editTask} \ & \text{takes an initial value of any type and delivers an iTask of that type. When the task is activated, an extended iData element is created by calling taskEditor. A unique identifier is generated by this system (function doTask, which is explained below), which eliminates the shortcoming that was mentioned above. Any iData element automatically remembers the state of any edit action, no matter how complicated the editor is. The HTML code produced by taskEditor is added to the accumulator of the iTask state. In the end all HTML code of all iTasks can be displayed by showing the HTML of the top-task. There can be many active iTasks, so in practice this is probably not what we want. However, for the core system this will do.} \\
\text{The function doTask is an internal wrapper function that is used within the iTasks toolkit for every iTask.} \\
\text{doTask :: (Task a) } \rightarrow \text{Task a } \mid \text{iCreate a} \\
\text{doTask mytask} & = \text{doTask'} \circ \text{incTaskNr} \\
\text{where doTask'} \ \text{tst} & =: \{\text{activated, tasknr}\} \\
\text{♯} \ (\text{not activated} = (\text{createDefault}, \text{tst}) \\
& = (\text{val, tst}) = \text{mytask} \ \text{tst}
\end{align*}
\]
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 \texttt{incTaskNr} below, whereas a subtask \( j \) of task \( i \) is numbered \( i.j \) with \texttt{subTaskNr} below. Fresh subtask numbers are generated with \texttt{newSubTaskNr}. We represent the numbering with an integer list, in reversed order.

\begin{verbatim}
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]\}\n\end{verbatim}

The systematic numbering is important because it is also used for garbage collection of subtasks (see Sect. 3.6).

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

\begin{itemize}
  \item If not activated, the \texttt{createDefault} value is returned. This explains the overloading context restriction of \texttt{doTask}. As a consequence, an \texttt{iTask} always has a value, just as an \texttt{iData} element.
  \item 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 \texttt{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 \texttt{doTask} and the task number is reset to its original value.
\end{itemize}

It is assumed that any task sets \texttt{activated} to \texttt{True} if the task is finished (indicating that the next task can be activated), and to \texttt{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. 2.5, sequential composition within the \texttt{iTask} toolkit is based on monads. Thanks to uniqueness typing we can freely choose how to thread the unique \texttt{iTask} state \(*TSt\) either in explicit environment passing style or in implicit monadic style. In the implementation of the \texttt{iTask} system we have chosen for the explicit style: it gives more flexibility because we have direct access to both the unique \texttt{iTask} state \(*TSt\) and the unique \texttt{iData} state \(*HSt\) as is shown in the definition of \texttt{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 \texttt{return\_D} function is straightforward:

\begin{verbatim}
return_D :: a → Task a | gForm(| |)
return_D a = doTask (λtst → (a, {tst & html = tst.html ++ toHtml a})
\end{verbatim}

The implementation of the prompting combinators \texttt{??>} and \texttt{!!>}> is also not very difficult:

\begin{verbatim}
(???) \textbf{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})

(!!>) \textbf{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})
\end{verbatim}

3.3 Reflection (Part I)

The behavior of the described core system is a combination of re-evaluating the application and having the enhanced \texttt{iData} elements retrieve their previous states that are possibly updated with the latest changes done by the application user. The \texttt{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 \texttt{iTask} editor created with \texttt{editTask} automatically remembers its contents and state (finished or not) while the other \texttt{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 \texttt{HTML} code. In this way we achieve the same result as in \texttt{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 \texttt{???>} and \texttt{!!>}>. 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.

\[\text{(-& & -) \text{ infixr 4 :: (Task a) (Task b)} \rightarrow \text{Task (a,b) | iCreate a & iCreate b}}\]

\[\text{(-||-)} \text{ taska taskb = doTask and}
\]

\[\text{where and tst::{tasknr}}\]

\[\begin{cases}
\emptyset \text{ (a.tst::{activated=adone})} = \text{mkParSubTask 0 tasknr taska tst} \\
\emptyset \text{ (b.tst::{activated=bdone})} = \text{mkParSubTask 1 tasknr taskb tst} \\
= ((a,b), \text{setactivated (adone & & bdone)} \text{ tst})
\end{cases}\]

\[\text{(-||-) \text{ infixr 3 :: (Task a) (Task a)} \rightarrow \text{Task a | iCreate a}}\]

\[\text{where or tst::{tasknr}}\]

\[\begin{cases}
\emptyset \text{ (a.tst::{activated=adone})} = \text{mkParSubTask 0 tasknr taska tst} \\
\emptyset \text{ (b.tst::{activated=bdone})} = \text{mkParSubTask 1 tasknr taskb tst} \\
= (\text{if adone a (if bdone b createDefault)}, \text{setactivated (adone || bdone)} \text{ tst})
\end{cases}\]

\[\text{mkParSubTask :: Int TaskID (Task a)} \rightarrow \text{Task a}}\]

\[\text{mkParSubTask i tasknr task = task o newSubTaskNr o setactivated True o subTaskNr i}}\]

The function \text{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 \text{foreverTask} which repeats a task infinitely many times.

\[\text{foreverTask :: (Task a) \rightarrow Task a | iCreate a}}\]

\[\text{foreverTask task = doTask (foreverTask task o snd o task o newSubTaskNr)}\]

As an example, consider the following definition:

\[t = \text{foreverTask (sequenceITask -||- editTask "Cancel" createDefault)}\]

In \(t\) the user can work on \text{sequenceITask} (Sect. 2.5), 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 workflows. As we have stated in Sect. 2.7, a crucial combinator for recursion is \text{newTask}.

\[\text{newTask :: (Task a) \rightarrow Task a | iCreate a}}\]

\[\text{newTask task = doTask (newTask o newSubTaskNr)}\]

\((\text{newTask } t)\) 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 workflow:
getPositive :: Int → Task Int
getPositive i = newTask (getPositive' i)
where
    getPositive' i = [Txt "Type in a positive number:"]
        ?> editTask "Done" i ⟷ λni →
        if (ni > 0) (return ni) (getPositive ni)

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 workflow 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 workflow 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 workflow 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. 2.8, 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
 | (+-)     infix1 1 HtmlTree HtmlTree
 | (+|+)     infix1 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 (t1+-t2) (and {t1++t2}) place output t1 left (above) of output t2.

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 (`[]`). 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 2.9). 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 workflow, 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.

```haskell
newTask :: (Task a) \rightarrow Task a | iData a
newTask t = doTask (\tst\{\tasknr\,\hst\})
  \$ (taskval\,\hst) = mkStoreForm (Init\,storeId) id hst
  \$ (done\,v) = taskval.value
  | done = (v,\{tst \& hst = hst\})
```

1. `newTask` takes a `Task a` and returns a `Task a` or an `iData a`.
2. Inside the `doTask` function, the tasks are evaluated.
3. The result is stored in a `StoreForm`.
4. The `done` action evaluates to the new value.
5. Otherwise, it returns the updated `hst`.

36
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. ??). 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 workflow patterns that can be found in contemporary workflow tools (see [21]). We show that the iTask toolkit does support this workflow pattern, and that it does so in a concise way. The complete realization of the \((p \mapsto t)\) is as follows:

\[
\begin{align*}
\text{(\(\mapsto\))} \quad \text{infix} 4 \, :: \, \{ \text{Task } s \} \, (\text{Task } a) & \mapsto \text{Task } (\text{Maybe } s, \text{TClosure } a) \quad 1. \\
\quad \mid \text{iCreateAndPrint } s & \& \text{iCreateAndPrint } a \\
\text{(\(\mapsto\))} \, p \, t & = \text{doTask } (\lambda tst::\{\text{tasknr, html}\}) \quad 2. \\
\quad \text{\(\textcircled{\#} \) } (v,\text{tst}::\{\text{activated } = \text{done, html } = \text{task}\}) & = t \, \{ \text{set } (\text{BT } []) \, \text{True } tst \, \& \, \text{tasknr } = \text{taskId} \} \quad 3. \\
\quad \text{\(\textcircled{\#} \) } (s,\text{tst}::\{\text{activated } = \text{halt, html } = \text{stop}\}) & = p \, \{ \text{set } (\text{BT } []) \, \text{True } tst \, \& \, \text{tasknr } = \text{stopId} \} \quad 4. \\
\quad \mid \text{halt} & = \text{return } (\text{Just } s, \text{TClosure } (\text{close } t)) \quad 5. \\
\quad & \quad (\text{set html } \text{True } tst) \quad 6.
\end{align*}
\]
| done | = return (Nothing, TClosure (return v))
| --- | ---
|    | (set (html +|+ task) True tst)
| otherwise | = return (Nothing, TClosure (return v))
|        | (set (html +|+ task +|+ stop) False tst)
|    | )

where close t = t o (set_tasknr taskId)
set html done = (set_html html) o (set_activated done)
stopId = [-1:0:tasknr]
taskId = [-1:1:tasknr]

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 HaskellCGI library [16]; the Curry approach [12]; writing XML applications [9] in SMLserver [8]. One sophisticated system is WASH/CGI by [20], 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 [19]. Both languages aim to deal with web programming within a single framework, just as the iData and iTasks 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” [18]. Graumke 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 [?], GdH [?], 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 workflow patterns of over more than 30 contemporary commercial workflow systems [21]. 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. ?? 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 workflow 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 workflow tools use a graphical formalism to specify workflow 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 workflow 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 workflows 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 workflows, embedded in Clean. The specification is used to generate a multi-user interactive web-based workflow 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 workflow systems, who design workflows, 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 workflow 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 workflow system on a sophisticated high level of abstraction.

Future work will be the investigation of more “unusual” useful workflow 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 workflow systems and as a source of many examples, and Wil van der Aalst for commenting on the difference between the combinator approach and contemporary workflow specification languages.

References


A iTask toolkit

This is the complete api of the iTask toolkit.

```haskell
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) !*HTst → [a, [BodyTag], !*HTst] | iCreate a
```

43
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) *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 */
(>>>) infixl 1 :: (Task a) a -> Task b -> Task b | iCreateAndPrint b
(>> ) infixl 1 :: (Task a) (Task b) -> Task b | iCreate 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 \{} \star\}, iCreateAndPrint a
As sign tasks to user with indicated id
(â:) :: will prompt who is waiting for task with give name
(â::) :: same, default task name given

/* 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 | iCreateAndPrint 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
(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
(orTask2 :: (Task a, Task a) → Task a | iCreateAndPrint a
& iCreateAndPrint a
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
(andTask :: both iTasks in any order, completion when both done
*/
andTask :: (Task a, Task a) → Task a | iCreateAndPrint a
(-&-) :: 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 (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

46
*/
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 iTdata 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 | iTdata a
appHSt :: (HSt → (a,HSt)) → Task a | iTdata a
appHSt2 :: (HSt → (a,HSt)) → Task a | iTdata a

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