iTask as a new paradigm for building GUI applications

Steffen Michels, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
s.michels@science.ru.nl, {rinus, p.achten}@cs.ru.nl

Abstract. The *iTask* system is a combinator library written in Clean offering a declarative, domain-specific language for defining workflows. From a declarative specification, a complete multi-user, web-enabled, workflow management system (WFMS) is generated. In the *iTask* paradigm, a workflow is a definition in which interactive elements are defined by editors on model values (abstracting from concrete GUI implementation details). The order of their appearance is calculated dynamically using combinator functions (abstracting from concrete synchronisation details). Defining interactive elements and the order of their appeareance are also major concerns when programming GUI applications. For this reason, the *iTask* paradigm is potentially suited to program GUI applications as well. However, the *iTask* system was designed for a different application domain and lacks a number of key features to make it suited for programming GUI applications. In this paper, we identify these key features and show how they can be added to the *iTask* system in an orthogonal way, thus creating a new paradigm for programming GUI applications.

1 Introduction

Workflow management systems (WFMS) are software systems that coordinate, generate, and monitor tasks performed by human workers and computers. The *iTask* system [10] is a combinator library written in *Clean*, which offers a high-level, declarative, domain specific language for defining web-based workflows. Tasks are defined by typed, pure functions, and are dynamically calculated: the actual work to do can depend on the outcome of previous tasks. One can sequence tasks, create parallel tasks in all kinds of flavors, tasks can be defined recursively and can be higher order. User interaction points are editors on model values.

Due to the use of generic programming techniques, an *iTask* programmer does not need to worry about boilerplate programming, such as the handling of the communication between client and server, and the form rendering and handling of form updates on the browser. Thus, the *iTask* paradigm allows the programmer of a workflow to fully concentrate on its logic: the description of tasks and the dependencies between them.

Programming GUI applications shares many concerns with programming workflows: user interaction points need to be defined and their order of appearance needs to be controlled as well. However, existing approaches for programming GUI applications are very different from the *iTask* paradigm (see Section 5). Earlier experience in using the *iTask* system to create an interactive application to explore extended state machine specifications [7] suggested that the *iTask* paradigm is suited to define GUI applications. However, essential features for defining GUIs are lacking in the current *iTask* system. Those shortcomings emerge from the different nature of normal workflows and GUI applications. Typically, when the user fills in some form in the *iTask* system, she works on local data in a browser, and by pushing a button a value of required type is returned to the server. An average GUI application offers many more options for the user to choose from, using GUI elements like buttons, menus, and dialogs. It offers multiple interactive windows one can work on simultaneously. Information being modified is no longer local, because doing something in one window might affect the contents of another.

In this paper we identify the missing key features for GUI programming in the *iTask* system and show how they can be added orthogonally, thereby making the *iTask* paradigm fit for defining GUI applications. To obtain a GUI specification which is as declarative as possible, we realised that we had to restrict ourselves as well. We do not provide the programmer with fine-grained control over the layout of forms, dialogs, menus, and windows. Such information has to be defined separately, e.g. in style sheets. We also do not offer primitives for drawing on canvas but stick to handling forms with generically generated layouts and the like. The contributions of this paper are:

- We show that the *iTask* system can be extended to support programming GUI applications. The result is a new paradigm for programming GUI applications in a declarative way, i.e., **only** data and processes (tasks) need to be defined.
- The extensions are fundamental GUI elements: windows that can be dynamically opened and closed, and user actions that are (dynamically) organized in buttons and menus.
- In the new system stand-alone web-based GUI applications can be created. The extensions are orthogonal to the *iTask* system, i.e. they do not alter existing workflow applications.
- *iTask* users can now work simultaneously on GUI tasks which share information. A change made by one worker is made visible to others, enabling new kinds of tasks such as chat or dashboard applications.

The remainder of this paper is organized as follows. First, in Section 2, the *iTask* system is introduced using a running example of a text editor. This is a somewhat unconventional example of a workflow. However, the goal is to illustrate both that the *iTask* system is potentially sufficiently powerful to express such kinds of GUI applications as well as pin-pointing its current shortcomings for that domain. We introduce the required extensions to the iTask system step-by-step in Section 3 and apply them in the running example. In Section 4, we demonstrate

that the extensions are orthogonal to *iTask* and create a multi-user workflow application that uses the GUI features. Related work is discussed in Section 5. We conclude and discuss future work in Section 6.

2 Introducing the iTask System

The *iTask* system is a monadic combinator library for specifying workflows. In Section 2.1 we give a concise overview of the library. We present in Section 2.2 the running example of a simple text editor. Further, we discuss what kind of functionality is missing in *iTask* for programming GUI applications.

2.1 Library Overview

The *iTask* library consists of *basic tasks*, representing the atomic actions the user can take, which can be composed to build complex workflows using *combinators*. Basic tasks that are used in the running example are:

```
updateInformation :: String String  a  → Task a | iTask a
enterInformation  :: String String     → Task a | iTask a
enterChoice       :: String String [a] → Task a | iTask a
```

Note that in *Clean* the arity of functions is shown explicitly by separating argument types by spaces instead of →. Every task in the system returns a value of abstract type `Task a` where `a` is some concrete type. The basic task `updateInformation` generates a form for a value of type `a`. The form is an editor: the user can inspect and alter the value arbitrarily many times. The programmer provides a short title and description to inform the user about the purpose of this editor. When the form is completed in the editor, the information committed by the user is turned into a value of type `a`. The *iTask* system automatically performs the necessary conversions, using generic functions for serialisation, generating user interfaces, updating and verifying values included in the type class `iTask`. The function `enterInformation` is the same except that the system generates a form with blank fields for entering a value of type `a`. The function `enterChoice` lets the user choose a value from a list of options.

Other basic tasks that we need are concerned with reading and writing values of type `a` in a database, using typed references of type `DBId a`:

```
createDB :: a                → Task (DBId a)  | iTask a
readDB   :: (DBId a)         → Task a         | iTask a
writeDB  :: (DBId a) a       → Task a         | iTask a
```

iTask uses the monadic combinators `return`, `>>=`, and `>>|` to combine tasks:

```
return         :: a                          → Task a | iTask a
(>>=) infixl 1 :: (Task a) (a → Task b)      → Task b | iTask a & iTask b
(>>|) infixl 1 :: (Task a) (Task b)          → Task b | iTask a & iTask b
```

Finally, we need a combinator to compose tasks in parallel: `-&&-` performs both tasks and returns their combined result when both are terminated. The operator `@:` assigns a task to the indicated user:

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
(@:)   infix  3 :: User     (Task a) → Task a     | iTask a
```

The *iTask* paradigm lets the programmer concentrate on defining the workflow processes and the data involved. The generic machinery takes care of building and handling forms, storing intermediate results, and keeping track of the application state.

2.2 Running Example: a Simple Text Editor

In order to investigate the suitability of the *iTask* paradigm for programming GUI applications, we perform a case study of a typical GUI application, viz. a single-user text editor. Although *iTask* was not designed for such a purpose, it is possible to create such an application. The case study pin-points the shortcomings of the *iTask* system for programming GUI applications.

First, we define the "global" state of the editor. Since we are dealing with a pure functional language, this state is explicitly passed from one task to another. The state consists of the current text and in which file it is stored, if at all:

```
:: State    =  State Note FileInfo
:: Note     =  Note String
:: FileInfo =  NotStored | StoredFile FileName
:: FileName :== String
derive class iTask State, FileInfo

initState   = State (Note "") NotStored
```

The current content is of type `Note` which contains a string. Type `Note` is predefined and represented in a browser by a multi-line text-area. To keep the example simple, only one file can be opened at a time. The current content is either stored in a file with given name or it is not yet stored. For any user defined type used in the iTask system, such as `State` and `FileInfo`, an instantiation of the generic iTask type class is needed. The compiler can derive them automatically. Initially the application starts with empty content that is not stored yet.

There are a number of different operations the user can perform on the state. Examples are: editing the content, replacing substrings, or saving the content. To let the user choose which action to perform, the task `enterChoice` (see Section 2.1) is used. The different actions are straightforwardly represented by string constants:

```
Edit :== "Edit Content"
New  :== "New"
...
allActions = [ Edit, New, Open, Save, SaveAs, ... ]
```

To actually perform those actions, `enterChoice` has to be put in sequence with a task performing the selected operation:

```
1 textEditorApplication = performAction initState
2
```

```
3  performAction :: State → Task Void
4  performAction state =
5    enterChoice "Choose Action" "Which action to perform?" allActions >>= λaction →
6    case action of
7      New     →                   performAction initState
8      Edit    → edit    state >>= performAction
9      Open    → openFile      >>= performAction
10     Save    → save    state >>= performAction
11     SaveAs  → saveAs  state >>= performAction
12     Replace → replace state >>= performAction
13     Quit    →                   return Void
```

First, the user chooses an action (line 5). The resulting GUI is given in Figure 1a. When finished, the result `action` is inspected to dynamically determine how to proceed. If `Quit` is chosen, `performAction` terminates and returns `Void` (line 13). If the user requests editing a new file (`New`), `performAction` is called recursively with the initial state (line 7). Tasks for other purposes return an updated state which is used for the next recursion (lines 8–12). Hereafter the user can choose another action. The file operations are implemented using Clean's `StdFile` operations. Due to space limitations, we do not discuss their implementation, but restrict ourselves in the remainder of this paper to the `edit` and `replace` tasks.

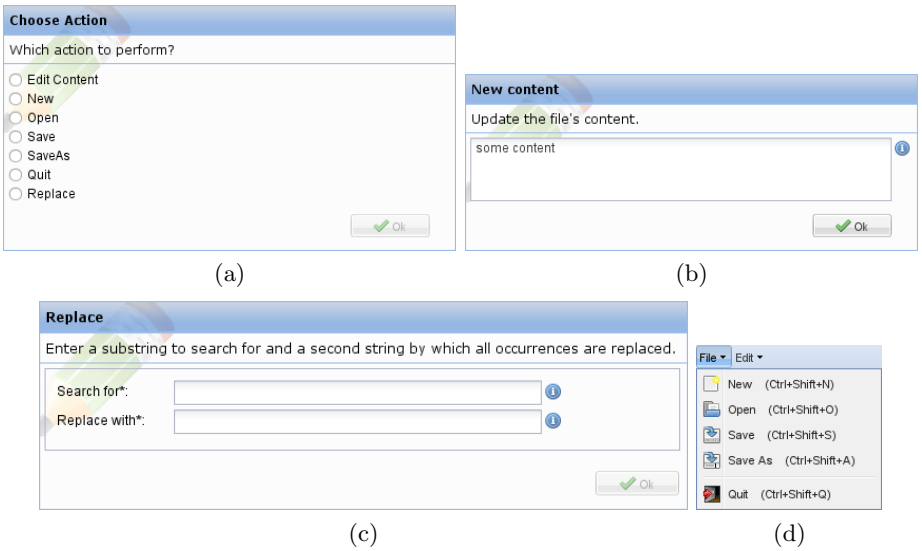

Fig. 1: Screenshots of the text editor example and the file menu

The task `edit` lets the user modify the current content stored inside the state:

```
edit :: State → Task State
edit (State content file) =
```

```
  updateInformation "New content" "Update the ..." content >>= λnewContent →
  return (State newContent file)
```

The current content is updated by the user, using the `updateInformation` task. The new content is returned in the new state. Figure 1b shows the corresponding GUI. In order to replace substrings, we need to know the search string and the replacement string. We derive a GUI for the appropriate type (see Figure 1c for a screenshot):

```
:: Replace = { searchFor :: String, replaceWith :: String }
derive class iTask Replace
```

Given a desired replacement, we can replace substrings in a `String`, `Note`, and `State` in a straightforward way:

```
class    replSubStr a :: Replace a → a
instance replSubStr String, Note, State
```

The task to replace substrings first asks the desired parameters, and applies the replacement function on the current note content of the state:

```
replace :: State → Task State
replace state =
  enterInformation "Replace" "Enter a substring to search for and..." >>= λr →
  return (replSubStr r state)
```

One might want to define conditions under which a certain action is possible. For instance, choosing `Replace` only makes sense for non-empty content. Because the task is dynamically generated at each recursion, this is easily achieved by making the list of actions depend on the current state:

```
allActions :: State → [String]
allActions (State (Note cont) _) =
     [Edit, New, Open, Save, SaveAs, Quit]
  ++ if (cont ≠ "") [Replace] []
```

The case study demonstrates that the *iTask* language is, in principle, powerful enough to describe all the functionality of this typical GUI application. However, the resulting application is an entirely untypical GUI application. Turn and turn about, the user must select an operation and must complete it fully to make progress. The user interface (selecting operations, editing text, replacing substrings) is scattered instead of integrated and being organized using menus and windows. We want to add menus and windows to *iTask* without sacrificing the conciseness and declarative nature of the above specification.

3 Extending the iTask System

We concluded in the previous section that in *iTask* the user interface is scattered along the application instead of being integrated and organized using common GUI elements such as menus and windows. In this section we show how to glue the scattered pieces in order to obtain a system that is fit for GUI programming.

These extensions are illustrated with the running text editor example to show their effect. The extensions are a significant design and implementation effort. For this reason, we break down their discussion into smaller units. We start with adding the concept of actions, from which the user can choose, to the *iTask* editor concept (Section 3.1). The next step is to associate and organize editor actions with menus (Section 3.2). Before we can introduce multiple windows, we need to enable editors to listen to each other (Section 3.3). Once this has been done, we can introduce windows that can be added and closed dynamically (Section 3.4). The final step is to make menus also dynamic (Section 3.5).

3.1 Giving the User More Choices

The control flow of the text editor in Section 2.2 allowed the user to edit the text only after invoking a command to do so. It is more natural to regard the task for editing the content as a central task that offers a number of optional actions to complete this task. So, instead of offering only the standard 'commit' button to an editor, we extend the editing combinators with a list of optional actions to terminate the editor and commit its value. Here we give the variant of `updateInformation`:

```
updateInformationActions :: String String [TaskAction a] a → Task (Action, a)
                                                            | iTask a
```

A list of *task actions* of type `TaskAction a` is included. Besides the updated value of type `a`, the selected *action* of type `Action` is returned. (Note that if the list is empty, the user has no way to terminate the editor and can only edit the value. This proves to be useful after we have taught editors to listen to each other in Section 3.3). Before we explain the roles of these new types, we show the improved text editor:

```
1  textEditorApplication = performAction initState
2
3  performAction (State content file) =
4    edit content         >>= λ(action, nContent) →
5    let state = State nContent file
6     in case action of
7      ActionNew          →                   performAction initState
8      ActionOpen         → openFile      >>= performAction
9      ActionSave         → save    state >>= performAction
10     ActionSaveAs       → saveAs  state >>= performAction
11     Action "replace" _ → replace state >>= performAction
12     ActionQuit         →                   return Void
13 where edit             = updateInformationActions "Text Editor" "..." allActions
```

The structure is very similar to the structure of the text editor defined in Section 2.2. The available action options are now a parameter of the editing task (line 4 and 14). This emphasizes the central role of the editor. The result of the editor task is a tuple of the chosen termination action and updated content (line 4).

Each time an action is performed the task is called recursively which generates an entire new user interface. However, the user interface is rendered as a form inside a browser and can be updated without producing a flickering window. Starting the same task recursively many times can be avoided by using grouped tasks running in parallel (see Section 3.4).

The type `TaskAction a` is defined in the library:

```
:: TaskAction a :== (Action, Selectable a)
:: Action       =   Action ActionID ActionLabel | ActionOk | ActionCancel | ...
:: ActionID     :== String
:: ActionLabel  :== String
:: Selectable a :== (Verified a) → Bool
:: Verified   a =   Invalid | Valid a
```

Instead of using only text labels, a list of task actions given to the editor task `updateInformationActions` adds an identification (`ActionID`) and a predicate (`Selectable a`) to each label. For convenience, we include a number of frequently used actions (`ActionOk`, `ActionCancel`, ...). Their appearance is dictated by the client platform. For custom defined actions, the text label appears in the user interface, and the action ID is used to identify the selected action. The selected `Action` is returned. The `Selectable a` predicate is a condition that determines whether an action is enabled or disabled. The condition is checked each time the value being edited is updated. The library contains a number of predefined conditions, such as `always`, `ifvalid` and `ifinvalid`. An editor can be in an invalid state if no value has been provided. The condition commonly depends on the state of the editor, hence it is parameterized with the value if valid. For instance, the replace-action is applicable only if the text is not empty. The editor actions can now be defined as:

```
allActions :: [TaskAction Note]
allActions = [ (ActionNew,     always), (ActionOpen, always), (ActionSave, always)
             , (ActionSaveAs, always), (ActionQuit, always)
             , (Action "replace" "Replace", notEmpty) ]
notEmpty (Valid (Note txt)) = txt ≠ ""
notEmpty _                  = False
```

The result of this step is that optional actions are integrated with the editor concept without reducing the declarative nature compared to the original specification.

3.2 Structuring Choices Using Menus

Instead of presenting a long list of buttons representing the optional actions, a more user-friendly and common solution is to use menus to organize the commands. Conceptually, they act just like buttons. We separate the definition of optional actions in a particular context from the way they are presented to the user: either in the shape of a button or as a menu item. The programmer can define the desired menu layout in a task annotation. By default, a button is generated for actions in the task not mentioned in the menu declaration. Actions

mentioned in the menu annotation which are not defined in the task, are ignored. Before explaining the extension, we show how to extend the running example with menus (Figure 1d illustrates the corresponding menus):

```
textEditorApplication = performAction initState <<@ StaticMenus menus

menus :: Menus
menus = [ Menu "File"  [ MenuItem ActionNew    (hotkey N)
                       , MenuItem ActionOpen   (hotkey O)
                       , MenuItem ActionSave   (hotkey S)
                       , MenuItem ActionSaveAs (hotkey A)
                       , MenuSeparator
                       , MenuItem ActionQuit   (hotkey Q)
                       ]
        , Menu "Edit"  [ MenuItem "replace"    (hotkey R) ]
        ]
hotkey key = Just {ctrl = True, alt = False, shift = True, key = key}
```

The overloaded *iTask* tuning combinator `<<@` can be used to annotate arbitrary tasks, for example to change the initial worker or the priority. It is extended with an instance for menu annotations, defined as:

```
:: MenuAnnotation = NoMenus | StaticMenus Menus
```

The constructor `NoMenus` is added as a more readable way of using `StaticMenus []`. The menu structure is inherited by all children of that task, but can always be overridden with another annotation. For example, if a subtask should have no menus, it can be annotated with `NoMenus`.

The types available for defining menus are:

```
:: Menus     :== [Menu]
:: Menu      =   Menu MenuLabel [MenuItem]
:: MenuItem  =   ∃action: MenuItem action (Maybe Hotkey) & menuAction action
             |   SubMenu  MenuLabel [MenuItem]
             |   MenuSeparator
:: MenuLabel :== String
:: Hotkey    =   { key :: Key, ctrl :: Bool, alt :: Bool, shift :: Bool }
:: Key       =   A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | ...
```

The type variable `action` in the `MenuItem` data constructor is existentially quantified. The notation `& menuAction action` means that instances for a class `menuAction` are required and available within the data constructor for the existentially quantified `action` type. In this way, the programmer can choose to use the default `Action`s (`ActionOk`, `ActionCancel`, ...), `ActionID`s, or a combinations of `ActionID` and `ActionLabel`. For this purpose, instances have been defined for `menuAction` to map these items to their corresponding `Action`:

```
class menuAction :: a → (ActionID,ActionLabel)
instance menuAction Action, ActionID, (ActionID,ActionLabel)
```

In the first two cases the label shown in the menu is determined by the action given to the actual task the menu is built for. The last case is used to define

another label. In this way items generating actions with the same ID but different labels can be added to the menu. An example how this can be used is given in Section 3.5.

3.3 A View on Shared State

In this section we discuss how parallel tasks can mutually influence each other. This is required when constructing applications that handle a dynamic number of windows. The actual creation of such tasks is discussed in Section 3.4. We illustrate mutual influence by means of the running example: instead of switching between atomically editing the text and atomically replacing substrings in the same text, we want both actions available at the same time and in an interleaved way. While the `edit` task is running, any update of the text performed by invocations of the `replace` task (and also any other action) should update the text within the `edit` task as well. Hence, these tasks need to *share* the same state. Shared state is readily available in *iTask* by creating a database to the state, obtaining a typed reference `DBId State`, and updating this with the derived task `updateDB`[1]. The result is that all editor actions are parameterized with the shared state instead of the state. In this way, `replace` can simply update the shared state:

```
replace :: (DBId State) → Task Void
replace ref =
  enterInformationActions "Replace" "Enter..." actions >>= λ(action, r) →
  case action of
    Action "replaceAll" _ → updateDB ref (replSubStr r) >>| return Void
    ActionClose           → return Void
where
  actions = [ (ActionClose, always)
            , (Action "replaceAll" "Replace All", ifvalid) ]

updateDB :: (DBId a) (a → a) → Task a | iTask a
updateDB ref f = readDB ref >>= λa → writeDB ref (f a)
```

To keep the `edit` task informed of updates to the shared state, we add the well-known *model-view-controller* (MVC) concept [8] to *iTask*. In the case study, the model is the state, and `edit` is a view (it is only interested in the text of the state) on the state. The `edit` task registers itself as a view on this model in the following way:

```
edit :: (DBId State) → Task Void
edit ref = updateShared "Text Editor" "..." [ActionQuit] ref view >>| return Void
where
  view :: Bimap State (Display String,Note)
  view = ( λ(State content info)          → (Display (title info),content)
         , λ(_,newContent) (State _ info) → State newContent info
         )
```

[1] The *iTask* system ensures that all tasks a derived task is composed of is performed as one atomic operation.

```
  title NotStore          = "New Text Document"
  title (StoreFile name) = name
```

The difference with its previous version is that it is applied to a reference of the state, instead of the state value, and that it describes its view on the state with a separate bimap, defined by `view`. Here only the content part of the state is given as view to the user. Additionally a title is shown, wrapped by the special constructor `Display` to make it not editable.

Views are closely related to the theoretical concept of lenses [3]. The type class `SharedVariable` contains a new generic function that takes care of the automatic update of registered views whenever a shared value is modified. This generic function applies a merging algorithm in case of conflicting values. The signature of the function to register a viewer to a shared model value is an extension of the signature of `updateInformationActions` (Section 3.1):

```
updateShared :: String String [TaskAction a] (DBId a) (Bimap a v)
                              → Task (Event, a) | iTask a & iTask v & SharedVariable a
:: Bimap m v :== (m → v, v m → m )
```

We have shown how the tasks `replace` and `edit` can mutually influence each other. We have deliberately ignored the issue of creating these tasks in parallel. This is discussed in the next section.

3.4 Dynamic Task Groups

In general, a GUI application controls zero or more windows, which are dynamically created during the life-cycle of the application. Therefore, we need a means to identify a *group* of (windowed) tasks the number of which can vary during execution. It is convenient to attach a set of global actions to a group which can be chosen even if no window is opened, in addition to the local actions which can be attached to each task individually. Before we discuss the details, we first adapt the running example to use this feature to dynamically create tasks:

```
1  textEditorApplication =
2    createDB initState >>= λref →
3    dynamicGroup [edit ref] (allActions ref) (doAction ref) <<@ StaticMenus menus
4
5  allActions :: (DBId State) → [GroupAction Void]
6  allActions ref = [ (ActionNew,     Always), (ActionOpen, Always), (ActionSave, Always)
7                   , (ActionSaveAs, Always), (ActionQuit, Always)
8                   , (Action "replace" "Replace", SharedPredicate ref notEmpty) ]
9
10 doAction :: (DBId State) Action → DynAction
11 doAction ref action = case action of
12   ActionNew          → Extend [new      ref]
13   ActionOpen         → Extend [openFile ref]
14   ActionSave         → Extend [save     ref]
15   ActionSaveAs       → Extend [saveAs   ref]
16   ActionQuit         → Stop
17   Action "replace" _ → Extend [replace  ref]
```

The editor application starts initially with one task, the text editing task `edit` as presented in Section 3.3, that registers itself as a view on the current state (line 3). The list of editor actions, `allActions`, is now promoted to group actions because they are always available. Because the availability predicate for the `replace` task depends on a shared state, this function is parameterised with the proper reference. The group behaviour is defined with the function `doAction`. It is very similar to the case distinction in the earlier examples: the difference is that the list of chosen tasks given to `Extend` are dynamically added to the group. A separate window is created for every task thus created, such that one can work on all tasks simultaneously. The group, and all tasks in it, *terminate* when `Stop` is chosen. These are all alternatives of the `DynAction` type. A group is created with the `dynamicGroup` combinator:

```
:: DynAction = Extend [Task Void] | Stop

dynamicGroup :: [Task Void] [GroupAction Void] (Action → DynAction) → Task Void
```

It is applied to the tasks that initially belong to the group, the group actions, and the function that handles an action event. The group actions are similar to task actions, except that conditions on group actions do not depend on an edited value. Instead, they are either always possible or if a predicate on a shared state is valid. In the GUI there is a menu bar for the entire group for triggering those group actions. Optionally, there is also a toolbar for buttons triggering actions not included in the menu. So group actions are handled in the same way as task actions.

The final part to discuss is to decide the rendering of the tasks `replace` and `edit`. For this purpose we introduce a new instance for the task annotation operator `<<@`:

```
:: GroupedBehaviour = Fixed | Floating | Modal
```

By annotating a task as `Fixed`, it displays its content in a fixed window; `Floating` tasks can be moved around by the user, and `Modal` tasks are shown in a modal dialog, forcing the user the finish this task before any other. Floating tasks inside the group have their own menu bar and buttons which are generated as usual. Fixed tasks have no own menu bar. Therefore, buttons are generated for all task actions. A screenshot illustrating a fixed editor task and a floating replace dialog is given in Figure 2.

3.5 Dynamic Menus

Sometimes one wants to dynamically extend a menu structure. An example is to extend the text editor with a menu for re-opening recently opened files. We model such a dynamic menu structure as a view on a state. The menu annotation (see Section 3.2) is extended with a third alternative. In this way the entire structure of the menu is determined dynamically:

```
:: MenuAnnotation = ... | ∃m: DynamicMenus (DBId m) (m → Menus) & iTask m
```

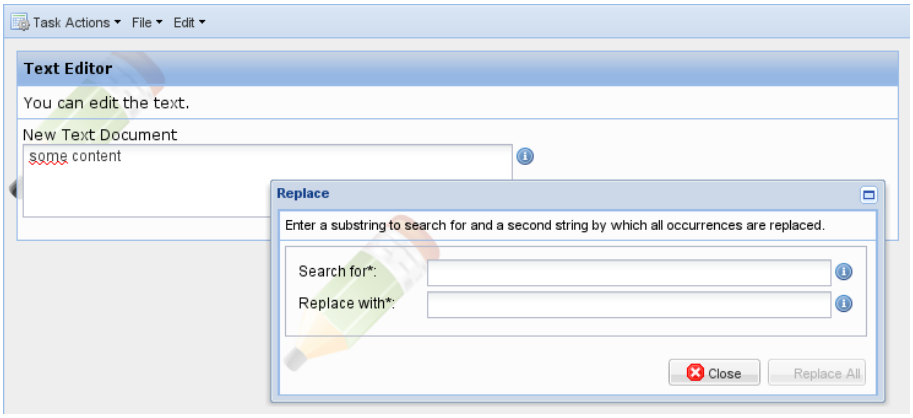

Fig. 2: A screenshot of the text editor GUI application

As an example, we extend the text editor with the above mentioned menu. We first extend the state with a history of recently opened file names:

```
:: State = State Note FileInfo [FileName]
```

We deploy the (`ActionID`,`ActionLabel`) instance of the `menuAction` class to pass the correct file name to be opened with an action with ID `"openFile"`:

```
dynamicMenus ref (λ(State _ _ history) →
  [ Menu "File"  [ MenuItem ActionNew  (hotkey N)
                 , MenuItem ActionOpen (hotkey O)
                 , SubMenu "Recently opened" (recentItems history)
                 , ...
  ])
recentItems hs = [MenuItem ("openFile", fname) Nothing \\ fname ←- hs]
```

4 Mixing Workflow and GUI

In the previous section we have presented the new features of *iTask* that are concerned with programming GUI applications. We emphasize that this has been done in an orthogonal way: existing *iTask* workflow definitions do not change because of these extensions. However, it is now possible for workflows to use the extended functionality. For instance, tasks providing views on the same shared state can be assigned to different users. This approach gives multi-user functionality for free. To illustrate this, we give an example of a simple chat program:

```
1 chat = createDB (Note "", Note "") >>= λref →
2   (NamedUser "user1" @: updateShared "Chat" "" [(ActionQuit,always)] ref [editor1])
3   -&&-
4   (NamedUser "user2" @: updateShared "Chat" "" [(ActionQuit,always)] ref [editor2])
```

```
5 where
6   editor1 = view { viewFrom = λ(note1, note2) → (Display note2,note1)
7                  , viewTo   = λ(_,note1) (_, note2) → (note1, note2) }
8   editor2 = view { viewFrom = λ(note1, note2) → (Display note1,note2)
9                  , viewTo   = λ(_,note2) (note1, _) → (note1, note2) }
```

There is a shared state containing two separate notes each user can edit (line 1). Two tasks assigned to different users are started (line 2, 4). Those tasks provide views on the same state (lines 6–9). The special constructor `Display` is used to make sure that one user can only see but not edit the text of the other user. In this way editing conflicts are prevented. A screenshot of the generated GUI is given in Figure 3.

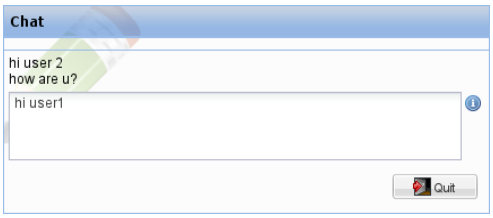

Fig. 3: A screenshot of the chat example

5 Related Work

There already exist many proposals and libraries for defining GUIs in a functional language. A detailed comparison is out of scope of this paper. Instead we choose a number of approaches that characterize a programming paradigm.

A large number of approaches have adopted the traditional widget-callback GUI paradigm (e.g. *Object I/O* [1] and *wxHaskell* [9]). Here, the programmer is responsible for the entire life-cycle of the GUI elements: creation, management, event handling, and destruction. This style of programming mixes up the visualisation, the program logic, the processed data, and the current state of the application. If we compare these approaches to the enhanced *iTask* approach, we conclude that the resulting code of the former approaches is harder to understand and therefore harder to maintain.

Other approaches make better use of the level of abstraction provided by functional languages. Programmers compose the user interface in a declarative way using basic elements and are freed from explicitly managing them. There are different approaches for realizing interaction between different components. In *Haggis* [6] each component is treated as a virtual I/O device. User events are represented by messages generated by components like buttons. Additionally, a separation between the user interface and the application, i.e. between the representation and the actual value or interaction with the user, is made. *Fudgets* [4]

uses the model of *stream processors* to represent GUI elements. They pass messages and are hierarchically combined to build up the application. There is no explicitly shared state. State is realised by routing messages between components. An even more formal model of continuous time-varying *signals* transformed by pure *signal transformers* is used by *Fruit* [5]. A drawback of the formal model used by *Fruit* is that all kinds of I/O must explicitly be added to the input and output signal. The *iData toolkit* [11] models web-applications as interconnected forms. *Generic programming* techniques [2] are used to automatically generate forms for editing values for any type. This allows for a way of modelling applications which abstracts from the visualisation.

With all those approaches the programmer has to handle the application's execution state manually. Having the possibility to define the control flow in a declarative way is a unique advantage of the *iTask* paradigm.

6 Conclusions and Future Work

We have shown that the workflow-based *iTask* system can be used as a paradigm for building GUI applications. Because the *iTask* system was originally designed for managing workflows, this involved a significant implementation effort to add the required features. One reason is that dealing with tasks sharing data generates extra dependencies between them which required changing the way tasks are calculated. Also the way the state of parallel tasks is handled had to be changed since tasks can be added dynamically. A last point is that on each change not the entire GUI is replaced but update instructions generated by the server are used to adapt only changed parts (details are explained in [10]). Form fields being changed by the underlying data model and dynamically added forms also required extending this mechanism.

One design goal was to retain the level of abstraction as offered by the *iTask* system. When defining GUI applications, programmers only have to define the processed data and the application's control flow in the same spirit as with the 'original' *iTask* system. They are freed from manually handling the application's execution state. Dependencies between the application's state and the user interface are declaratively defined using predicates and views. The system automatically creates proper GUIs, stores intermediate results, and keeps track of the execution state.

Another advantage is that applications are embedded in the web-based *iTask* WFMS. All work the user performs is synchronised with a server. The user can stop working at any moment and continue on any (other) computer. Also, applications can be used as part of a workflow, and, vice versa: workflow functionality can be used by GUI applications. In this way implementing multi-user GUI applications comes for free.

Although generically generated GUIs are very powerful, for some applications one might want to influence the layout more precisely. One direction for future work is to develop a way to influence the layout with more fine-grained annotations in the same spirit as done for menus.

We want to conduct more case studies. Examples are sophisticated GUI components, and applications that use huge amounts of data such as spreadsheet applications. Also, multi-user applications are good candidates since one gets much functionality from the WFMS for free. Multi-user applications more easily lead to editing conflicts. A good error reporting and recovery system has to be explored.

Acknowledgements We would like to thank the anonymous referees and Bas Lijnse and Thomas van Noort for their valuable comments.

References

1. Peter Achten and Rinus Plasmeijer. Interactive functional objects in Clean. In Chris Clack, Kevin Hammond, and Tony Davie, editors, *Selected Papers of the 9th International Workshop on the Implementation of Functional Languages, IFL'97*, volume 1467 of *LNCS*, pages 304–321. Springer-Verlag, September 1998.
2. Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005.
3. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.
4. M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.
5. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.
6. Sigbjörn Finne and Simon Peyton Jones. Composing the user interface with Haggis. In *Advanced Functional Programming: Second Interational School, LNCS #1129*, pages 26–30. Springer-Verlag, 1996.
7. Pieter Koopman, Peter Achten, and Rinus Plasmeijer. Validating specifications for model-based testing. In Hamid Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Research and Practice, SERP'08*, pages 231–237, Las Vegas, NV, USA, 14-17, July 2008. CSREA Press.
8. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August-September 1988.
9. Daan Leijen. wxHaskell: a portable and concise gui library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM.
10. Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for End-users. In Marco Morazán and Sven-Bodo Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL'09, South Orange, NJ, USA*, volume 6041, pages 36–54. Springer-Verlag, 2010.
11. Rinus Plasmeijer and Peter Achten. iData for the world wide web - programming interconnected web forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), volume 3945 of LNCS*, pages 24–26. Springer Verlag, 2006.