Building iTask Applications

A GUI Paradigm Based on Workflows

Master Thesis Computer Science

Steffen Michels

August 2010

Supervisors: Rinus Plasmeijer and Peter Achten

Radboud University Nijmegen, The Netherlands
Institute for Computing and Information Sciences

Research Number: 639

Abstract

The *iTask* system [25] is a *workflow management system* (WFMS) using a *workflow description language* (WDL) embedded in the *functional general purpose programming language Clean*. The WDL is declarative in the sense that only process and data and no representation is defined. A webapplication for carrying out workflows is automatically generated using *Clean*'s *generic programming* [3] facilities. In this thesis the *iTask* system is extended to support building of office-like applications. The result is a novel paradigm for implementing GUI applications based on workflows. It is shown that those applications can be described in terms of *grouped tasks* using a uniform concept of *shared data* to communicate. Additionally a menu system is added. A multi-file text editor and a prototype showing essential concepts for realising an *integrated development environment* (IDE) for *Clean* are implemented using those concepts.

The extended WDL stays orthogonal to the basic *iTask* WDL and fits into its task based concept. Also information included in the extended WDL not dealing with process or data are kept minimal. Programmers are released from using callbacks, keeping track of the application's controlflow and managing GUI elements manually, like in most widget-based approaches. Also functional concepts like higher-order tasks turns out to be very suited for building a general *multiple document interface* (MDI) infrastructure and generalising GUI patterns.

Acknowledgements

First I thank Rinus Plasmeijer for supervising my previous research leading to the idea for this master thesis and also for supervising my thesis and spending much time on discussing my progress during the last half year. Then I want to thank Peter Achten for also supervising my thesis, especially for critically reading it and bringing in expertise about *Object I/O* in particular and functional GUI programming in general. Further I would like to thank Bas Lijnse for helping me to keep an overview of the principles and the code of the *iTask* system and for giving constructive criticism about the extensions I added to it. Then my thanks goes to Erik Crombag for giving me hints about implementation details especially for the client-side, like the group panel and the source code control, and for discussing new concepts I added. Finally I thank John van Groningen for explaining to me how external processes can be called from *Clean* and showing me details about the compilation process of the *Clean* IDE.

Contents

List of Figures

Chapter 1

Introduction

Workflows are used to define relationships between tasks to achieve a certain goal. They define which tasks depend on each other, in which order they have to be performed and who has to do them. The *iTask* system [25] is a *workflow management system* (WFMS), which uses a *workflow description language* (WDL) to describe such workflows.

The *iTask* WDL is embedded in the *functional general purpose programming language Clean*. This approach has the advantage that arbitrary computations, recursion and also functional concepts like higher-order tasks are added to the WDL. So tasks can recursively call themselves and accept or produce functions or other tasks. Additionally all tasks are *strongly typed*, since *Clean*'s type system is used. Each task accepts and produces typed values.

iTask makes it possible to create interactive multi-user web applications to carry out the workflows. For this another powerful feature of *Clean* is used which is *generic programming* [3]. This makes it possible to automatically generate web interfaces to edit data values of any type.

Although originally designed for workflows, the *iTask* system can be seen as a more general I/O paradigm for implementing applications with *graphical user interfaces* (GUI). There are several differences with traditional approaches, like the imperative platform independent libraries *wxWidgets*[1] or *Java's Swing*[2] and also functional approaches like *Object I/O* [1, 2] or *wxHaskell* [18] (also discussed in Chapter 7).

Those paradigms are based on callback functions. The programmer has to split up the program logic and manually handle the current state to coordinate consecutive callbacks. The *iTask* WDL is designed to describe the program's control flow using standard *workflow patterns* (identified by van der Aalst et al. [28]), like a sequence or a collection of parallel tasks, or recursion. An important property of this language is that a semantics can be given to it [16] which might help to reason about programs written in that language.

Also compared to more functional approaches not based on callback functions like *Fudgets* [6, 7] or *Fruit* [9, 10] a unique property of the *iTask* approach that it is based on workflow semantics. We think that this kind of semantics is more suited for expressing the application's control flow from a higher level than for example the model used by *Fruit*, which is based on *signals transformers.* (Details are discussed in Chapter 7.)

Virtually all paradigms are based on widgets which explicitly have to be created, managed and destroyed by the programmer. This is a great source of errors. In more functional approaches like *Fudgets* or *Fruit* widgets can declaratively be described

[1]`http://www.wxwidgets.org/`
[2]`http://java.sun.com/`

and combined as long as the user interface is static. For dynamically changing user interfaces widgets also have to be created and removed. The *iTask* paradigm completely abstracts from widgets and only deals with editing data of a certain type. Generic algorithms are used to automatically generate GUIs for editing this data. This allows for a more abstract view of the process and can save much time when writing applications.

Although using standard controls also makes it possible to implement platform independent libraries, the level of abstraction of the *iTask* WDL allows for generating user interfaces even more independent from the kind of client. The actual representation of the user interface could depend on the device on which the interface is rendered, which can be a normal PC but also a mobile device with a small screen. The program may even be used by blind persons using a Braille display, since the WDL description only defines what to do (for example which data has to be entered) and not how this is done. This level of abstraction is also suited for describing interaction with other computers systems. A GUI is not needed here, only data of the correct type has to be provided. Currently the *JavaScript* library *ExtJS*[3] is used to render the GUI inside a browser on the client-side, but the system is designed in such a way that the user interface could in principle be rendered by any client.

Recently more and more office-like software is realised as web application. One example for this is *Google Docs*[4] which includes for instance a web based word processor, spreadsheet and presentation application. A client running in a browser has the advantage that the resulting applications are platform independent. Also one does not have to install special software on the client computer and the data the user works on is available everywhere since it is stored on a server (or a cloud of servers). In the *iTask* system the entire current state of a workflow is always synchronised with a server. At each moment working on a task can be stopped and continued later, also on a different computer.

Web technology originally was not designed for such kinds of applications. Although it is shifting more and more in a direction which makes it possible to implement applications as powerful as traditional offline software, it is still important to provide the programmer with a formalism, like the *iTask* WDL, which abstracts from tricky issues like keeping the state of the application.

A last unique property of the *iTask* paradigm is that it is embedded in a WFMS. This makes it possible to add applications as part of workflows. Also workflow functionality like assigning tasks to different users and giving them a priority and a deadline can be used inside applications. This suggest that this approach is highly suited for realising multi-user applications.

Although past experience with using *iTask* for a non-workflow interactive application turned out to be successful [15], the *iTask* system lacks some features needed to realise modern user interfaces. The goal of this master thesis is to extend *iTask* in such a way that it can be used to generate user-friendly applications. The challenge is to make *iTask* powerful enough to create arbitrary office-like applications, but keep the task semantics and declarativeness of the WDL as intact as possible.

We also expect some drawback of having a GUI paradigm with that high level of abstraction. The disadvantage is that the programmer has less control over how the application will look like and can realise less interactivity than with approaches working on a lower level of abstraction, like widgets or even pixels. If the programmer builds the GUI by putting widgets together it is possible to influence the layout. In the case the programmer can only tell the system that the user should edit a value of a certain type the layout is determined automatically by the generic

[3]`http://www.extjs.com/`
[4]`http://docs.google.com/`

algorithm.

The best method to show that a system is usable for creating applications is to do so. So we will use two case studies which we are going to implement using the *iTask* system. We start with a multi-document text editor. Although this is a relatively simple application, its realisation requires to solve the more general problem of implementing a *multiple document interface* (MDI) application. A more sophisticated application is an *integrated development environment* (IDE) for *Clean*. This is very useful since the current version is restricted to *Windows* only. The new IDE would run in a browser and will therefore be platform independent. Additionally it will be easy to add facilities for multiple users working together. In this thesis not a full implementation of an IDE is given, but important problems like calling a compiler and highlighting syntax are solved. This suggest that this approach is powerful enough for this complex kind of application.

Summarised the contributions of this thesis are:

- Missing concepts needed to realise modern GUIs are identified and added to the *iTask* WDL.
- The *iTask* WFMS is extended to support those new concepts.
- It is shown how common GUI concepts, like MDI applications, can be expressed in terms of the workflow-based WDL.
- An implementation of a MDI text editor, which is a typical office-like application, and also of some features needed for an IDE is given.

This thesis is organised as follows. Before focussing on how the system has to be extended first in Chapter 2 an overview of the architecture of the *iTask* system and its WDL is given. Then in Chapter 3 general requirements about the extensions are discussed. This includes a discussion of important capabilities needed for realising modern GUIs and how they fit into the *iTask* approach. Also possible problems and limitations of this approach are discussed. All extensions of the *iTask* system are described in detail in Chapter 4. In Chapter 5 the case studies described above are used to show the suitability of the extended system for this kind of applications. The extended WDL and the experiences with the case studies are evaluated in Chapter 6. The results are compared to related work in Chapter 7. Chapter 8 concludes the thesis and gives some ideas for future work. Finally in Appendix A a short reference of the *iTask* WDL is given.

Chapter 2

iTask Architecture

Before focussing on extending the *iTask* system in this chapter an overview of the general architecture, as described in [19], is given. First the structure of the *iTask* WDL is discussed in Section 2.1. Then in Section 2.2 an overview of the system for executing workflows defined in this WDL is given. Because automatically generating user interfaces for editing data of arbitrary type is one of the most sophisticated parts of the system, special attention is paid to it in Section 2.3.

2.1 The iTask WDL

The *iTask* system comes with a language for declarative descriptions of workflows. Declarative here means that only the process and the data involved are described. All other details such as how data is stored or represented by the user interface are handled automatically. The language consists of two building blocks: *basic tasks* and *task combinators*. The result of a combinator is a task again. In this way complex workflows can be build by repeatedly combining tasks to more complex ones. A workflow therefore is simply a task. Because the language is embedded in *Clean* all kinds of computations and data structures can be added inside a workflow specification.

Here only the general structure of the WDL is discussed. An overview of available basic tasks and combinators is given in Appendix A.

2.1.1 Basic Tasks

Basic tasks are the smallest unit of work that can be done like entering data or writing data to a database. Some examples are given below. This is only a small collection of simple basic tasks. A more complete overview of available basic tasks is given in Section A.1.

```
enterInformation    :: question      → Task a       | html question & iTask a
showMessageAbout    :: message  a    → Task Void    | html message  & iTask a
writeDB             :: (DBid a) a    → Task a       | iTask a
```

Each task has type `Task a` which means that it returns a result of type `a`. The first task asks the user to enter some information. Which information can be entered is determined by the type `a` and therefore depends on the context in which the task is used. The user can enter information of any type which can be defined in *Clean* as long is the context restriction `iTask` is fulfilled. It consists of generic functions for storing data, generating forms, updating values and generating error and hint messages:

```
class iTask a
    | gPrint {|*|}
    , gParse {|*|}
    , gVisualize {|*|}
    , gHint {|*|}
    , gError {|*|}
    , gUpdate {|*|}
    , TC a
```

Those functions are discussed later. The class `TC` (type code) indicates that values of this type can be packed into *dynamics*[1].The class `html` is used for questions and messages, which are converted to a *HTML* representation before they are sent to the client. Most of the time this will be a simple string, but also a type for describing arbitrary *HTML* constructions is provided.

Another task which requires user interaction is `showMessageAbout` which shows a message together with a value of arbitrary type to the user. The task has en empty result which is indicated by the type `Void`. There are also tasks which require no user interaction. The task `writeDB` writes a value (second argument) to a store identified by a reference (first argument) and finishes immediately. The example of those basic tasks shows that everything specified is **what** has to be done and not **how** it is done.

2.1.2 Combinators

To define complex workflows basic tasks can be put together to *combined tasks* by using combinators. Again here only the idea is explained and a more detailed overview is given in the appendix (Section A.2).

One of the most important combinator is the *monadic bind combinator*:

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

This combinator first executes the first task. After the first task is finished it gives the result to a second one. The result this second task returns is given as result of the combinator. The combinator is a task itself, which means that it can again be combined with other tasks using combinators. In this way for example sequences of arbitrary length can be defined using the bind combinator repeatedly. Examples for other combinators are combinators to execute tasks in parallel or assign tasks to other users.

Example

The following example demonstrates how the bind combinator is used:

```
bindExample :: Task Void
bindExample = enterInt >>= showMessageAbout "you entered"
where
    enterInt :: Task Int
    enterInt = enterInformation "enter integer"
```

First the user can enter an integer. It is necessary to indicate the type of `enterInt` because somewhere it has to be specified what kind of data is entered by the user. The integer entered by the user is given to `showMessageAbout`. This time the compiler can automatically derive the type of the argument given to this function. So the user gets to see the integer she just entered. The task `bindExample` can be used as workflow which can be started by the user directly, but can also be combined with other tasks to build a more complex workflow.

[1] *Clean* provides a hybrid static/dynamic type system. Nearly all values (including functions) can dynamically be packed and unpacked at runtime [29].

The remarkable thing here is that the author of such a workflow can just specify a sequence of actions in a very natural way. It seems as if execution of the function stops at the first task as long as the user is working on it and then continues with the second one. The system automatically keeps track of the current execution state of the workflow and restores it if a request of a client arrives.

2.1.3 Adding Computations

The fact that the language is embedded in *Clean* makes it even more powerful, because this makes it possible to add all kinds of computations inside of workflow definitions.

Example

How this can be done is shown by the following example:

```
computationExample :: Task Void
computationExample =
            enterInformation "enter integer"
    >>= λi. showMessageAbout "Fibonacci number" (fib i)
where
    fib 0 = 0
    fib 1 = 1
    fib n = fib (n-1) + fib (n-2)
```

The user first enters an integer, like in the previous example. This time the type of `enterInformation` does not have to be given explicitly, since it can be derived by the compiler automatically. The type is determined by the context in which the result is used later, that is as argument of `fib`. This functions is just the naive functional definition of the *Fibonacci numbers*. The computed Fibonacci number is shown to the user. This shows that arbitrary computations can be used inside workflows.

2.1.4 Specialised Types

Data structures are always composed out of basic types in *Clean*. For some datastructures special widgets could be used to let the user edit the value in a more convenient way. A number of specialised types are included in the system. The most commonly used are collected in the common domain described in Section A.3.

The idea of those specialised types is that they only define the kind of the data. Nothing is specified about the representation. Therefore it belongs to the description of the workflow's data. How those types are actually represented by the GUI is determined by the generic algorithm and mainly by the client.

Example

For instance a date could be defined as follows:

```
:: Date =  { day   :: Int
           , mon   :: Int
           , year  :: Int
           }
```

It would not be very user friendly and could lead to invalid dates to let the user edit the tree integers directly. So a special widget, depicted in Figure 2.1, is used if a value of type `Date` is edited.

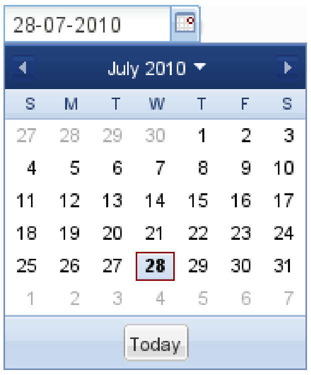


Figure 2.1: The Widget Used for Editing Dates

2.1.5 Exceptions

Exceptions are a powerful feature to handle errors in a way that separates the handling of errors from the program logic. Because of this exceptions have been added to the *iTask* language. Each task can throw an exception if an exceptional situation occurs using the following task:

```
throw :: e → Task a | iTask a & TC e
```

Any value which can be packed into a dynamic can be used as exception. The task itself returns a value of arbitrary type to make it combinable with other tasks in all contexts. Actually the task never produces a result. As soon as an exception is thrown it moves its way up the calling stack until it is caught or the top of the stack is reached. This is the expected behaviour for exceptions.

To catch an exception the task in which an exception is thrown has to be surrounded by this task:

```
try :: (Task a) (e → Task a) → Task a | iTask a & iTask e
```

If the task given as first argument finishes normally its result is returned as result of `try`. In this situation `try` has the same effect as the task given as first argument. If an exception of a type matching `e` occurs it is used to build a new task which then gives the result of `try` instead.

Examples

There is a problem with the example computing Fibonacci numbers given in Section 2.1.3. If the user enters a negative number the computation fails and in the worst case the entire server crashes.

The function for computing Fibonacci numbers can be replaced by a task doing the same computation in a safe way. It throws an exception if called with a negative argument:

```
fibTask :: Int → Task Int
fibTask i = if (i ≥ 0)
                (return (fib i))
                (throw "negative argument!")
where
    fib 0 = 0
    fib 1 = 1
    fib n = fib (n-1) + fib (n-2)
```

If the number is not negative the function `fib` as defined before is used. For this the basic task `return` is used which simply generates a task returning a given value.

Otherwise an exception is thrown. In this case it is a string but arbitrary types can be used as exceptions.

With this a new workflow for calculating Fibonacci numbers can be defined:

```
exceptionExample :: Task Void
exceptionExample =
          enterInformation "enter integer"
    >>= λi. try (showResult i) handleException
where
    showResult :: Int → Task Void
    showResult i =
          fibTask i
       >>= showMessageAbout "Fibonacci number"

    handleException :: String → Task Void
    handleException err = showMessageAbout "Error computing fib:" err
```

First the user can enter an arbitrary integer. The task `showResult` just computes and prints the Fibonacci number without caring about possible errors. If the user enters a positive number the workflow behaves as if `showResult` was used without the surrounding `try`. If the user enters a negative number the string exception is caught by `handleException`. The user gets to see an error message instead of the computed number.

Of course, there are more elegant ways to enforce properties on user input. The example is only meant to demonstrate the use of exceptions. More realistic examples are given in Sections 4.6.2 and 5.2.3.

2.2 Architecture Overview

As depicted in Figure 2.2 the *iTask* system consists of a server and a client application written in *JavaScript* running inside a browser. The client uses *Ajax* to access services the server provides. The server maintains a list of all workflow definitions and stores to keep track of user sessions and current instances of running workflows. For most of the services a so called task tree is computed as intermediate step.

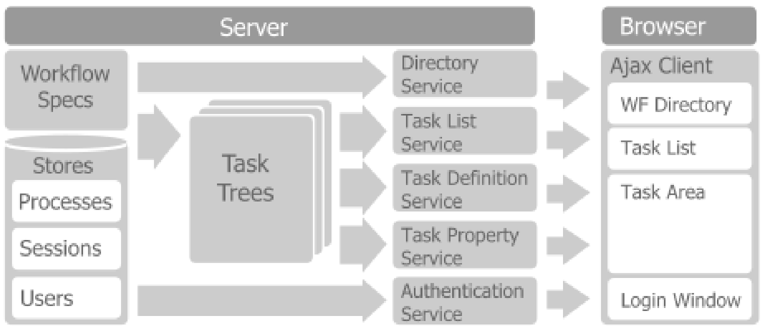


Figure 2.2: An Overview of the Client/Server Architecture of the *iTask* System (taken from [19])

2.2.1 Processes

In the *iTask* system a process is a portion of work a user is currently working on. A process is always generated if a user starts a new instance of a workflow, but can also be started by other processes.

Each process has its own identifier, a user who is currently working on it and can have a manager who can for instance suspend the process or assign it to someone else. Also additional information like the creation time, the priority or a deadline are assigned to each process. Relations between parallel processes can be defined hierarchically.

2.2.2 Server Services

The server makes a number of different services available to clients. They are briefly described here.

Authentication Service A client can start a session by authenticating to the server. For this a username and a password is required. Each user also can have several roles which can be used to restrict certain workflows to a group of users. The client can also log out and destroy a session.

Workflow Directory Service This service provides a list of all available workflows. Users can browse this list and initiate new workflow instances.

Tasklist Service A list with all work the user currently has to do is provided together with some information about parent/child relations.

Task Service This service is used to actually work on tasks. The server provides the client with a high-level definition of the user interface which is interpreted and rendered by the client. If the user changes something an event is sent to the server, which updates the current state of the task on the server and sends back updates for the user interface. The client either renders a new user interface or adapts the existing one. How this is done is described in more detail in Section 2.3.

Property Service Each workflow also has some meta data like the user to which it is assigned or its priority. Those meta data can be changed by this service.

2.2.3 Task Tree Computation

The task tree provides an intermediate representation of the structure and progress of tasks and separates this from their representation as a user interface. So the implementation of basic tasks and combinators only have to build a tree. This tree is then queried by the services to obtain the information necessary to handle the client's request. For instance the *Tasklist Service* only retrieves a list of all tasks a user is working on while the *Task Service* can use the tree structure to generate a GUI definition and send it to the client.

The leaves in a task tree represent basic tasks, while the nodes are compositions. There are several different types of leaves:

Interactive Task A task that can be worked on through a GUI. Such a leaf contains either a high-level definition of or updates for the user interface. Mostly tasks creating such a node are *interaction tasks* (Section A.1.1).

Monitor Task A task that upon evaluation monitors a condition and may give status output. It contains a *HTML* representation of the status. An example is a timer task which forces the execution of the workflow to pause for a certain amount of time. In this case the status shown to the user is the remaining time.

Instruction Task A task which displays an instruction, which is an *HTML*-message, to the user.

RPC Task A task that represents a *remote procedure call* invocation, containing a reference to the call.

Finished Task A finished task. Each task can eventually result in such a node. The only information that is still needed about a finished task is its result.

Those leaves can be combined by the following nodes in the tree:

Main Task A main chunk of work added at the top of each process. For parallel processes assigned to different users such nodes can also occur as child of other nodes in the entire hierarchy.

Sequence Task A sequence of consecutive tasks, for instance generated by the bind combinator.

Parallel Task A composition of tasks executed in parallel and carried out by different users.

Grouped Task A group of task running in parallel and carried out by the same user. (The distinction between grouped and parallel tasks is discussed in Section 4.3.)

2.3 User Interface Handling

Automatically generating user interfaces for editing arbitrary data types is one of the most sophisticated parts of the *iTask* system. For any type used inside a workflow a GUI which has a layout suited for representing the actual structure and reacts dynamically to user inputs without interrupting the user's work has to be generated. For this first the definition of the GUI has to be generated, then user-inputs have to be mapped back to the server and finally the GUI at the client side has to be adapted.

First *data paths* used to point to substructures inside of datastructures are discussed (Section 2.3.1). This is followed by the description of the concept of *data masks* in Section 2.3.2. They are used to keep track of which parts of the datastructure the user already gave a value to. The structure of the GUI definitions generated for data of arbitrary types is discussed in Section 2.3.3. In Section 2.3.4 it is explained how an event on the client side is used to update the value stored on the server. After this there may be the need to adapt the user interface (Section 2.3.5). Finally an example is given in Section 2.3.6.

2.3.1 Data Paths

Data paths are used to point to substructures inside of datastructures. A client can send an event telling exactly which part of the value to update in this way.

Data paths are basically just a list of integers defining a path through the generic representation of the datastructure. Additionally a sub-editor index, used for putting several views into one form (Section 4.2.4), can belong to it:

```
:: DataPath = DataPath [Int] (Maybe SubEditorIndex)

:: SubEditorIndex :== Int
```

The list of integers encodes the path of the datastructure from top of the generic tree to the element to which the path points. Each integer encodes the index for each constructor choice.

For a value of type `::Tree = Node Tree Int Tree | Leaf Int` an example of data paths is given in Figure 2.3. Each constructor choice is encoded by the constructor's index (starting from 0). So the root node of the tree has data path [0]. For each field of a constructor also the index (again starting from 0) is given. So the integer in the root node has data path $[0, 1]$. The 0 encodes the choice for the first constructor (`Node`) and the 1 encodes that the integer is the second argument of this constructor. For all other constructors and values in the tree a path can be determined analogously.

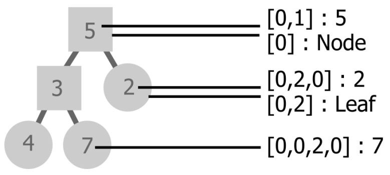


Figure 2.3: The Data Paths of an Example Binary Tree (taken from [19])

2.3.2 Data Masks

If the user edits data not all parts of the data structure might already have a value. For instance if the user adds a new element to a list of integers, before the user entered a valid value for the integer the field should be treated in a different way than field with a value already filled in. It should be blank instead of showing a value and should not be validated. This cannot be encoded in the edited value itself, because each field always has a value. Instead the notion of a data mask is used, defined as:

```
:: DataMask :== [[Int]]
```

Basically a data mask is a list of all data paths (defined as `[Int]`, because the sub-edit index is not needed here) which have been accessed by the user.

Example

Suppose that the user enters a value of the following type:

```
:: IntList = Nil | Cons Int IntList
```

Further suppose the current value is `Nil`. But then the user decides to add a new element to the list. The new value is therefore `Cons 0 Nil`. This is because 0 is the default value for integers. But actually the user did not fill in a number yet, so in the GUI there should be a blank field instead of a field showing `0`. This is encoded by not including the field's data path, which is $[1, 0]$, in the data mask. The first time the user enters a valid value, the field's data path is added to the mask.

2.3.3 Generating User Interface Definitions

User interface definitions are generated by the generic function `gVisualize`, which can generate high level user interface definitions which are *JSON*-encoded [11] and can be interpreted by the client. Also a *HTML* or text representation of a value can be generated for static elements and labels. The generic function can be derived for all *algebraic data types* (ADT) and also for records. Also many types like tuples or lists are treated in a special way to generate a better user interface. The following functions are used to wrap calls to `gVisualize` for the different purposes:

```
visualizeAsEditor       :: String (Maybe SubEditorIndex) DataMask a → ([TUIDef],Bool)
                                                               | gVisualize{|*|}
                                                               , gHint{|*|}
                                                               , gError{|*|} a
visualizeAsHtmlDisplay  :: a → [HtmlTag]                       | gVisualize{|*|} a
visualizeAsTextDisplay  :: a → String                          | gVisualize{|*|} a
visualizeAsHtmlLabel    :: a → [HtmlTag]                       | gVisualize{|*|} a
visualizeAsTextLabel    :: a → String                          | gVisualize{|*|} a
```

The first function is responsible for generating a definition for an editor which lets users edit a value of arbitrary type for which there is an instance of `gVisualize` which actually generates the definition. This generic function is also used for generating *HTML* and text visualisations as done by the latter four functions. Additionally there are generic functions for generating error messages and hints (`gError` and `gHint`) which are not discussed in detail here.

The first parameter of `visualizeAsEditor` is a name for the generated editor, then a subeditor index for the form can be given. Finally the value's data mask and the value itself is given to the function. The result is a list of user interface definitions and a flag telling if the editor is valid, which means that all fields are filled in and have a valid value. The reason why a list of values of type `TUIDef` (TUI means *"Task User Interface"*) is returned is that for a constructor with multiple arguments a GUI description for each argument is generated. GUI definitions are used to describe the user interface in an abstract way:

```
:: TUIDef   = TUIStringControl  TUIBasicControl
            | TUICharControl    TUIBasicControl
            | TUIIntControl     TUIBasicControl
            | TUIRealControl    TUIBasicControl
            | TUIBoolControl    TUIBasicControl
            | TUINoteControl    TUIBasicControl
            | TUIDateControl    TUIBasicControl
            ...
            | TUITupleContainer TUITupleContainer
            ...

:: TUIBasicControl =
    { name          :: String
    , id            :: TUIId
    , value         :: String
    , fieldLabel    :: Maybe String
    , staticDisplay :: Bool
    , optional      :: Bool
    , errorMsg      :: String
    , hintMsg       :: String
    }

:: TUITupleContainer =
    { id            :: TUIId
    , items         :: [[TUIDef]]
    , fieldLabel    :: Maybe String
    }

:: TUIId :== String
```

First there are controls for basic values like strings and integers, but also specialised ones like `Note` and `Date` which are also treated like basic values. They are all defined using the same information, encoded in `TUIBasicControl`, like a name, an identifier, the actual string-encoded value and some other information.

There are also controls, like `TUITupleContainer` for describing nested structures. In `TUITupleContainer` there is a field `items` which can contain a number of user interface definitions, representing the values stored inside of the tuple. The GUI description of each item is of type [`TUIDef`] since it can be a constructor with multiple arguments. Therefore the list of GUI descriptions is of type [[`TUIDef`]]. Such special containers are also used for nested constructions like lists and records.

The entire datastructure is encoded into a JSON-string and sent to the client which uses this abstract definition to render the GUI.

It is possible to hide parts of a datastructure in its visualisation by wrapping it with:

```
:: Hidden a = Hidden a
```

Also in an editable representation of a datastructure parts can be made static. Substructures inside of those parts can be made editable again:

```
:: HtmlDisplay a    = HtmlDisplay a
:: Editable a       = Editable a
```

For example if a value of type (`HtmlDisplay Int`, `String`) is edited, in the GUI a static non-editable integer is shown together with a field for editing the string.

2.3.4 Updating a Value

If the user changes a value this event is sent to the server where the current value of each editor is stored. In the current implementation this is done if a field loses focus, but also for example a timeout could be used as trigger. Each event includes the identifier of the main task, the data path of the changed value and a new value. Actually more than one value can be updated in one request, for instance if the user changes a field and unfocuses it by unchecking a checkbox. Then the event for changing the field and the checkbox are put together in one request.

The current value which is stored on the server is updated by the generic function `gUpdate`. The function goes trough the datastructure until it finds the corresponding data path and updates the value accordingly. The new value is stored on the server which keeps track of the current state of the workflow in this way.

2.3.5 Adapting the GUI

The GUI may have to be adapted after the user changed the value. For instance if a different constructor of an ADT is chosen the content of the old one has to be replaced by the content of the new one. For this `gVisualize` can also generate updates for the user interface by comparing the old with the new value. The generated list of updates is sent to the client which can interpret them and adapt the GUI. An advantage of this approach above redrawing the entire form is that the user can continue using the form without being disturbed when the answer of the server arrives.

The updates sent to the client consists of a list of update commands:

```
: TUIUpdate
    = TUIAdd        TUIId TUIDef
    | TUIAddTo      TUIId TUIDef
    | TUIRemove     TUIId
    | TUIReplace    TUIId TUIDef
    | TUISetValue   TUIId String
    ...
```

Some examples for such commands are adding additional components after (`TUIAdd`) or as child of (`TUIAddTo`) a component with given identifier. Also components can be removed (`TUIRemove`), replaced (`TUIReplace`) or given new values (`TUISetValue`).

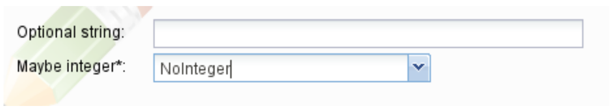
2.3.6 Example

Suppose that the user should enter a value of this type:

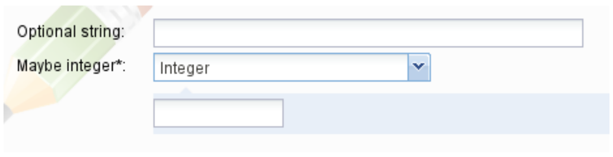
```
:: SomeRecord = { optionalString   :: Maybe String
                , maybeInteger     :: MaybeInteger
                }

:: MaybeInteger = NoInteger | Integer Int
```

The generically generated form shown in Figure 2.4a is in a valid state. The labels of the record fields are also used as labels in the form. The type `Maybe` is treated in a special way such that the first field becomes optional. No asterisk, indicating required fields, is displayed behind its label and it may be left blank. For the second field the constructor `NoInteger` is chosen.

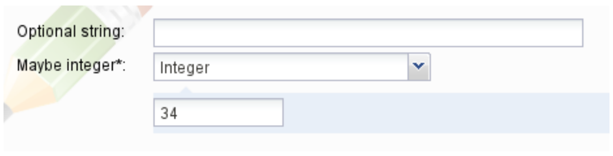
If the user selects the other constructor `Integer` the server generates an update instruction to add a field for entering the integer. The updated user interface is shown in Figure 2.4b. The field is still blank which means that its data path is not included in the data mask. Therefore the form is invalid because the second field is not optional. After the user fills in some integer (Figure 2.4c) the form is valid and the user can proceed.


Optional string:
Maybe integer*: NoInteger

(a)


Optional string:
Maybe integer*: Integer

(b)


Optional string:
Maybe integer*: Integer
34

(c)

Figure 2.4: An Example of a Generically Generated GUI for Editing a Record Structure

Summary

In this chapter the structure of the *iTask* WDL and the architecture of the WFMS executing workflow specified in this WDL are discussed. Special attention is paid to generically generating and updating GUIs, because this is one of the most sophisticated parts of the system and makes it possible to abstract from widgets in the WDL.

In the remainder of this thesis this system is extended to be usable as a paradigm for creating office-like GUI applications.

Chapter 3

Design Goals

Here first it is discussed which essential facilities of GUI libraries there are and are missing in the *iTask* language and to what extend they are added (Section 3.1). Then general principles which should be fulfilled for the new features added to the *iTask* language are given (Section 3.2).

3.1 Scope

There are a number of facilities which are essential parts for developing modern user interfaces. The focus of this research is to design a language for describing GUIs, that fits neatly into the task based *iTask* system. The goal is to get a language with a high level of abstraction. Necessarily this gives the programmer less control over the resulting application than with traditional libraries.

Consequently the system will not be suitable for all kinds of applications. It is not the intention to make it possible to implement highly interactive applications like games, but to give a better way to realise applications based on editing data using standard widgets. So the kind of applications that the new paradigm is suited for are office-like applications, like text editors, word processors and IDEs.

In the following it is discussed to what extend which facilities will be integrated into the *iTask* system:

Menus Menus are essential parts of nearly every user interface. They provide the user with a set of commands in a structured way. The commands are grouped using headings and a graphical representation of all available commands is provided.

Clearly menus are a concept important for the kind of applications that is intended to be realisable with *iTask* and will therefore be included.

Windows & Dialogs Windows and dialogs are the top level objects in most GUI systems. They include all the lower level objects like textfields, textareas and checkboxes. Complex programs contain several windows to separate data, which for example belongs to different opened documents.

It is one of the goals to extend the system in such a way that different parallel tasks can be visualised as separate windows.

Layouts Modern graphical applications aim at giving the user a user-friendly and aesthetic user interface. To achieve this ways to place the widgets the user interface is composed of are needed. Layouts are often a complex and sophisticated part of GUI toolkits.

One of the powerful features of *iTask* is that the programmer does not have to specify which widgets are needed in order to enter information. This is automatically solved by generic programming techniques. The goal of the *iTask* system is to provide a generic interface generating adequate GUIs for editing arbitrary datastructures. Because the programmer does not deal with widgets directly, there is no way to determine their layout. Also there is a tension between the approach to send an abstract user interface definition to the client which then determines how to render it and to give the server the possibility to influence the layout. Consequently this project does not deal with layouts.

Custom Widgets Some applications need widgets not included in the set of standard widgets provided by the library. Examples for this are textareas providing syntax highlighting, a widget for choosing a colour or an area for making drawings.

One already existing solution for this are *iEditors* [14] which uses *Java applets* to move parts of the functionality to the client-side. So the client runs code submitted to it instead of interpreting a high level GUI definition. This has the advantage that the only thing needed on the client-side is an interpreter for *Java applets*. The disadvantage is that the client cannot decide how to represent the GUI which is one of the advantages of having an abstract GUI definition. Also applets generate some overhead and do not fit well into the GUI.

Another solution, which fits better with the idea of sending an abstract GUI definition to the client, is to add new specialised types and extend the client with new widgets representing them. This is already done for a set of specialised types (Section A.3). The idea is to add more of those specialised types. The disadvantage is that this requires that all possible future clients will have to implement a suitable control for this special type. However currently there is only one standard client and therefore in this thesis this technique is used to realise for instance text areas doing syntax highlighting for *Clean* code.

General Canvas For instance for implementing games or programs for film editing the programmer needs to get more direct access to the computer's hardware instead of using widgets. This means that the programmer for example handles keyboard event and draws pixels directly.

A *general canvas* is not added to the system for several reasons. First it does not fit with the level of abstraction of the *iTask* WDL. Second the interface between the server and the client is not designed for realising that kind of interaction. Third using a browser as client restricts the possibilities to access hardware directly, although modern web technology developes into a direction which makes it possible to make drawings (for instance using the canvas element included in *HTML5*[1]) or even to use 3D graphics (using *WebGL*[2]).

Views Many applications provide the user with different *views* on the same data, for example a *What You See Is What You Get* (WYSIWYG) editor and a structured view on a document. This is known as *model-view-controller* (MVC) *paradigm* [17]. If data is changed in one view also all other views are updated. Most GUI toolkits do not have explicit support for views. The programmer has to modify the representation of all views manually if a shared model is changed.

[1]`http://dev.w3.org/html5/html-author/#the-canvas-element`

[2]`https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/WebGL-spec.html`

In an *iTask* workflow it is not possible to realise this manually because there is no way how grouped tasks, including different views, can communicate in a way which makes different views possible. So to implement applications using the MVC concept a way to define views on data shared among grouped tasks has to be added to the system.

3.2 General Design Principles

The *iTask* system is based on workflows which is a unique property of the new GUI paradigm. Consequently new facilities should never change the structure of the language in such a way that it has nothing to do with tasks any more. Trying to develop a GUI paradigm which is based on workflows, with the expected advantage that this results in more intuitive and abstract descriptions of user interfaces, is the main goal of this research. Another important point is that users of the *iTask* language expect a consistent interface.

One design goal of the *iTask* WDL was declarativeness. Lijnse and Plasmeijer define declarativeness in the context of an *iTask* workflow description as a specification defining only data or process [19]. Clearly this definition is too strict for a GUI paradigm. Even specifying that the user can give a command with a menu instead of a button or defining a menu structure is not declarative according to this definition. So in some cases it is necessary to define how a task is presented to the user. Information about the representation should nevertheless only be given were really necessary. Also they should stay as abstract as possible. Additionally this information should not interfere too much with the actual process description and should be kept minimal. In the best case it should only be added as annotations, which does not influence the functionality of the workflow, or they should clearly be separated from the process description.

Another nice property of the *iTask* WDL is that simple workflows definitions are very concise compared to solutions which would require creating widgets to edit data manually. This results in a low learning curve. This advantage should not be lost. Necessarily some GUI concepts will make workflow definitions more difficult to understand and write. Still this should not make workflow definitions which make no use of those new concepts more complex.

Summarised the following design principles will be used for all features added to the language:

- All new concepts should be integrated into the task structure of the *iTask* WDL.
- The language should stay as declarative as possible. Information not describing data or process should be kept minimal and interfere with the actual process as less as possible.
- The extensions should be orthogonal to the basic *iTask* paradigm. New features should not make the language more difficult to use for cases they where are not needed.

Summary

In this chapter the goal, the scope and general requirements for the extended *iTask* system are defined. In the remainder of this thesis those extensions are discussed in detail.

Chapter 4

iTask Extensions

To provide the features discussed in Chapter 3 the *iTask* system is extended. Those extensions are described in detail in this chapter.

Actually three new basic concepts are added to the system. The first one is the possibility to use menus to generate *actions* (Section 4.1). The second one is the concepts of different views on shared data (Section 4.2). Finally the third one are grouped tasks which can be rendered in different windows (Section 4.3). Also tasks can be added to those groups dynamically without interfering with other ones.

On those concepts a high-level combinator for MDI functionality is built (Section 4.4). Also the set of specialised types of the system is extended with types for formatted text, source code and colours (Section 4.5). Finally some new tasks for accessing operating system functionality like calling processes are added (Section 4.6).

4.1 Menus

Menus are essential parts of nearly every complex user interface. It allows the programmer to give a structured overview of available commands. Those commands are called actions in *iTask* terminology and were originally added to handle button inputs. The goal is to integrate menu actions neatly in the already existing concept. There is no essential difference between a button being pressed and a menu item being selected. So menus are just another mechanism to generate actions.

In traditional I/O paradigms the programmer has to take care that the menu always shows options applicable to the current context by adapting the structure and enabling or disabling entries in the menu. This is contrary to the design goal that everything should be as declarative as possible. In the solution described here the goal is that the programmer only has to specify a global structure of the menus and define which actions are possible for each context. This has the advantage that menu items do not have to be enabled or disabled explicitly, because this is done by the system automatically.

Menus have a structure which is separated from the actual actions triggered by menu items. Because menus are based on the idea of actions generated by buttons, first in Section 4.1.1 the concept of actions is explained. In Section 4.1.2 a datatype for defining the structure of menus is introduced. How this structure is applied to a workflow and can dynamically be changed during executing is described in Section 4.1.3. After the structure has been discussed it is focused on how interaction tasks are extended to be able to use button and menu actions (Section 4.1.4). Finally a mechanism for passing parameters with actions is discussed in Section 4.1.5.

4.1.1 Actions

Before menus were added to the system there was already the possibility to specify different actions for interaction tasks. Those actions can be triggered by the user by pressing buttons. The observation is that actions are a more general concept which could also be used to represent menu events in the WDL. For the process description there is no difference between a button being pressend and a menu item being clicked.

Actions are defined as follows:

```
:: Action  = ActionLabel String
           | ActionIcon String String
           | ActionOk
           | ActionCancel
           | ...
           | ActionParam String String
```

There are several predefined actions for common cases like `ActionOk` or `ActionCancel` for which the label and icon of the button is derived automatically. Also the programmer can specify custom actions with or without an icon for the generated button (`ActionLabel` and `ActionIcon`). The constructor `ActionParam` is used for parameter passing which is discussed in Section 4.1.5.

4.1.2 Structure of Menus

Most applications have a number of top-level menus with menu items triggering actions. Also submenus can be used inside of the menu structure. Finally items can be grouped using separators. An ADT is used to enumerate the elements of the menu, because each menu has a relatively simple uniform structure:

```
:: Menu     = Menu String [MenuItem]
:: MenuItem = MenuItem      String Action (Maybe Hotkey)
            | SubMenu       String [MenuItem]
            | MenuSeparator
            | MenuName      String MenuItem
```

What is described here is a menu bar with several top-level menus. Each entry in the menu bar is represented by one instance of `Menu`, which has a name and contains a number of menu items. Each menu item is either an entry which triggers a given action, a submenu which contains another list of items or a separator. Additionally `MenuName` can be used to give a part of a menu a name which can be used to modify it independent from the structure in which it is embedded, as will be explained in Section 4.1.3. An example of how such a structure will look like on the client is given in Section 4.1.4.

A menu item may additionally be triggered using a hotkey defined by the following type:

```
:: 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 | P | Q | R | S | T | U | V | W | X | Y | Z
```

For each hotkey one letter key is required. Additionally for the control, the alternate and the shift key it can be specified if they have to be pressed to trigger the action assigned to the hotkey.

4.1.3 Setting & Dynamically Changing Menus

In contrast to buttons the structure of a menu is relatively stable during the run of an application. Specifying the structure in each context where it is needed would clutter up the code. Further the menu's structure is neither process nor code. So the part of the workflow dealing with it should be kept minimal.

Because an (single-user) application will run inside one process in the *iTask* system, the structure of the menu is set for an entire process. At each moment there is one menu structure assigned to each process. The structure is used to construct a concrete menu bar for the current context, as discussed in Section 4.1.4.

Tasks to get, set and remove the entire menu system for the current process are added to the *iTask* WDL:

```
getMenus    :: Task (Maybe [Menu])
setMenus    :: [Menu] → Task Void
removeMenus :: Task Void
```

Initially each process starts with no menu. Consequently workflows not using menus will never have to use one of those tasks. This corresponds to the design goal that extensions should be orthogonal to the basic *iTask* WDL.

Although `setMenus` already makes it possible to dynamically change the structure of the menu, sometimes it is more convenient to change only a small part of the menu instead of the whole one. For example one might want to add files to a submenu opening recently opened files without worrying about where in the menu this submenu is located. For situations like this it is possible to give a name to a part of the menu structure (using `MenuName`), as already discussed. For changing parts of the structure the following tasks can be used:

```
setMenuItem :: String MenuItem  → Task Void
getMenuItem :: String           → Task (Maybe MenuItem)
```

Programmers should not use the same name twice. If they do for performance reasons only the first occurrence of the name is taken into account by those two tasks.

Example

A submenu for recently opened files can be defined like this:

```
MenuName "recOpened" (SubMenu "Recently Opened" [])
```

This submenu can be placed at an arbitrary place inside the menu structure. The name of an opened file can be added to the submenu by the following task which can abstract from the position of the submenu inside the entire menu structure:

```
addToRecentlyOpened :: String (DBRef TextFile) → Task Void                        1
addToRecentlyOpened name (DBRef id) =                                              2
              getMenuItem "recOpened"                                              3
  >>= λitem. case item of                                                          4
        Just (SubMenu label entries) = setMenuItem "recOpened"                     5
            (SubMenu                                                               6
                label                                                              7
                [MenuItem name (ActionParam "open" (toString id)) Nothing:entries] 8
            )                                                                      9
        _ = return Void                                                            10
```

First the current state of the submenu is retrieved by its name (line 3). Then the new file identified by its name and identifier is added to the submenu (lines 5 – 9). In the case the submenu does not exist nothing is changed (line 10). How the special action `ActionParam` is used is explained in Section 4.1.5.

4.1.4 Actions & Interaction Tasks

The only kind of tasks for which it makes sense to let them generate actions are interaction tasks. The reason for this is that those tasks interact with the user. One exception are menus attached to a group of tasks, which is discussed later in Section 4.3.4.

In a first approach interaction tasks with additional parameters are used to indicate possible actions. Actually two lists are used. For actions in both lists buttons are generated. The buttons in one of them are only enabled if the editor is in a valid state.

For this thesis this idea was extended to include also menu actions. Additionally a condition indicating in which situation an action is possible is added:

```
:: TaskAction a = ButtonAction          (ActionWithCond a)
                | MenuAction            (ActionWithCond a)
                | ButtonAndMenuAction   (ActionWithCond a)
                | MenuParamAction       (String, ActionCondition a)

:: ActionWithCond   a :== (Action, ActionCondition a)
:: ActionCondition  a  = Always | IfValid | Predicate ((EditorValue a) → Bool)
:: EditorValue      a  = Invalid | Valid a
```

A list of `TaskAction` values is used to indicate possible actions for a task. The first constructor `ButtonAction` just generates a buttons.

The situation for menu actions is a little bit more complex because a menu structure with items, triggering actions, is already defined for the process. The idea is that not for all situations each action might have a meaning. For some tasks shown to the user only a restricted set of actions might by applicable. Because of this a `MenuAction` entry in the list means that this action is accepted by the task. The actual menu bar is constructed for each task and contains only items producing actions the task accepts.

Also an abbreviation for adding the same action as button and menu action is provided (`ButtonAndMenuAction`). The constructor `MenuParamAction` is used for parameter passing with actions and will be explained in Section 4.1.5.

For each entry a condition is given under which the action can be triggered (`ActionCondition`). Items generating actions not applicable at the moment are disabled in the GUI. The action can always (`Always`) be triggered or only if the editor is in a valid state (`IfValid`). Also a predicate on the editor's current value can be used.

So for all interaction tasks defined in Section A.1.1 a list of actions should be attached. From a process description point of view, if ignoring how those actions are triggered, they define which actions can be taken in that context in a declarative way. To achieve the goal of being orthogonal to the basic *iTask* WDL the old versions are kept and for all tasks an additional version is added. There are some exceptions for which it makes no sense to define another set of actions, like `requestConfirmation` or `showStickyMessage`. An example of a variant of an interaction task is:

```
enterInformationA :: q [TaskAction a] → Task (Action,a) | html q & iTask a
```

Compared to the normal `enterInformation` there is an additional parameter which is the list of actions. Also the result of the task includes the action the user triggered to end the task.

The approach to add additional versions of interaction tasks doubles their number. But the goal to keep workflow definitions readable for cases no different actions are needed is more important than reducing the number of tasks. Actually the same choice is make for the versions of tasks with additional context information. They could be added as optional parameter which would reduce the number of interaction tasks to ten. But the about 35 variants of interaction tasks in the current version lead to code which is more readable for simple cases.

Example

Assume that the following menu structure is set for a process using `setMenus`:

```
[Menu "menu 1" [
    MenuItem "item 1" ActionOk      Nothing,
    MenuSeparator,
    MenuItem "item 2" ActionCancel  Nothing]
]
```

If none of the two actions is in the action list of an interaction task, a disabled menu entry in the menu bar is created (Figure 4.1a). In the case the list is `[MenuAction (ActionCancel,Always)]` only the item that generates `ActionCancel` is shown in the menu (Figure 4.1b). If the current editor does not have a valid value a menu like in Figure 4.1c is generated for the following action list:

```
[MenuAction (ActionCancel, Always), MenuAction (ActionOk, IfValid)]
```

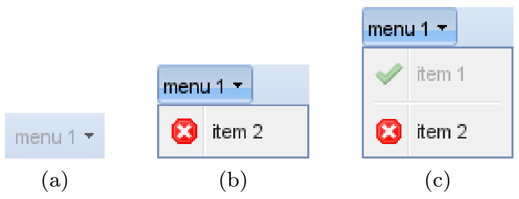


Figure 4.1: Examples of Menus for Different Contexts Generated from the Same Structure

The icons shown in the menu are predefined for the default actions `ActionOk` and `ActionCancel`.

4.1.5 Parameter Passing with Actions

Sometimes it is helpful to pass arguments together with actions. For example a menu with a list of files to be opened could use the same actions but pass the filename as parameter. For this a special action `ActionParam` with two string parameters is used. The first parameter is the name of the action and the second one is the parameter passed along with the action. Actually using a string as parameter is not a very beautiful solution, but using a typed value would make it impossible to use differently typed parameters in one list. An interaction task can accept this kind of action with given name but arbitrary parameter using `MenuParamAction`.

Example

In the example in Section 4.1.3 the following action was added to a menu for recently opened files:

```
ActionParam "open" (toString id)
```

This action includes the name of the action (`"open"`) and also a parameter which is the identifier of the file encoded as string. If such an action is returned as result of an interaction task (in this case for instance with a `MenuParamAction ("open", Always)` in its action list) all actions with the same name but possibly different parameter values can be handled with the same piece of code. In this case:

```
case action of
    ActionParam "open" fid  = openFile (DBRef (toInt fid))
    ...
```

The parameter can be given to another task. In this example `fid` is converted into a reference to a file and given to a task opening this file.

Summary

In this section a menu system is added to *iTask*. It is an extension of the actions generated by buttons and fits into the task structure of the WDL. The menu's structure clearly has nothing to do with data or process, but it is only set for the entire process in a way that nowhere else in the workflow description it has to be dealt with at least the static part of the structure. The actual menu bar in the GUI is generated based on the context. So non-declarative information is kept minimal in the workflow description. Which actions can be triggered in a certain context can be specified for each interaction task. The way they are extended makes sure that code not using this functionality does not have to be changed.

4.2 Shared Data

Shared Data is data which is shared among different tasks running in parallel. This is very important for programming complex applications. Such applications have a complex internal state which is modified by different parts of the application. The places where the state is modified can be inside the same form or spread among different dialogs, windows or even processes executed by different users. This can be seen as a classical MVC paradigm [17], where the internal state of the application is the *model* containing the entire state of the program and all places where this state is changed are called views.

One characteristic of views is that they can represent the state of the application or only a small part of it in a different way. For example the field for changing the text in a text editor is only used for changing the text stored in the application's state without dealing with the file which is currently opened, probably stored in the same state. Also for instance an IDE might show both the source code of a file and a list of all functions defined inside of it. If one view is changed, for example a function is removed, this change should instantaneously be reflected in all other views.

First in Section 4.2.1 it is explained how destructively updated variables shared among different tasks are realised in a pure functional setting. How the storage used by those variables can be freed again is discussed in Section 4.2.2. Based on this mechanism, which can serve as a shared model, in Section 4.2.3 it is shown how views on such models can be defined. Further new interaction tasks for providing those views to the user are introduced (Section 4.2.4). Some implementation details are discussed, like how the definition of views relate to the usage of data masks (Section 4.2.5) and how shared editors are updated in the *iTask* system (Section 4.2.6). Shared editors can also work on data shared among different processes which might result in editing conflicts. How those are handled is described in Section 4.2.7. Documents are specialised data types which handle uploaded files. Shared documents need special attention because of their structure (Section 4.2.8). Actually how views are defined is a realisation of the theoretical concept of lenses [5] which is discussed in Section 4.2.9.

4.2.1 Reference Types

In a pure functional language it is not possible to define shared variables which can be updated destructively in a straightforward way. For this *reference types* can be used. This solution was already described for *iData* earlier [23] and is also part of the *iTask* system. Therefore it is not a contribution of this thesis.

In *iTask* a reference is a string which gives a unique name to a variable. Additionally it has a type to make sure that it is only used in a correct context. The type used for this is:

```
:: DBid a :== String
```

The actual value of such a variable is stored at a place where the reference type points to, which can be a file, a database or some place in memory. To read and destructively update the value of the variable the following tasks are used:

```
readDB      :: (DBid a)             → Task a | iTask a
writeDB     :: (DBid a) a           → Task a | iTask a
modifyDB    :: (DBid a) (a → a)     → Task a | iTask a
```

The tasks dealing with stored data are also defined in Section A.1.2.

While it makes sense that the unique name has to be provided by the user in situations where the variable is shared among different processes which are started independently, there are many situations in which a variable is created by a process and is only used by itself or other processes started by it. In such a case it is very error prone to let the user determine a unique name, because there could be name conflicts. To not force programmers to implement the generation of unique names, the task `createDBid` for automatically creating unique references is added as contribution of this thesis:

```
createDBid  ::       Task (DBid a)
createDB    :: a → Task (DBid a) | iTask a
```

The implementation of `createDBid` is trivial because each task already has a unique identifier which can be used to make sure that a name generated by a task is also unique. The second task `createDB` is just an abbreviation for first creating a unique identifier and then writing an initial value to it.

Ideally `DBid` should be an abstract type in this case, because such an identifier should never be modified or created without making sure that there is also a store in which a value of the given type is stored. However it is not possible to make the type abstract due to current limitations of the *Clean* compiler[1].

4.2.2 Deleting Data

Memory used by shared variables has to be freed again after they have been used. In contrast to normal variables for variables managed by the *iTask* system garbage collection cannot be done by the *Clean runtime-system* automatically because the values are stored in some other place and exists independently from the references which can be handled by *Clean*'s garbage collection.

More complexity is added by the fact that references can also be stored in other shared variables or be written to disk. Also for reference names which are given by the user it is impossible to determine if somewhere in the future a process uses this name to access a certain variable. Most of the times such names reference a database used by different workflows which should never be garbage collected.

Because of this shared variables are not garbage collected but can only be explicitly deleted by the user. A task for this is added as contribution of this thesis:

[1]A value returned by a task has the context restriction `TC`, which means that it can be packed into a `Dynamic`. Abstract datatypes currently cannot be packed.

```
deleteDB :: (DBid a) → Task Void
```

Of course, this makes it more difficult to deal with shared variables because they can be deleted if trying to read their value. `readDB` gives a default value if one tries to read a deleted variable, but also another task is introduced which only gives a value if the variable is not deleted:

```
readDBIfStored :: (DBid a) → Task (Maybe a) | iTask a
```

4.2.3 Views

With the new tasks described above it is possible to create variables and modify their current values at different places in the workflow. However this is still not as interactive as it should be. It is only possible to get the current value of the shared variable, to edit it and to store the changes back after the user finished the edit task. What we want to achieve is that changes in the editor are immediately mapped back to the shared state and all other editors referring to the same value are updated instantaneously. For this the concept of views is introduced, which can either be an *editor* modifying the value or a *listener* only showing a part of the current state:

```
:: Editor   s a = { editorFrom     :: s → a, editorTo :: a s → s }
:: Listener s a = { listenerFrom   :: s → a }
```

Here `s` is the type of the shared state and `a` is the type of the value shown to the user. The listener has only one function converting the state to a value which is then shown to the user. The editor has a function with the same purpose but additionally one to put the changed value back into the shared state. This function additionally gets the current value of the state to make it possible to update only a small part of it and leave the rest unchanged. This is actually a realisation of lenses [5] and is further discussed in Section 4.2.9.

As will be explained later (Section 4.2.4) one might want to create forms with multiple views on the same state. For this different views have to be put together in one list to give them as argument. The problem is that they are possibly of different type and therefore cannot be put together into one list. To solve this problem functions are used to transform editors and listeners to the abstract type `View`:

```
:: View s
listener    :: (Listener s a) → View s | iTask a & iTask s & SharedVariable s
editor      :: (Editor   s a) → View s | iTask a & iTask s & SharedVariable s

class SharedVariable s | gMerge{|*|} s
```

The trick here is that the type `a` is hidden inside the abstract type `View`. This type has only one type variable indicating the type of the model on which it provides a view. The type of the value edited by the user is completely hidden. In this way it is possible to create lists of different views. The class `SharedVariable` contains a context restriction for a generic function used to merge outdated values (Section 4.2.7). Two special views `idEditor` and `idListener` which do no conversion are predefined:

```
idEditor    :: View s | iTask s & SharedVariable s
idEditor    = editor    {editorFrom = id, editorTo = (λa _ → a)}

idListener  :: View s | iTask s & SharedVariable s
idListener  = listener  {listenerFrom = id}
```

The definition of the type `View` is shortly discussed to illustrate how the view's type can be hidden. Actually users of the *iTask* WDL never have to deal with this:

```
:: View s = ∃a: Listener (Listener‘ s a) | ∃a: Editor (Editor‘ s a)
```

A view is either an editor or a listener. The type of the value edited by or shown to the user is hidden using existential quantification.

The structure of `Listener‘` is simple:

```
:: Listener‘ s a = { visualize :: s → [HtmlTag] }
```

There is only one function transforming a model-value into a *HTML* representation. The type of this representation is independent from the type of the value it represents. Therefore the type of it can completely be hidden.

The same trick is used for editors:

```
:: Editor‘ s a =
    { getNewValue       :: ViewNr [(DataPath,String)] s s     *TSt
                                                            → *(s,*TSt)

    , determineUpdates :: TaskNr ViewNr s [(String,String)] *TSt
                                                            → *(([TUIUpdate],Bool),*TSt)

    , visualize         :: TaskNr ViewNr s                    *TSt
                                                            → *(([TUIDef],Bool),*TSt)
    }
```

There are functions to process events to get a new value, to determine GUI updates and to generate an editor visualisation. They all use the unique type `TSt` which is the core task state used for the implementation of tasks. This state is for instance necessary to retrieve the current value of shared variables.

The functions are not discussed here in detail. The essence is that they are independent form the type presented to the user.

4.2.4 Shared Editors

There are two basic tasks for updating shared values:

```
updateShared        :: q [TaskAction s] (DBid s) [View s] → Task (Action, s)
    | html q & iTask s & SharedVariable s

updateSharedLocal   :: q [TaskAction s] s        [View s] → Task (Action, s)
    | html q & iTask s & SharedVariable s
```

Both look similar to the normal `updateInformationA` but additionally get a list of views. The global version of the task (`updateShared`) gets a reference to a shared variable which is edited by all views given to the task. There is also a local version which is derived from the first one and automatically generates a variable which exists as long as the task is carried out and is shared among the views given to it.

Example 1

A simple example which uses shared editors to instantaneously calculate the sum of two integers is given:

```
quitButton = ButtonAction (ActionQuit, Always)

calculateSum =
    updateSharedLocal "Sum" [quitButton] (0,0) [
        idEditor,
        listener {listenerFrom = λ (x,y) → x + y}
    ]
```

The internal state is a tuple of two integers. The user can edit this state with the identity editor. The second view is a listener which calculates the sum and shows it to the user. The resulting GUI is shown in Figure 4.2.

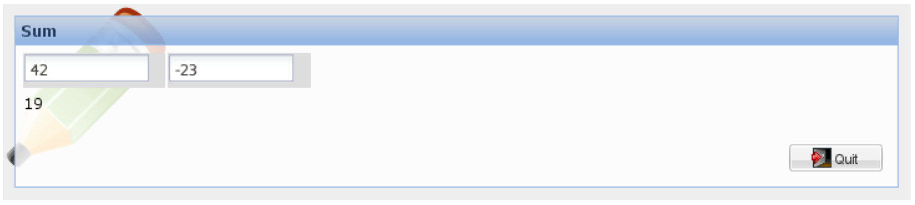


Figure 4.2: The GUI Generated for the Calculate Sum Example

Example 2

Editors can also be used to put certain constraints on an edited value. For example it is possible to let the user edit a list which is sorted automatically each time it is changed by the user:

```
autoSortedList :: Task Void
autoSortedList =
    updateSharedLocal "Automatically Sorted List" [quitButton] emptyL [
        editor {editorFrom = sort, editorTo = λlist _ → list}
    ]
where
    emptyL :: [String]
    emptyL = []
```

The type of the shared model and the view is both [`String`] in this example. The editor behaves as the identity editor with the exception that it sorts the list before showing it to the user using the library function `sort`.

4.2.5 Views & Data Masks

For normal interaction tasks data masks (Section 2.3.2) are used to keep track which fields the user already filled in. For views on shared data the situation is different. The shared variable always has a value and is therefore always converted into a view value with all fields filled in. For instance if one has a shared view on a list of integers and adds an element automatically the default value 0 is filled in instead of a blank field.

The fundamental problem is that the functions defining views can only transform a value and no mask. It would be possible to extend the concept of views such that also data masks are converted to solve this problem. However this makes using views much more difficult and inconvenient. Because in most cases this feature is not needed anyhow we decided to accept that shared views always have values filled in for all fields and leave the concept of views being defined only in terms of transforming a value.

4.2.6 Updating Shared Editors

Shared editors are more difficult to handle than normal ones. The reason is that the local state of normal editor is only changed by themselves. In contrast the state shared editors work on can also be changed by other tasks.

In this section it is explained how the *iTask* system is extend to deal with the problem of updating multiple editors working on the same shared value at the same time.

Generating GUI Updates for Shared Data

For normal interaction tasks an interface definition or a set of updates is generated during the construction of the task tree and stored inside of it. This is possible because all tasks have a local value which is edited only by them and cannot be changed by other subtasks.

This is not possible for views because at the moment a node for the task in the task tree is created there are possibly other views which were not added to the tree yet and will change the value of the shared state. So the calculation of the interface definition or updates has to be done after the entire tree is built and all editors updated their values.

To achieve this the task node of an interactive task can also contain a function calculating an interface definition or updates for it at a moment after the entire tree is built:

```
:: InteractiveTask  = Definition   ([TUIDef],[TUIButton])  [(Action,Bool)]
                    | Updates      [TUIUpdate]             [(Action,Bool)]
                    | Message      ([TUIDef],[TUIButton])  [(Action,Bool)]
                    | Func         (*TSt → *(InteractiveTask, *TSt))
```

So an interactive node in the task tree can include a GUI definition, GUI updates or a message for the conventional interaction tasks. Additionally a list of accepted actions together with a flag indicating if the action can currently be triggered is included. For shared editors a function is used to calculate a new node after the entire tree has been built. In this way all views are rendered with the same, current value of the shared variable.

Updating Basic Values

Events from the client can either change a basic value or a constructor. Basic values are for instance `Int`, `String` or `Bool`, but also specialised types like `Note` or `Date` are treated like basic types in this context. An important property of such events is that it is not necessary to update the user interface on the client side as response to the event. This is because the form element representing the basic value already contains the new value the user inserted into it. The situation is different for events changing the constructor for a value inside the datastructure. In this case the GUI on the client side does not represent the new value at the moment the event is sent. Form elements possibly have to be removed and new elements have to be added.

Consequently for forms editing a local value the server never sends updates for changing fields representing basic values. The situation is different for shared editors, because basic values can possibly be changed by other views or the function in the view definition. So the visualisation function has to be extended to send also updates for basic values which have been changed.

However just comparing the old and the new value of a basic type and sending an update if it has changed gives a problem for some cases. This is illustrated by the following example: assume there is a single field for entering an integer, but a function in the view always changes it back to 0 before storing it back in the shared state:

```
constIntExample = updateSharedLocal
                    "constant integer"
                    [quitButton]
                    0
                    [editor {editorFrom = id, editorTo = λ_ _ → 0}]
```

The correct behaviour of the server would be to always set the field back to 0. The problem with just comparing the new value with the previous one when generating updates is that the value of the integer is always 0. So no update is send to the client although the user may have typed in a different integer.

In the following some attempts for solving this problem are discussed.

A Naive Approach: Updating all Basic Values A simple solution for this problem seems to always send updates to set all basic value fields to their correct value. The first obvious drawback is that superfluous update commands are sent, which wastes bandwidth and computation time. But worst of all, the user may already work on a different field at the moment the update from the server arrives. In this case the information the user already typed in is overwritten with the old value from the server. Summarised this solution destroys the concept of updating the user interface in an iterative way. So this approach is not a solution.

Solution 1: Redefining the Meaning of Old Values I suggested this approach for solving the problem in some early work about iteratively updated web-forms [20].

The generic function for generating updates expects two values, which is the old value before the event and the new values updated by the event (Section 2.3.5). In this solution the meaning of the old value is redefined. Instead of using the value before the event, the value the GUI on the client-side represents at the moment the event is sent is taken as old value. For instance if an integer field is changed from 0 to 1, the old value according to the original definition is 0. Actually the GUI already represents a 1 at the moment the event is sent.

To retrieve the old value according to the new definition first only events updating basic values are processes. In this way a value representing the state on the client-side is constructed. The new value is this value with additionally also the events updating constructors applied.

This solves the problem, because fields representing a value differing from the new one are updated.

However giving the old value a different meaning causes problems in some cases. An example is a new implementation of the visualisation of the list type where automatically a new optional blank element is added to the end of the list. With the redefined meaning of old value the visualisation function cannot derive if a new value was added as last element and a new blank field has to be added or if the last element was changed.

So a solution which uses the original definition of the old value is needed.

Solution 2: Resetting Fields Actually the server can detect when a problem as described above occurs. This happens when a basic value's old and new value are equal but there has been an event for it, which means that it was changed by the user. In this case the value shown in the client is not correct and must be reset to the previous value. The server can achieve this by sending an update instruction for setting the value back to the old one.

This solution solves the problem by detecting the situation causing the problem, but leaves the definition of the old value intact and therefore causes no other problems. Consequently this solution has finally been implemented for the *iTask* system.

Overview

How views are updated is summarised in Figure 4.3. The client sends a list of events to the server which all originate from the same editor. First the previous value stored on the server is updated using those events after conversion to the editor domain. This new value is used to determine updates for the editor by comparing it with the old value. Additionally the new value after conversion to the model domain is stored as new value of the shared variable. In the phase of building the user interface also this new value is used for determining the new user interfaces of all other editors.

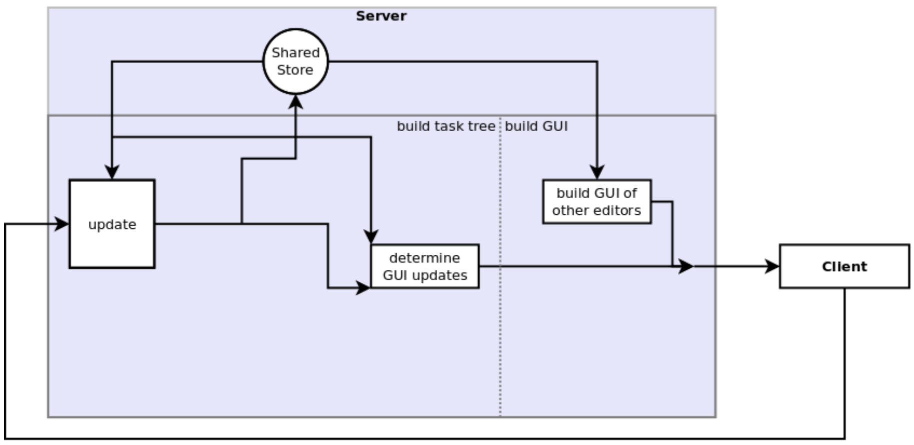


Figure 4.3: An Overview how Shared Editors are Updated

4.2.7 Merging Values

If all editors providing a view on the same shared variable are inside the same process, all of them are updated for each update request to the server. Consequently they all have a value representing the same shared state. But if shared values are edited in different processes possibly by different users, it can happen that an editor representing an outdated value is sending an update request. There are two easy solutions to deal with this situation:

- Discard the changes of the outdated editor.
- Overwrite all changes the outdated editor missed.

Obviously the second approach is not very practical. First of all, an arbitrary number of updates can be overwritten, possibly the outdated editor missed many of them. Also if a user changes something and sees her change happen on the screen, she expects the change to be successful and does not expect it to be overwritten later. With the first approach only one update is discarded and the user immediately gets feedback about this because the new value the editor is updated to does not represent her changes.

So in principle the first solution is preferable. This was also done in previous work about *iData* [23] where a version number was used and only editors with the correct number were allowed to change the shared variable. But there are cases were it might be safe to do (part of) the update even if the editor was outdated. While the entire shared value might be outdated, it can be safe to change parts of it which were not changed by the missed updates. For this each editor locally saves the last value of the shared variable it has seen and a function (wrapping the generic function `gMerge`) to merge the different values is used:

```
mergeValues :: a a a → a | gMerge{|*|} a
```

There are three arguments of the same type:

old The last value of the shared variable the editor has seen before the update.

current The current, up-to-date value of the shared variable.

new A new value resulting from the user's edit action.

Intuitively what has to be done is that *current* is updated to *new* if no changes between *old* and *current* are overwritten. In the case that the editor is up-to-date *old* and *current* are equal and the result is the value of *new*.

The generic algorithm for merging the values has to make a decision each time there is a difference in the structure or in the value of basic types of the three values:

- If there is no difference between *old* and *current* it is safe to construct a result according to *new*.
- If there is a difference between *old* and *current* this difference should not be overwritten and the structure or value of the result is given by *current*.

This algorithm works fine for changed basic values and if constructors are used to indicate choices. For datastructures which can change their size dynamically such as lists it can have unexpected results. This is because the algorithm cannot make a distinction between cases in which the value of items is changed and cases in which the position changes because other items were added or removed. So a better solution might be needed here. Also for notes one can imagine that there are better ways to merge different versions than treating the entire string as one value.

A situation were two users change the same part of a data structure is undesirable in many cases, because it always leads to problems. So most applications should make sure that not more than one user is allowed to modify the same part of a data structure at the same time on a higher level.

On the other hand there are applications which let users work on the same data at the same time. Examples for this are *Google Wave*[2] and *Etherpad*[3]. For realising this kind of realtime interaction the *iTask* system would have to be extended with the possibility to send notification to clients and not only to answer to requests. For this first *Clean* would have to provide the possibility to start multiple threads. So this remains an interesting issue for future work.

Example

The value of a local variable is [1,2] at the moment an editor is generated. The user changes this value to [3,4]. Between the generation of the form and the update sent to the server another process changes the value to [0,2]. The new value of the state is determined by `mergeValues` [1,2] [0,2] [3,4]. The first element of the list has been changed from 1 to 0. This change is not overwritten and therefore the first element of the result is 0. For the second element the situation is different because there is no difference between the first two arguments. Consequently the second element of the result is 4. The entire result is [0,4].

4.2.8 Shared Documents

Documents are used to represent any kind of file in the *iTask* system. The user can upload, download and delete them. Documents are handled in a special way inside *iTask*. This is because files of arbitrary size can be uploaded and possibly a large amount of data is involved. Therefore documents are stored in such a way that not the entire data has to be copied too often. For this references to the actual data

[2]`http://wave.google.com/`
[3]`http://www.etherpad.org/`

are used. This requires special attention if documents are shared among a number of views.

There are two approaches to handle documents. In a first approach documents are *mutable*. This means that old data is overwritten if the user uploads new one. In the end it turned out that *immutable documents* are easier to handle. Here each time the user uploads data a unique identifier is given to it and it is never changed later. This avoids many problems when sharing documents. Still it is an interesting issue how mutable shared documents can be handled and is therefore also discussed in this section. After this it is discussed how immutable documents avoids many problems which have to be solved for mutable ones in a complicated way.

Mutable Shared Documents

First it is described how mutable documents are handled by normal editors. This is followed by a discussion how documents edited by shared editors are handled. Finally some remarks about deleting documents are made.

Local Documents As already mentioned documents are stored in a special way. The document type does not directly contain the data of the document but a store, which contains the data, is created at the moment document data is uploaded. The document type itself only contains a reference to this store which is identified by the task by which it was created. To avoid copying the data unnecessarily tasks can use document data which was not created by themselves, but by previous tasks. This is safe since this previous task is finished at the moment data created by it is used by another task and can therefore not change the data any more. If the document is changed, which means that new data is uploaded, a new store for the current task is created. Tasks are never allowed to overwrite data created by previous tasks because they can possibly be used by other parallel tasks.

The actual data is only send to the user if this is requested. If generating a form for a document only a button for requesting the download is generated. For downloads and uploads of documents a special service (added to the services discussed in Section 2.2.2) is made available to the client.

An example is given in Figure 4.4. First a document is created by *Task 1*. Its data is stored in a store identified by this task. Then two new tasks are started which both give the user the possibility to update the document. Although the two documents are conceptually different documents which can be changed independently the data does not have to be copied. Only the document information containing a reference to the data is copied and given to the two parallel tasks. Both still use the data created by *Task* 1. If there is no change this is no problem, because *Task 1* is finished and the data will never be changed. In the right task the user does not update the document and therefore the returned document still points to the data created by *Task 1*. In the left task the user uploads new data. At the moment the new data is uploaded a new store for it is created. So the returned document contains a reference to the new data without interfering with the results of the right task.

The values given by the tasks do not have to be single documents. They can also be datastructures possibly containing an unbounded number of them. For dealing with this each document also has an index to make it possible to create more than one data store for a single task.

Shared Documents The data of documents stored inside a shared variable do not belong to a task but to this shared variable. This makes sure that each task accessing it using an `updateShared` has access to the current version which might be data uploaded by a different task. To handle this for this thesis the types

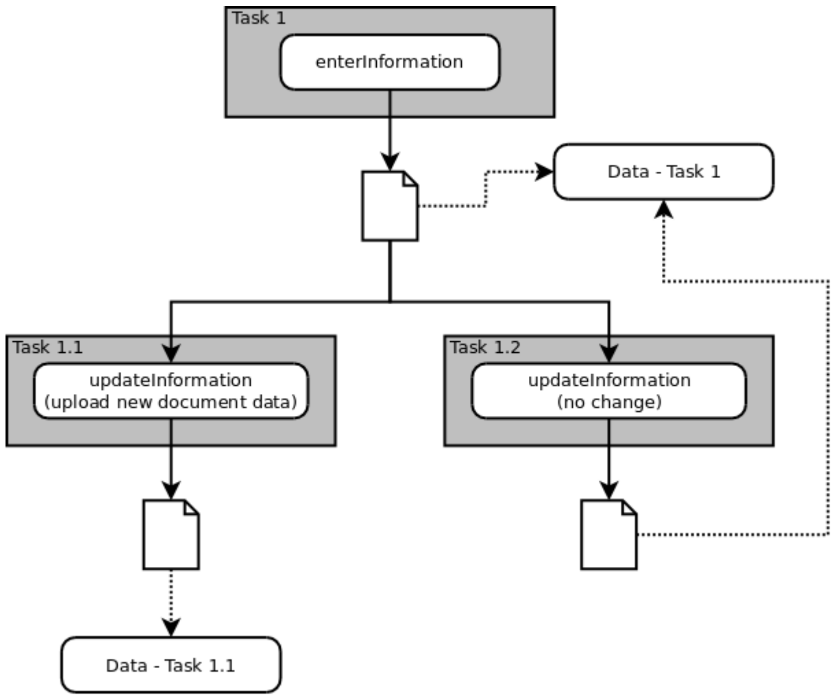


Figure 4.4: An Example of How a Local Document is Handled

representing documents are extended to make a distinction between the two kinds of documents:

```
:: DocumentType = Local | Shared String
```

So a document can either be local or shared indicated by a value of type `DocumentType`. For local documents it is not necessary to store the task identifier they belong to because the identifier of the current task is always available. For shared documents the identifier of the shared variable has to be stored because it is not available at the moment a document upload is handled. It is stored as string because of type problems occurring otherwise[4].

There are situations in which a shared document can safely use local data. This is possible if a local document already containing data is used by a shared editor and therefore turned into a shared one. Because of this each document can be local or shared and independently has a local or shared location where its data is stored. This is modelled by the following types:

```
:: DocumentDataLocation = LocalLocation  TaskId
                        | SharedLocation String SharedDocumentVersion
:: SharedDocumentVersion :== Int
```

For the data location either the identifier of a task or of a shared variable is used to point to a store with the document's data. Additionally for shared documents a version number is used to make sure that outdated editors never overwrite changes they did not see.

Finally the new datatype for mutable documents is:

```
:: Document        =  { type          :: DocumentType
```

[4]The shared variable in which the document is embedded can be of arbitrary type. So the type should actually be `:: DocumentType = Local | Shared (∀a: (DBid a))`. For quantified types generic functions cannot be derived automatically. Using a string is more convenient than doing this for all generic functions needed for this type.

```
                         , content      :: DocumentContent
                         }

:: DocumentContent  =    EmptyDocument | DocumentContent DocumentInfo

:: DocumentInfo     =    { fileName     :: String
                         , size         :: Int
                         , mimeType     :: String
                         , dataLocation :: DocumentDataLocation
                         , index        :: Int
                         }
```

Each document always has a type (`DocumentType`). If the document is empty no further information is required (`EmptyDocument`). If it is not empty (`DocumentContent`) it has a name, a size, a mime-type, a data location and an index, which is used to handle datastructures containing more than one document, as already discussed.

Empty documents are created as local document. Even if they are stored inside a shared variable they are still local until they are used in the context of an `updateShared`. This is done because in this way no efficiency is lost if a value is stored in a database but no shared editors are ever using this document. In the case that a shared editor is used on a variable all documents inside the datastructure are changed to shared documents. For this the following generic function is used:

```
generic gMakeSharedCopy a :: a (DBid b) → a
```

The first argument is the value in which all documents are changed and the second one is the identifier of the shared variable. The function only changes the type but not the data location of the documents.

In the result of an `updateShared` or of a `readDB` all documents are changed back to local copies by this function:

```
generic gMakeLocalCopy a :: a *TSt → (a,*TSt)
```

If the document still has a local data location only the type is changed back, as shown in Figure 4.5. First a local document is created which is then stored in a shared variable by `createDB`. The document inside of the shared variable is still a local one. The identifier of the shared variable (*DBid 1*) is given to an `updateShared` task. Here the document is changed to a shared one. The number identifies the shared variable it belongs to. Still the document points to the original data created by *Task 1*. The user does not upload new data. If the `updateShared` task finishes a local document is returned, which still points to the original data. So the data is never copied.

If new document data is uploaded for a shared document it is stored at a shared location. Data at a local location is only modified by the task owning it. If the task is finished the data stays the same. This is because possibly other parallel tasks also use the same data. This situation is depicted in Figure 4.6. A local document is given to two parallel tasks. In the first one its content is shown to the user. In the second one first it is stored inside a shared variable and given to an `updateShared` after that. Before new data is uploaded in the shared editor both documents still point to the same data. If new data is uploaded a new store has to be created because in the left task still the old content of the document has to be available. So after the upload the new value of the shared variable (*shared 1"*) has a document pointing to a new data store belonging to the shared variable.

Finally it has to be discussed what happens if a shared document with its data at a shared location is turned into a local copy. In this case a copy of the data has to be made at a local location. This is the reason why `gMakeLocalCopy` modifies the task state. This new copy is needed because one wants to have the document at the

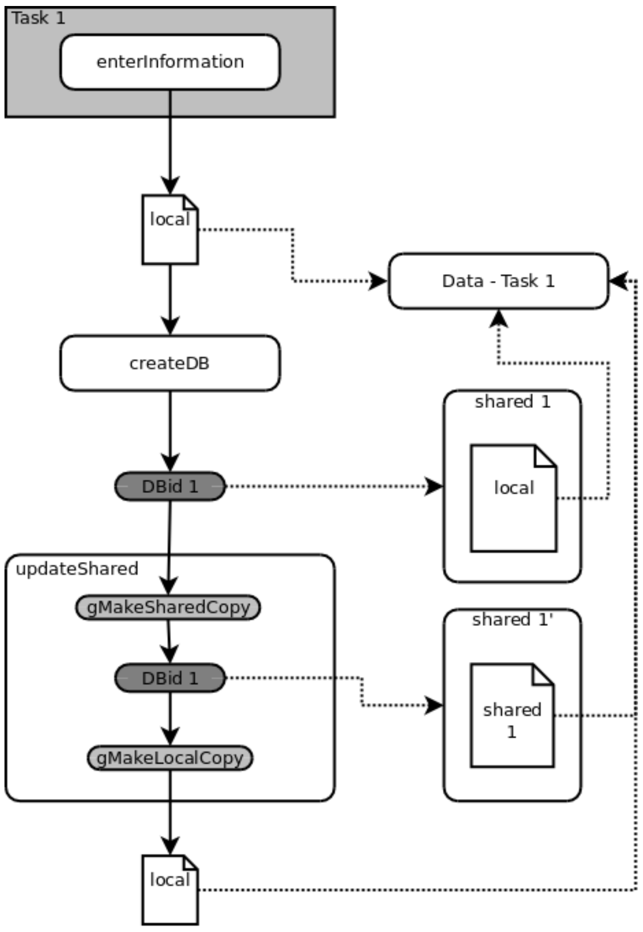


Figure 4.5: An Example of How a Shared Document Without New Data is Handled

moment of time when the document is retrieved, either because an `updateShared` task finished and returned it or because `readDB` is called. If the value inside the shared variable is changed later this should have no effect on the local copy.

Deleting Document Data For local documents it is not necessary to delete the data store if a document is deleted. The only thing that has to be done is to do not point at this data any more. Of course, it would be nice if also in this case the document data is freed up.

Mutable shared documents explicitly have to be deleted because other views accessing the document also have to notice that the document to which they still have a pointer was deleted. So a new method to explicitly delete documents is added to the document service.

Also document data for which no references are existing any more should be deleted by some kind of garbage collection system, which is not implemented yet.

Immutable Shared Documents

One way to drastically simplify handling of documents is to make them immutable. This means that each time a document is uploaded a unique identifier is determined for it and that its data never changes. If a new file is uploaded the document is replaced by a new one without erasing the data of the previous one.

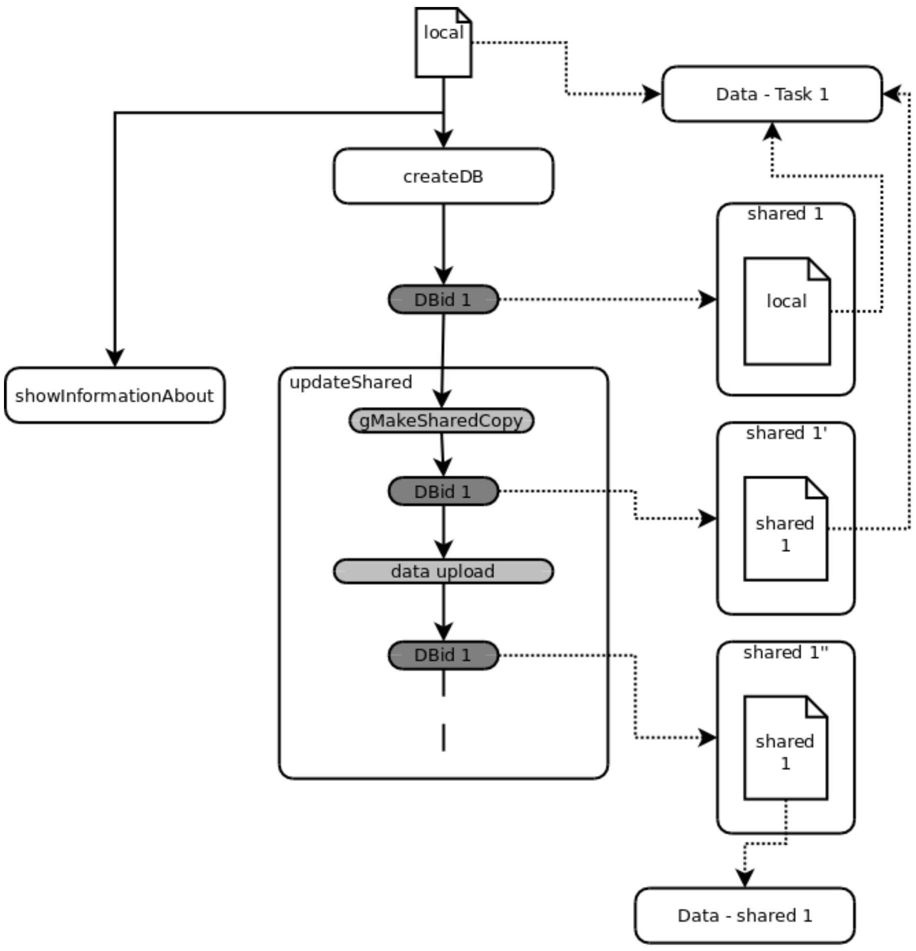


Figure 4.6: An Example of How a Shared Document With New Uploaded Data is Handled

This solves all problems of documents used by parallel tasks and possibly being modified and makes the distinction between shared and local document superfluous. The drawback is that each version of the document keeps stored, as long as there is no garbage collection.

Compared to the previously defined type for mutable documents the structure of documents becomes very simple now:

```
:: Document = { documentId  :: DocumentId
              , name        :: String
              , mime        :: String
              , size        :: Int
              }

:: DocumentId :== String
```

Each document has a unique identifier, a name, a mime type and a size.

However this simple solution behaves not the same way as the solution discussed for mutable documents. If a document is downloaded from an outdated view the user does not get the most recent version of the document, but the version that was recent the last time the view was refreshed. If an outdated editor tries to update a new document or delete it, the action has no effect and the value is changed back to the recent version by the merging algorithm (Section 4.2.7).

Because downloading the file cannot be seen as an editing action, immutable documents behave in the same way as basic values or other specialised types. To see the recent value of a basic value in an outdated view, the user either has to press the refresh button or try to change the value, which results in the field being updated to the recent value. The problem therefore exists on a higher level and is not specific to documents.

4.2.9 Lenses

To show that the concept of views as introduced here is general enough to make all kinds of state updates possible, it is compared with the concept of lenses [5].

Definition of Lenses

A lens is a pair of functions:

$$v \nearrow \in \Sigma \rightarrow \Delta \tag{4.1}$$

$$v \searrow \in \Delta \times \Sigma \rightarrow \Sigma \tag{4.2}$$

The first function $v \nearrow$, also called "get", corresponds with `editorFrom` and $v \searrow$, also called "putback", corresponds with `editorTo`. So the editors defined in this thesis are as expressive as lenses.

Totality

It is important that for each possible value of the shared state an editor value exists and that for each possible editor value in combination with a shared value a new shared value can be computed. In short $v \nearrow$ and $v \searrow$ should be total.

In *Clean* it is possible to define partial functions. In the case a function is evaluated for an argument for which it is undefined, the entire program crashes. The compiler can give warnings about functions which can possibly fail, but it is the programmers responsibility to provide functions to the editor that do not let the program crash.

Actually it is not necessary to be that strict, because there might be values of the state domain which are never given to the variable, given a certain start value and a list of editors. But nevertheless we would regard it as good practise to define only total editors, such that they can safely be combined which other editors working on the same state type.

Well-behaved Lenses

Well-behaved lenses have the following property:

$$v \searrow (v \nearrow (I), I) = I \qquad \text{for all } I \in \Sigma \tag{4.3}$$

$$v \nearrow (v \searrow (J, I)) = J \qquad \text{for all } (J, I) \in \Delta \times \Sigma \tag{4.4}$$

Editors not fulfilling this can show unexpected behaviour. For example if an editor value is changed one might expect that while all other editors might be updated the value of the changed editor remains unchanged. However not fulfilling this property makes it possible to define interesting editors showing values with enforced properties, such as automatically sorted lists (see also Example 2 in Section 4.2.4). Therefore this property is not required for *iTask* views.

Summary

In this section it is shown how views on shared data can be defined as realisation of the theoretical concept of lenses [5]. Further interaction tasks presenting those views to the user are defined and it is shown how the *iTask* system is extended for handling those shared editors. Special attention is paid to documents which in contrast to other types include references to the actual document's data instead of the data itself. Finally a simple way to deal with possible editing conflicts is shown. Making it possible that multiple users can work on the same data in realtime remains an issue for future work.

4.3 Grouped Tasks

More complex applications consist of multiple windows with which the user can perform different tasks, for instance editing different documents. Most applications at least have dialogs to do basic operations such as opening files. This can be modelled as different tasks running in parallel. In this section it is discussed how tasks can be grouped together in such as way that they are usable to model different windows or dialogs.

In *iTask* there is a distinction between parallel tasks and groups of tasks. Parallel tasks are conceptually meant as independent tasks which are carried out at the same time. Each of such task runs in its own process and can be assigned to different users. The manager of such a parallel task can monitor the progress and reassign tasks. In contrast groups are tasks which belong together and run in a single process. They are a good way to model different dialogs and windows running inside the same application.

Because parallel tasks are not needed for programming (single-user) GUI applications, only groups are discussed in this thesis. Both concepts are based on the same concept for dynamically adding new tasks, discussed in Section 4.3.1. Based on this the contribution of thesis is to extend the concept of groups with some concepts needed for realising GUI applications. First, tasks can get different behaviours such that they are rendered in windows or modal dialogs (Section 4.3.2). Second, tasks inside of groups can be focussed by other ones (Section 4.3.3). Third, actions can be attached to groups to make it possible to add new tasks independent from other ones triggered by menu events (Section 4.3.4). The core group combinators eventually added to the WDL and supporting all of those concepts is discussed in Section 4.3.5. Because the core combinator is difficult to handle some versions derived from it are introduced in Section 4.3.6. Finally the behaviour of nested groups is described in Section 4.3.7.

4.3.1 Dynamically Adding Tasks

A basic requirement for a combinator for grouping tasks is that tasks can be added dynamically. For example dialogs are always added dynamically if the user wants to perform a certain action. In the following a first approach for dynamically adding tasks is discussed followed by an improved approach finally added to the *iTask* system.

The Todo-list Combinator

In some earlier work [21] I proposed the so-called *Todo-list Combinator* which gives the intuition that there are a number of tasks which have to be performed and can possibly generate additional tasks. The combinator task finishes if all required tasks are performed:

```
todoList ::
        (taskResult gState → (gState,[Task taskResult]))
        gState
        [Task taskResult]
    →
        Task gState
    | iTask taskResult & iTask gState
```

The third parameter is a list of initial tasks which are executed in parallel. A group has an internal state of type `gState` which's initial value is given as second parameter. This state is returned when all tasks are finished. A function given as first parameter is used to update the group's state when a task finishes. For this the task's result and the old value of the state is used. The function also produces a second value of type `[Task taskResult]` which is a list of new tasks. Those tasks are dynamically added to the list of parallel tasks.

Intuitively this combinator represents a todo-list of tasks. Finishing a task can result in other tasks added to the list. If all tasks are finished the result accumulated in the state is given as result. The original problem solved by this combinator is the definition of a function with possible undefined variables. Each undefined variable results in another task for giving the definition of this variable. The combinator stops if there are no undefined variables left. The accumulated result is a list of variable names together with a definition.

More General Actions

A major drawback of the Todo-list combinator is that it is not possible to stop the execution of all tasks if not all of them are finished. This is solved by giving the possibility to return a `Stop` instead of a list of new tasks. So the function returns a value of the following type instead of a list of tasks:

```
:: PAction x = Stop | Continue | Extend .[x]
```

`Continue` is just a more readable way to write `Extend []`.

This idea is used in the *iTask* system. A simplified version (which is extended with other concepts in the remainder of this section) of the resulting group combinator is:

```
groupSimplified ::
    (taskResult gState → (gState,PAction (Task taskResult)))
    gState
    [Task taskResult]
  →
    Task gState
  | iTask taskResult & iTask gState
```

The only difference with the Todo-list combinator is that the update function generates a value of type `PAction (Task taskResult)` instead of `[Task taskResult]`.

4.3.2 Grouped Behaviour

One goal of groups is to model windows, so there has to be a way to specify whether a given task is a window, which can be modal or not, or results in a non-floating task in the GUI. The *tuning combinators* (Section A.2.4) are used to assign one of the following behaviours to a grouped task:

```
:: GroupedBehaviour = GBFixed
                    | GBFloating
                    | GBAlwaysFixed
                    | GBAlwaysFloating
                    | GBModal
```

First of all tasks can initially be fixed (`GBFixed`) or floating which means visualised as window (`GBFloating`). The user has the possibility to undock fixed tasks to make them floating or to dock windows to make them fixed. To make tasks always fixed or floating the behaviour can be set to `GBAlwaysFixed` or `GBAlwaysFloating`. Finally a task can be a modal dialog (`GBModal`) which means that interaction with all other tasks inside the group is blocked as long as this task is running. If no behaviour is specified tasks have the default behaviour `GBFixed`.

The server just sends the behaviour along with the normal GUI definition. To render the tasks accordingly is the responsibility of the client. This conforms to the design goal that the GUI definition is as abstract as possible. There are several possibilities how to render non-floating tasks. A good and simple choice is to arrange them vertically down but in a way that there is a clear separation between them to show that they are different tasks. An idea was to save space by making a shared menu bar and a footer where buttons are shown. For this always one task has the focus, which of course should be indicated visually. Only the menu bar and the buttons of the focused task are shown.

While this is a good idea for menus, in practise it turned out to be not very user-friendly to have a shared button footer. One reason is that this is not the expected behaviour and therefore users may not even notice that there are buttons at the bottom of the screen. Another reason is that the distance between the buttons and the form can become large and therefore it is inconvenient to use them. So buttons are rendered inside the form as usual, but only the buttons of the focused task are visible while buttons of the other ones are hidden. This also saves the same amount of space than the first solution but is more convenient to use, although it can be discussed whether hidden buttons may confuse the user.

Because the details of the user interface are handled by the client, information like positions and sizes of windows and whether a task has been pinned or unpinned are not synchronized with the server. One might want to do that to make sure that the user interface is in the same state if the user stops working and continues later. But synchronizing this information with the server is difficult because different clients may render the user interface in a very different way. So the standard client stores the state of the user interface locally. This is done by storing it inside of cookies. Consequently as long as the user continues working on the same machine the user interface is restored. Continuing the work on a different machine results in different positions and sizes of windows and the initial pinned states.

Example

In Figure 4.7 an example of a group with four tasks showing all possible grouped behaviours except `GBModal` is given. For the initially fixed task there is a button to unpin it making it floating, but not for the task which is always fixed. Analogously only one of the windows has a button to pin it. The button of the first fixed task is hidden since it does not have the focus.

4.3.3 Focussing Tasks

In most user interface systems the programmer has full control over the stacking order of windows, which determines which of them overlaps or hides other ones. Also the programmer can freely choose their position, size and other properties. The *iTask* system is designed in such a way that as much as possible details of the concrete user interface are left over to the client. This is also true for grouped tasks where the server only controls which tasks there are and which behaviour they

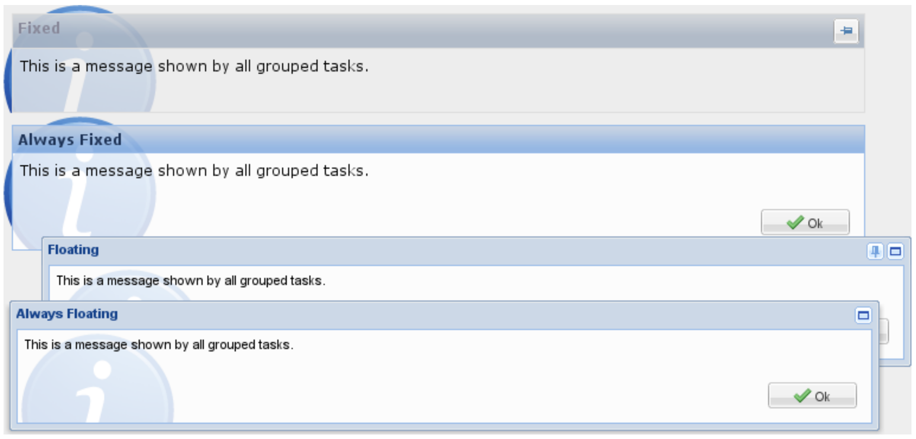

Fixed
This is a message shown by all grouped tasks.
Always Fixed
This is a message shown by all grouped tasks.
Ok
Floating
This is a message shown by all grouped tasks.
Always Floating
This is a message shown by all grouped tasks.
Ok

Figure 4.7: Examples of How the Different Grouped Behaviours are Visualised

have. All details, like all properties of the windows currently shown to the user, are controlled by the client.

However focussing windows is an essential action needed for many applications. This is especially important for applications with many windows. Because of this the actions of a group are extended with a special command focussing tasks with a given tag:

```
:: PAction x t = Stop | Continue | Extend .[x] | Focus (Tag t)
```

Tags can be given to tasks using the tuning combinators and are defined as:

```
:: Tag s = Tag s & toString s
```

In *Clean* this notation imposes a context restriction on the argument of the constructor `Tag`. This means that values of all types for which an instance of `toString` is defined can serve as tags. Internally tags are handled as strings, but defining tags in this way makes it more convenient to use for instance integers as tags. Tags can be assigned to tasks using the tuning combinators. A task can have more than one tag. Giving more than one tag to a task can be achieved using the tuning combinator several times or using another type which represents a list of tags:

```
:: Tags s = Tags [s] & toString s
```

In contrast to group behaviours which have to be given to the top task which is child of the group, tags for focussing an interaction task shown in a group have to be assigned to that task directly. So if a grouped task is a sequence with several interaction tasks, each of them can have different tags.

If the command to focus tasks with a certain tag is given it can happen that there is more than one task with this tag. If this situation occurs for a number of fixed tasks, only one of them is focussed. The choice which task to focus is left to the client. For floating tasks the windows of all tasks with the tag are brought to front, above the windows of other tasks. The order of the focussed windows also depends on the client.

Example

The following example illustrates how one task can focus another one inside of a group:

```
focusExample = groupSimplified func Void [task i \\ i ← [1..4]]
where
```

```
task :: Int → Task Int
task i = enterInformationAbout "Task" i <<@ Tag i <<@ GBFloating

func :: Int Void → (Void,PAction (Task Int) Int)
func i _ = (Void, Focus (Tag i))
```

The group does not make use of the state which is therefore of type `Void`. Initially it consists of four floating tasks tagged with 1 – 4. The user can enter an integer in all of those tasks. In Figure 4.8 the user enters a 1 in the task tagged with 4.

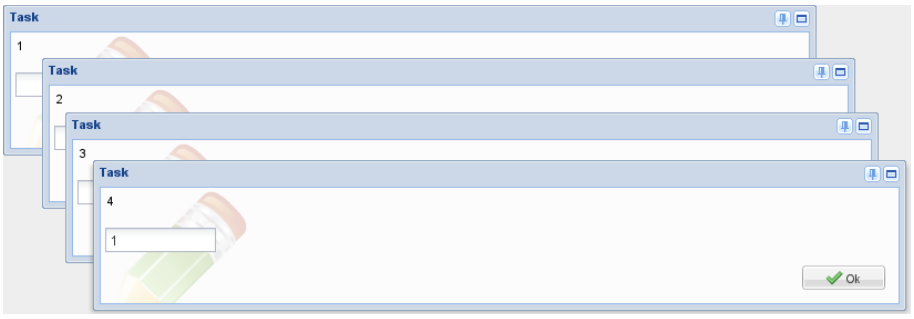


Figure 4.8: The Focus Example Before a Task Focused Another One

If the user now presses the okay button the task returns 1 which is then given to the update function. The function generates a focus command for all tasks tagged with 1. So, as shown in Figure 4.9, the window of the task tagged with 1 is brought to front and focussed.

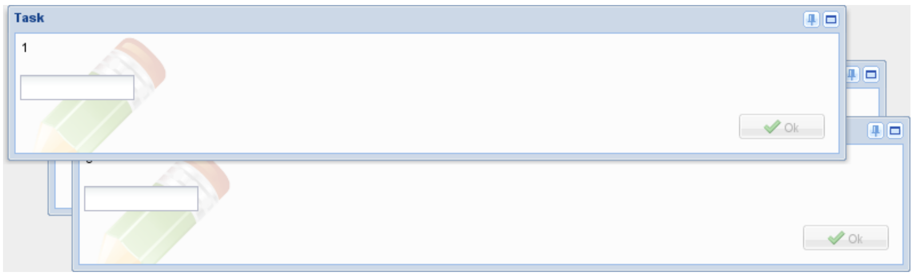


Figure 4.9: The Focus Example After a Task Focused Another One

4.3.4 Group Actions

One of the observations which led to groups of tasks which can be extended dynamically, was that there are situations in which the result of finished tasks may make new tasks necessary. Those tasks should be added without interfering with other tasks still running in the group.

But there are situations in which this is not sufficient. The user might want to add new tasks without stopping one of the tasks currently running inside the group. For example the user might want to add a task editing a file, without interfering with tasks editing other files running in the same group. There might even be groups without any running task at all, for instance a MDI application without any opened document. In this case the user should be able to open a file in some way. The task for doing this cannot be started by a task inside the group, because there is no such task. The task has to be started by the group itself.

For this purpose *group actions* are invented. They are a way to produce a value which is treated as if it was given as result of a finished task and can therefore have the same effect on the group's state. They can add new tasks, stop the group or focus tasks. Also conceptually it is nice to distinguish between actions independent from a certain task in the group, like opening a new file, and actions concerning a certain task, like doing some modification of a file edited by a task inside a group.

Group actions can only be triggered by menus. The reason is that menus can be added to each kind of user interface. If a task already has its own menu it is easy to merge the group actions into it. Also for groups without any task a menu bar can be added at the top. If a task is rendered inside of a window, menu items referring to group actions can be hidden because they are shown in the top menu of the group anyhow. This gives a nice separation between global actions and actions only related to that task. If a floating task's menu only includes group action related items it can be hidden completely. For this the server sends a flag for each menu item, indicating whether it refers to a group action or not. If the task is pinned its menu, then shown in the group's toolbar, includes all items.

A list of group actions is added as extra parameter to the group combinator:

```
groupSimplifiedA ::
    (taskResult gState → (gState,PAction (Task taskResult)))
    gState
    [Task taskResult]
    [GroupAction taskResult gState shared]
  →
    Task gState
  | iTask taskResult & iTask gState & iTask shared
```

A group action is defined as:

```
:: GroupAction taskResult gState shared =
      GroupAction      Action taskResult            (GroupCondition gState shared)
    | GroupActionParam String (String → taskResult) (GroupCondition gState shared)
```

Like for menu actions accepted by interaction tasks (Section 4.1.4) there is a distinction between normal actions and parametrised ones. In the first case if a given action is triggered by the menu a value of type `taskResult` is generated as if it was the result of a task inside the group. In the second case the parameter is used to produce this value. For both cases there is a condition telling whether the action is possible at the moment or not:

```
:: GroupCondition gState shared =
      GroupAlways
    | StatePredicate  (gState → Bool)
    | SharedPredicate (DBid shared) ((SharedValue shared) → Bool)

:: SharedValue shared =
      SharedDeleted
    | SharedValue shared
```

In the first case (`GroupAlways`) the action is always possible. Then a predicate on the group's state can be used (`StatePredicate`). Often the condition may depend on a shared variable which is used by tasks running inside the group. So also a predicate on an arbitrary shared variable (`SharedPredicate`) can be used. For this a reference to the variable has to be given together with a predicate. The reference to the shared variable has to be dereferenced before a predicate on the state can be used. Each time the predicate is evaluated first it is determined if the shared variable is deleted or not. A value of type `SharedValue` is produced accordingly and given to the predicate.

It can be prevented that tasks inherit the actions from its group. If group actions are used is determined by this type which can be assigned to a task using the tuning combinators:

```
:: GroupActionsBehaviour    = IncludeGroupActions
                            | ExcludeGroupActions
```

By default all tasks inherit group actions from their parent group.

Example

This example shows how group action can be used to dynamically add tasks. Also it shows how grouped tasks are handled by the menu system:

```
groupActionsExample =
        setMenus    [Menu "Example" [ MenuItem "Add task"   ActionAdd   Nothing
                                    , MenuItem "Close"      ActionClose Nothing
                                    ]
                    ]
    >>| groupSimplifiedA func Void [] groupActions
where
    ActionAdd = ActionLabel "add"

    func :: Bool Void → (Void,PAction (Task Bool) tag)
    func True  _ = (Void, Extend [  showMessageA
                                        "Dynamically added task!"
                                        [MenuAction (ActionClose, Always)]
                                >>| return False])
    func False _ = (Void, Continue)

    groupActions :: [GroupAction Bool Void Void]
    groupActions = [GroupAction ActionAdd True GroupAlways]
```

The group's state is not used and is therefore of type `Void`. The initially empty group has a menu with one item for dynamically adding an arbitrary number of simple tasks to it, as depicted in Figure 4.10. The menu action generates the value `True`. This value is given to the update function which dynamically extends the group.

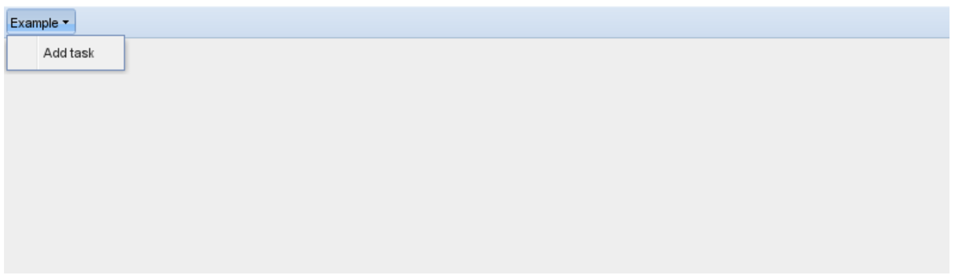


Figure 4.10: The GUI Generated for the Group Combinator Example With Empty Group

The situation when some tasks have been added is shown in Figure 4.11. Because those tasks accept the action `ActionClose`, the menu item triggering it is also added to the menu system. If this action is triggered this finishes the task which then disappears from the group. Because the dynamically added tasks return `False` no new task is added in this case. The update function generates a `Continue`.

As shown in in Figure 4.12 the menu of an unpinned and therefore floating task only includes the item to close the task. This improves usability because the other item does not refer to this task but to the group. The menu item for adding a task is still available in the group's menu bar.

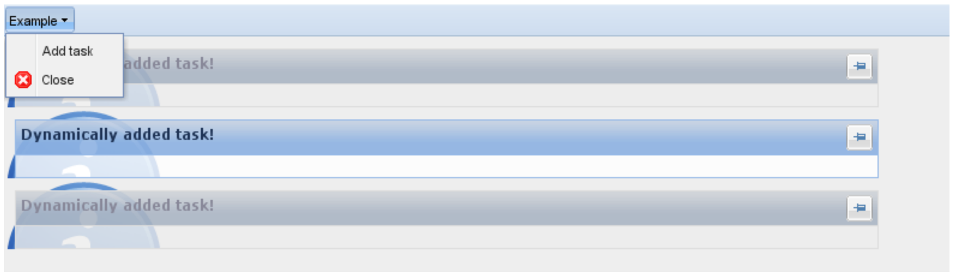


Figure 4.11: The GUI Generated for the Group Combinator Example With Dynamically Added Tasks

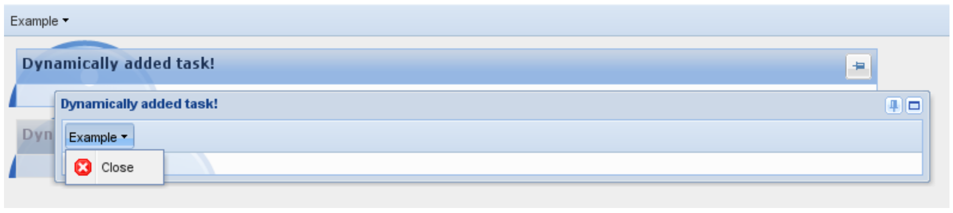


Figure 4.12: The GUI Generated for the Group Combinator Example With Unpinned Task

4.3.5 Core Group Combinator

The group combinator finally included in the WDL has some additional parameters:

```
group ::
        String
        String
        ((taskResult,Int) gState → (gState,PAction (Task taskResult) tag))
        (gState → gResult)
        gState
        [Task taskResult]
        [GroupAction taskResult gState shared]
    →
        Task gResult
    | iTask taskResult & iTask gState & iTask gResult & iTask shared
```

The first two parameters are a label and a description. In this version the state during execution and the final result have a different type. For this a conversion function is used as fourth parameter. The goal is to get a fully configurable construction in one step. A last difference is that the update function also gets an index for each finished task to indicate which of them finished.

All other group combinators are derived from this core group combinator.

4.3.6 Derived Dynamic Group Combinators

In many cases one wants to have a group to which tasks can be added dynamically, but has no need for the internal state. This is because the interaction between the group's task will be realised using the more powerful concept of shared data. In this case dealing with an accumulator function producing a new state and an actions is unnecessarily complex.

So a combinator for dynamic groups not using an update functions but letting the tasks generate new tasks or stop the group directly is defined:

```
:: GAction = GStop | GContinue | GExtend [Task GAction] | GFocus String
```

```
dynamicGroup :: [Task GAction] → Task Void
```

First a new set of actions similar to the ones described in Section 4.3.1 is defined. The reason why the existing parametrised type `PAction` is not used with the proper parameter is that this would result in a cyclic definition. Each task in such a group directly returns such an action, since the update function is left out. Consequently the only parameter of `dynamicGroup` is a list of initial tasks returning such an action. The constructor `GFocus` accepts a string instead of a tag, because using a tag adds unnecessary complexity to the type of tasks based on such a group. Actually this does not influence the expressiveness but only forces the programmer to convert a value to a string manually.

One might additionally want to use group actions for directly adding new tasks. For this an additional parameter has to be added:

```
dynamicGroupA :: [Task GAction] [GroupAction GAction Void shared] → Task Void
        | iTask shared
```

The type of the group actions (`GroupAction GAction Void shared`) reflects the fact that group actions here can add new tasks directly, by giving a value of type `GAction`. The internal state is of type `Void`, because it is not used at all. The type of the shared variable on which predicates can be used is not fixed.

A last combinator is used for groups in which the entire group can only be influenced by group actions, but tasks cannot dynamically extend or stop the group:

```
:: GOnlyAction = GOStop | GOContinue | GOExtend [Task Void] | GOFocus String

dynamicGroupAOnly:: [Task Void] [GroupAction GOnlyAction Void shared] → Task Void
        | iTask shared
```

The only thing that is changed here is that the tasks itself produce an empty result (`Void`).

Example

The example from Section 4.3.3 can be rewritten using `dynamicGroup`:

```
focusExample = dynamicGroup [task i \\ i ← [1..4]]
where
    task :: Int → Task GAction
    task i = GBFloating @>> (
                    enterInformationAbout "Task" i <<@ Tag i
        >>= λfocus. return (GFocus focus))
```

No state and no update function is needed here. Consequently the code is simpler than the original example, which would be even more complex if not the simplified version of the group combinator was used.

4.3.7 Nested Groups

Because a group is a task itself, groups can be added as children of other groups. A special meaning is given to nested groups because the obvious one would lead to strange results like windows inside of other windows. Like nested parallel combinators nested groups are flattened which means that all tasks are rendered as if they were in the top group. So for the user a possibly complex structure of nested groups looks like a single one.

The fact that groups are flattened does not mean that using nested groups is the same as using one group. First of all if a group which is child of another group is stopped, only the tasks inside this group are stopped. The parent group keeps running. Also actions for adding and focussing tasks only work on the group they are generated in. This makes it possible to structure tasks in a hierarchical way and to stop a subset of them at the same time. Only stopping the top group stops all tasks.

Also each group can have its own group actions. Each group inherits all actions from its top group. If the same action is used in a child and in a parent group both actions have an effect. An example of how this is used for different levels of actions is given in Section 5.1.5. For floating tasks the client only hides menu items referring to actions of the top group, because only they are certainly included in the group's menu. Actions of all child groups are handled like actions of tasks.

A remaining unsolved issue are mixed groups and parallel constructions. While it is no problem to use a parallel combinator as child of a group or a group as child of a parallel combinator, it is unclear what should happen if for instance a parallel combinator which has a group as child is child of another group.

Summary

In this section the concept of groups to which tasks can be added dynamically based on the result of other tasks is extended. First, tasks inside of groups can have different behaviours. They can for example be floating and therefore rendered as windows or be modal dialogs. Second, tasks can be focused by other tasks. Third, the possibility to add tasks triggered by menu actions is added. Finally, groups can be nested to define a hierarchical structure of tasks, with the possibility to assign actions to and close subsets of them.

This concept is essential for realising GUI applications. The extended concept essentially stays a group of tasks with some added annotations and an extra list of actions. Therefore it fits into the task structure of the WDL.

4.4 MDI Applications

MDI applications make it possible to work on multiple documents at the same time in the same application. In such applications new windows are used for each document opened by the user. An example are text editors in which the user can open an arbitrary number of files and edit them in separate windows.

With the new *iTask* paradigm it is possible to realise such applications by using grouped tasks each giving a view on another document. The state of all documents should be stored in such a way that tasks dealing with one document can only modify that one. But also some tasks should have the possibility to access all of them. This is for example necessary to check for unsaved work before closing the application. This problem can be solved in a general way such that programmers do not have to implement this state management separately for each MDI application.

Here a general combinator for creating MDI applications is discussed. This is an example of how functional programming concepts such as higher-order tasks can be used to supply the programmer with general behaviour which can help to implement concrete applications. Also it is shown how functionality can be encapsulated such that tasks can only use an abstract interface to manipulate a hidden application state.

For MDI applications a distinction is made between global and editor tasks (Section 4.4.1). The same distinction is made for actions as discussed in Section 4.4.2. After those basic concepts are explained in Section 4.4.3 it is shown how the state of the application, which includes the states of all editors, is stored. An interface for creating and accessing editors is given in Section 4.4.4. The main result is a high-order combinator for MDI applications which is introduced in Section 4.4.5. Finally in Section 4.4.6 an example of the structure of an MDI application is given.

4.4.1 Global and Editor Tasks

A MDI application consists of a number of tasks grouped together. In this way they can interact with the user using different windows. In general a distinction between global and editor tasks can be made. Global tasks do not deal with a specific document but operate on a global level. A simple example is a task providing a help document to the user. Then there are tasks for editing a specific document, called editors in the following. In general for each opened document there should be an editor.

4.4.2 Global and Editor Actions

Global actions are actions not modifying a specific document. Global tasks and editors are always generated by global actions. For instance an action for opening an about dialog is a global action. Also the action to generate a new or open an existing document is such an action.

Editor actions are actions inside an editor and always deal with a specific document. Saving or closing a file are examples for such actions.

4.4.3 Application State

The state of the entire application can be split up in two parts. First there is some global data which is only stored once, for instance some default options applied to all new documents. Then for each document which is edited there is a state storing information about the editing process, for instance the current content of the document. So there is a global state and a collection of an arbitrary number of editor states. For each editor state in that collection there should also be a task using it.

A first approach to store the different states is to store the global and each editor state in a separate shared variable. While the global state can be accessed by all tasks, each editor could generate and delete its own state. The problem with this approach is that this makes it impossible to access any editor state from a global task, which is needed for example to check if there is unfinished work. Because generally there is no possibility to access tasks from another one, a global task cannot even determine if there are any editor tasks remaining. If all tasks are grouped the only possibility to quit the application is to close the entire group and lose all unsaved data.

To improve upon this the global and all editor states can be stored inside one shared variable to which all tasks have access. Each editor can look up and access its own state in the collection of states. While this solves the problem of a global task not being able to access all editor's states, this has the drawback that bad encapsulation is achieved. Each editor can see and access also the data of all other ones and the programmer has to be careful not to manipulate data of other editors. Also each editor searching up and putting back its own state into a collection of possibly many states is not very efficient.

A combination of both methods gives a way to store the application state in such a way that it is possible to inspect all editor states for global tasks, but also let editors only access there own state and therefore achieves good encapsulation. As in the first approach the global and all editor states are stored in separate shared variables. The global state is accessible by all tasks and editor states only by the corresponding editors. But additionally a collection of references to all editor states is stored on a global level. A simple but sufficient type for storing this collection is:

```
:: MDIAppState editorState :== [DBid editorState]
```

Here simply a list of references to stored editor states is used. All editor states have the same type which has to be defined by the application's programmer to handle all kinds of documents the program is able to edit.

Each editor only gets a reference to its own state and can therefore not change the state of other ones. From a global level is it possible to access all editor states using the list of references.

4.4.4 Handling Editors

Giving global tasks direct access to the collection of editor states would be sufficient to realise the intended functionality. The disadvantage of such an approach is that global tasks could arbitrarily manipulate this collection. This could lead to inconsistent states of the application, for instance editor tasks without corresponding state.

To achieve a higher level of encapsulation the collection of editor states can also be hidden for global tasks. An interface for dealing with editors is provided to them. Creating and deleting editor states and putting them into the collection can be done automatically. Also it is impossible to remove an editor's state from the collection if the corresponding task is still running.

To provide global tasks with the possibility to deal with editors a collection of tasks put together into a record is used. The type of this record describes the interface for dealing with editors:

```
:: MDITasks editorState iterationState = {
    createEditor    :: MDICreateEditor      editorState,
    iterateEditors  :: MDIIterateEditors    editorState iterationState,
    existsEditor    :: MDIExistsEditor      editorState
    }

:: MDICreateEditor editorState :==
        editorState ((DBid editorState) → Task Void)
    →
        Task GAction

:: MDIIterateEditors editorState iterationState :==
        iterationState (iterationState (DBid editorState) → Task iterationState)
    →
        Task iterationState

:: MDIExistsEditor editorState :==
        (editorState → Bool)
    →
        Task (Maybe (DBid editorState))
```

The first task is used to create editors. It expects the editor's initial state as first argument. This initial value is stored in a shared variable and a reference is put into the collection of editors automatically. The second argument is the editor task which gets a reference to its state. After this task finishes the shared variable is deleted and its reference is removed from the collection automatically. The result of `createEditor` is a task which can dynamically be added to the application's group and therefore has to return a value of type `GAction`. Actually it always returns `GContinue`. The editor task itself is of type `Void` and can therefore not influence the group it is running in. The task `createEditor` is a nice example of a higher-order task.

The task iterateEditors is used to access all editor states without giving access to the collection of references directly. The task given as second argument is called for all editor states and gets a reference to each of them. During this iteration a state, which's initial value is given as first argument of iterateEditors, is accumulated. In each iteration the task gets the current value and produces a new value of this state. This could for example be used to count the total number of words in all opened documents. The reason why an accumulator task is used instead of a function is that the states can also be manipulated in this way. Also it is possible to prompt the user, for instance to ask if unsaved data should be saved. For simplicity reasons iterationState is quantified at the global level, which means that for each application the same type has to be used each time this task is used.

The last task existsEditor checks if an editor, for which a predicate given as first argument holds, exists. A reference to the state of the first editor for which the predicate holds is given as result. If the predicate holds for no editor the result is Nothing. Actually the same thing can be done using iterateEditors, but existsEditor is more convenient to use for its purpose and can also be more efficient because iteration stops at the first editor for which the predicate holds.

4.4.5 The MDI Combinator

Finally this combinator provides all functionality to create MDI applications:

```
mdiApplication ::
        globalState
        (
            (DBid globalState) (MDITasks editorState iterationState)
            → [GroupAction GAction Void globalState]
        )
    →
        Task Void
    | iTask, SharedVariable globalState
    & iTask, SharedVariable editorState
    & iTask iterationState
```

There are only two arguments. The first one is the initial global state. The second one is a function producing group actions which can be used to generate tasks added to the application's group, which is hidden in the combinator, dynamically. The group starts without tasks. For generating the group actions two things are provided by the combinator. The first one is a reference to a shared variable storing the global state. The second one is the collection of tasks for creating and accessing editors. Those things can be used by all tasks generated by the group actions. Actually giving the tasks to manipulate other editors to editor tasks is against the goal of encapsulation, but there is no way to prevent programmers from doing that.

After the application quits, which means that the task mdiApplication finishes, the combinator makes sure that all shared variables generated for the global state, the collection of editors and editor states are cleaned up.

4.4.6 Example

How the different states are handled for the case that there are two editors and one global task is visualised in Figure 4.13. On the top level there is a group with the three subtasks. There is a shared variable for the global state accessible by all tasks. Further each editor has its own state. There is a reference to both states in the collection *Editor States*. The global task cannot directly access this collection but can do so indirectly using the tasks stored in *MDI Tasks*.

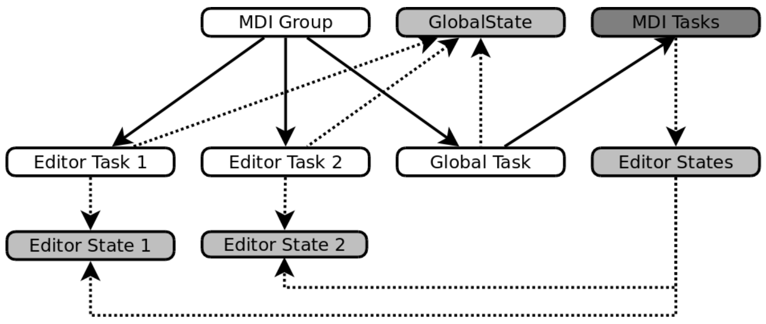


Figure 4.13: An Overview of How Different States in a MDI Application are Handled

A case study where this combinator is used to implement a multi-file text editor is done in Section 5.1.

Summary

In this section we show that MDI applications can be realised using the new paradigm. Further we show how functional concepts like higher-order tasks can be used to build a combinator providing basic MDI infrastructure. This combinator automatically keeps track of a collection of editor states and makes sure that it is kept in a consistent state and cleaned up properly. Also encapsulation can be achieved. Tasks for editing one document cannot access states corresponding to other ones. Tasks on a global level are provided with an interface to create and access editors, but cannot manipulate the collection of editor states directly.

4.5 Specialised Types

The *iTask* system already comes with a number of specialised types (Section A.3). This collection is extended with types for formatted text (Section 4.5.1), source code (Section 4.5.2) and colour (Section 4.5.3). As discussed in Section 3.1 this is the preferred solution for adding new widgets to the system.

4.5.1 Formatted Text

Formatted text is a very important part of many different kinds of software. It is not only used in traditional offline software like word processors but also in web applications like forums or wikis. Here it is shown how *iTask* can be extended with a specialised type for handling formatted text.

Formatted Text Editors on the Client

Many formatted text editors which can be used inside web applications are available. One widely used and powerful solution is *TinyMCE* [5], but also *ExtJS* has an editor for formatted text, which is not as powerful as other solutions, but is sufficient for this purpose. So the editor of *ExtJS* is used because it is easy to integrate.

What those editors have in common is that they are WYSIWYG editors, which means that the user sees the result immediately and can change the formatting of the text using controls. They behave similar as ordinary word processing software. In the end the result is *HTML* code representing the formatted text.

[5] `http://tinymce.moxiecode.com/`

Representation on the Server

Because the client side produces *HTML* code, it is convenient to also use *HTML* as representation of formatted text on the server. Of course, this forces all possible clients to understand and produce *HTML* but this is the case anyhow because also on other places in the current implementation of the *iTask* system *HTML* code is sent to the client.

A datatype for *HTML* is already present in the system. Using it would require parsing the code from the client at each event. This would produce much overhead. Because of this a string representation of the *HTML* code is stored. The programmer can parse it manually if needed. Another information stored in the type is which kind of controls the user can use in order to manipulate the text. The type for formatted text is defined as follows:

```
:: FormattedText = FormattedText String FormattedTextControls
:: FormattedTextControls =
    { alignmentControls :: Bool // Enable the left, center, right alignment buttons
    , colorControls     :: Bool // Enable the fore/highlight color buttons
    , fontControl       :: Bool // Enable font selection
    , fontSizeControls  :: Bool // Enable the increase/decrease font size buttons
    , formatControls    :: Bool // Enable the bold, italic and underline buttons
    , linkControl       :: Bool // Enable the create link button
    , listControls      :: Bool // Enable the bullet and numbered list buttons
    , sourceEditControl :: Bool // Enable the switch to source edit button
    }
```

The list of controls available is no general list of possible controls for editors but is just the list of controls the editor of *ExtJS* has. So the type could be more general in this point. Predefined values are provided to make handling of `FormattedTextControls` easier:

```
allControls :: FormattedTextControls     // all possible controls
noControls  :: FormattedTextControls     // no controls
```

Additionally there are some self explanatory auxiliary functions for dealing with formatted texts:

```
mkEmptyFormattedText    :: FormattedTextControls        → FormattedText
mkFormattedText         :: String FormattedTextControls → FormattedText

setFormattedTextSrc     :: String FormattedText → FormattedText
getFormattedTextSrc     :: FormattedText        → String

toUnformattedString     :: FormattedText Bool → String
```

The boolean parameter of `toUnformattedString` indicates if the cursor should be included into the resulting string. Details are discussed in the next section.

Dealing with the Cursor & Selections

There should also be the possibility to see and change the current cursor or selection on the server side. For example a simple search function needs to see the current cursor and set the selection to the next word found. Although it would be more convenient to implement such functionality on the client-side a solution where the cursor/selection is just seen as part of the value and can be manipulated by the server is more flexible.

There are two ways for indicating the current selection. The first one is to add offsets to the type `FormattedText` indicating the selection's position inside the code. This is a good solution for plain text. The drawback of this approach in combination with *HTML* code is that the offsets have to change in the case the code is changed. If a formatting is added but the text itself remains unchanged the offsets do not indicate the same position inside the text any more.

A better solution in this case is to use markers inside the string. In principle in *HTML* code a special tag could be used as marker. This has the drawback that one might want to get the unformatted version of a formatted text and still want to keep those markers. To add markers to an arbitrary string in *Clean* the following hack is used to define the markers:

```
SelectionStartMarker   :== "\0s"
SelectionEndMarker     :== "\0e"
```

A null-character should never occur inside a string but in *Clean* the length of a string is determined by the length of the array storing it and it is possible to process strings with null-characters inside.

To get rid of those markers this function can be used:

```
removeMarkers :: String → String
```

The implementation of the client side is very tricky. Because string with null-characters cannot be sent to the client, the server replaces those markers by special *HTML* tags before the code is sent. The client can then set the selection accordingly. The other way around if the client sends a value to the server first special tags are added at the position of the current selection. It turned out be very difficult to implement this behaviour in a reliable way. One problem is that the standard compliant interface for text selections is difficult to use. Also the *Internet Explorer* does not support the standard compliant interface for text selections and one has to find ways to work around this. Still the prototype, although it is not very stable, shows that in principle it is possible to deal with the cursor and selections in this way.

Example

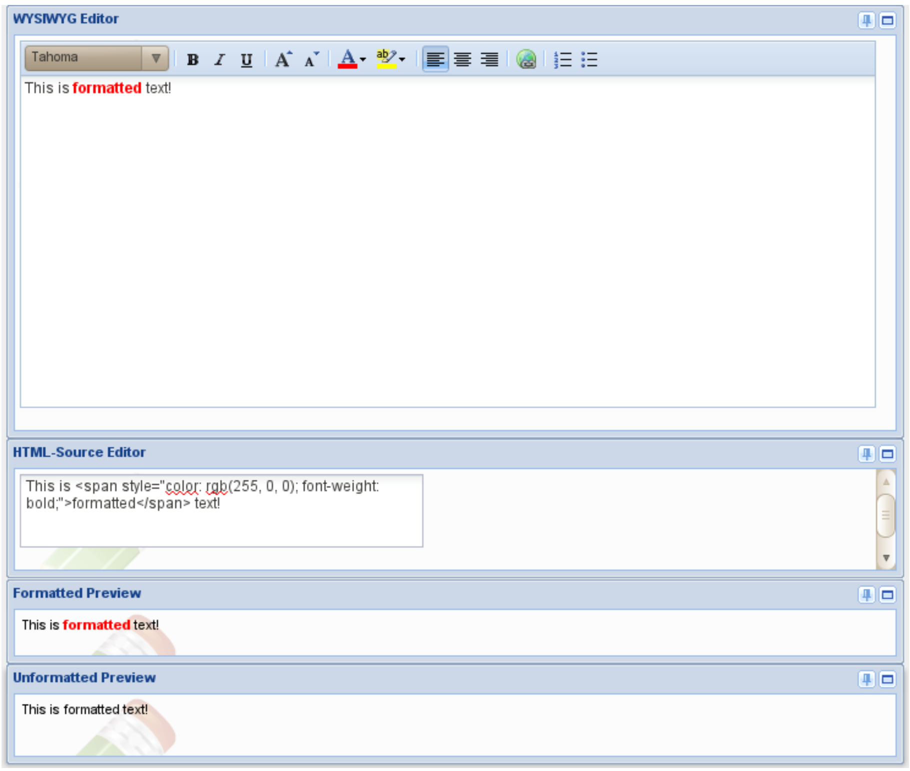
The example shows how different views on a formatted text can be defined:

```
formattedText :: Task Void                                                            1
formattedText =                                                                       2
                setMenus [Menu "Example" [MenuItem "Quit" ActionQuit]]                3
    >>|         createDB (mkEmptyFormattedText {allControls & sourceEditControl = False}) 4
    >>= λid.    dynamicGroupAOnly                                                     5
                  [(ignoreResult (t <<@ ExcludeGroupActions) <<@ GBFloating) \\ t ← tasks id] 6
                  actions                                                             7
    >>|         deleteDB id                                                           8
where                                                                                 9
    tasks sid =                                                                       10
        [ updateShared "WYSIWYG Editor"        [] sid [idEditor]                      11
        , updateShared "HTML-Source Editor"    [] sid [editor                         12
            { editorFrom    = λft              → Note (getFormattedTextSrc ft)        13
            , editorTo      = λ(Note src) ft   → setFormattedTextSrc src ft           14
            }]                                                                        15
        , updateShared "Formatted Preview"     [] sid [idListener]                    16
        , updateShared "Unformatted Preview"   [] sid [listener                       17
            {listenerFrom   = λft → Note (toUnformattedString ft False)}]             18
        ]                                                                             19
                                                                                      20
    actions :: [GroupAction GOnlyAction Void Void]                                    21
    actions = [GroupAction ActionQuit GOStop GroupAlways]                             22
```

First a shared variable with an initially empty formatted text is created (line 4). All possible controls except the one for switching to a *HTML* source view are available to the user. Then a group consisting of four different floating tasks representing the different views is defined (lines 5 – 7). After the group is finished the shared variable is deleted (line 8). The first view (line 11) is just an identity editor. Since the type of the shared state is `FormattedText` a WYSIWYG editor is shown to the user. The second view (lines 12 – 15) lets the user edit the *HTML* code of the formatted text.

Then there are two listeners. The first one (line 16) shows the formatted text to the user and the second one (lines 17, 18) first converts the text into an unformatted string.

The resulting application is shown in Figure 4.14. As soon as the value of the formatted text is changed either in the WYSIWYG or in the code editor, all other views are updated. The application is therefore also an excellent example for the usage of shared data.



This is **formatted** text!

```
This is <span style="color: rgb(255, 0, 0); font-weight:
bold;">formatted</span> text!
```

This is **formatted** text!

This is formatted text!

Figure 4.14: The GUI Generated for the Formatted Text Example

4.5.2 Source Code

The type described in this section is another example of a specialised type. Actually source codes consists of a string, but are rendered in a different way similar to the `Note` type, which is a string that is rendered in a multi-line textfield instead of a single-line one. In the type for source codes the code is represented by a string on the server side. How it is shown to the user, for instance with highlighted syntax, is left over to the client. The only additional information the server gives is the language of the source.

Source Code Type

Source codes are represented by this type with additional auxiliary functions:

```
:: SourceCode = SourceCode String SourceCodeLanguage
:: SourceCodeLanguage = JS | CSS | PHP | HTML | XML | Clean

mkSourceCode    :: String SourceCodeLanguage    → SourceCode
setSource       :: String SourceCode            → SourceCode
getSource       :: SourceCode                   → String
```

A source code consists of the code itself and its language. There is only a fixed set of supported languages, since the client also has to know how to highlight the syntax for each of the languages listed in `SourceCodeLanguage`. To add another language one would have to add a new constructor and provide the client with the capability to deal with the language.

Another possibility would be to use a specialisation for each language:

```
:: CleanSource  = Clean String
:: JSSource     = JS    String
...
```

The drawback of this solution is that for each of those types an instance of `gVisualize` generating and updating the proper control would have to be added.

Client Implementation

For the implementation on the client-side the *JavaScript* library *CodeMirror*[6] is used. It can already highlight the syntax for some popular languages. For our purposes we extended it with the ability to deal with *Clean* code. Additionally line-numbers are automatically rendered on the left.

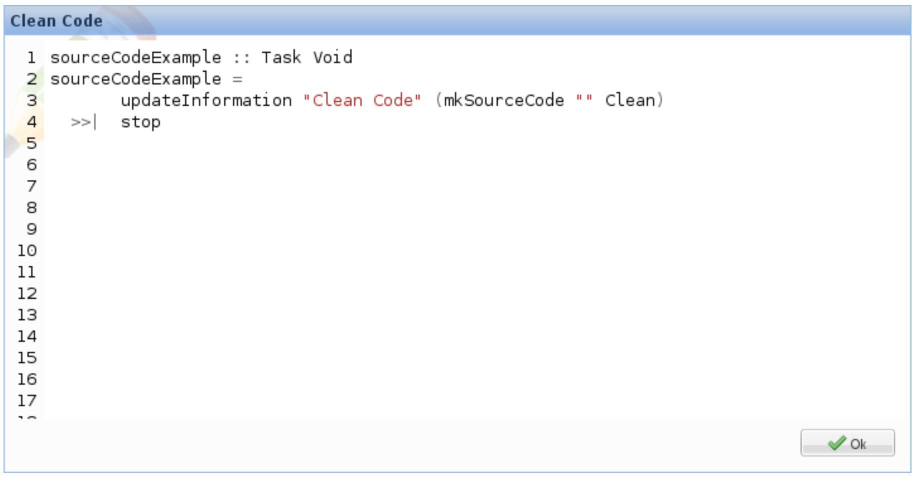
Because everything is computed on the client-side syntax highlighting is done instantaneously and it is possible to handle a large amount of code.

Example

The following example lets the user enter *Clean* code:

```
sourceCodeExample :: Task Void
sourceCodeExample =
        updateInformation "Clean Code" (mkSourceCode "" Clean)
    >>| stop
```

After the user entered the code of this example itself, it is represented on the client as shown in Figure 4.15.



```
sourceCodeExample :: Task Void
sourceCodeExample =
       updateInformation "Clean Code" (mkSourceCode "" Clean)
  >>|  stop
```

Figure 4.15: The GUI Generated for the Source Code Example

[6]http://marijn.haverbeke.nl/codemirror

4.5.3 Colour

Dealing with colours is another example of how an abstract representation on the server can be represented by a graphical user interface element on the client.

Colour Type

A colour is represented by a string with the hexadecimal representation of the colour's three RGB-components (for instance the colour white is represented by `"FFFFFF"`):

```
:: Color = Color String
```

Additionally some colours are predefined:

```
colorBlack     :== Color "000000"
colorRed       :== Color "FF0000"
colorGreen     :== Color "00FF00"
colorBlue      :== Color "0000FF"
colorYellow    :== Color "FFFF00"
colorFuchsia   :== Color "FF00FF"
colorAqua      :== Color "00FFFF"
colorPurple    :== Color "800080"
colorOrange    :== Color "FF9900"
colorWhite     :== Color "FFFFFF"
```

This representation is used because it is close to the one used for web-languages like *HTML* and *CSS* and is also used for the colour picker of *ExtJS*.

Client Implementation

For the client-side the colour picker of *ExtJS* is used. It gives a list of colours from which the user can choose. This restricts the user to a number of predefined colours. It would be possible to replace the implementation by a more general one which allows the user to pick an arbitrary colour.

Example

In this example the user can choose a foreground and a background colour:

```
:: ColorRecord =   { foregroundColor   :: Color
                   , backgroundColor   :: Color
                   }

derive class iTask ColorRecord

colorExample =
        updateInformation "Choose colors"   { foregroundColor = colorBlack
                                            , backgroundColor = colorRed
                                            }
```

First a record for storing the two colours is defined. The shorthand **derive class** is used to derive instances for all generic functions required for a class. The user can update the given start values for the colours. How this is represented on the client is depicted in Figure 4.16.

Summary

In this section specialised types for formatted text, source code and colour are added. They are required for the case studies (Chapter 5). It has been shown that

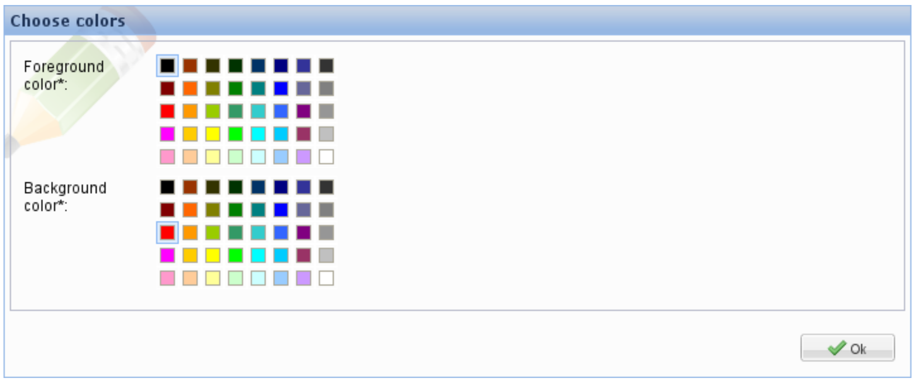


Figure 4.16: The GUI Generated for the Colours Example

this way of adding new widgets provides the author of *iTask* workflows with an abstract functional representation of the edited values. On the client-side already existing solutions can be integrated.

4.6 OS Tasks

There are situations in which one wants to directly use functionality of the operating system on the server. For example one wants to store information in files instead of stores and run external tools on them. Concretely, to implement an IDE one has to store source code files in the file system on the server and run a compiler.

It is possible to lift existing operating system tasks to the task domain, using the *lifting combinators* (Section A.2.5). However it is more convenient to add special tasks for this purpose, since they can fit better in the *iTask* WDL. For instance all tasks discussed in this section throw proper exceptions, which would not be possible using lifted library functions. Also parts of the functionality added here is not available in the standard libraries. Consequently in this section new tasks are added.

All tasks are realised calling functions written in *C* using the interface between *Clean* and *C*. The fact that currently the *iTask* server only runs on *Windows* platforms reduces the complexity of the problem. So just a number of *Windows* system calls have to be done.

Although the current implementation of the *iTask* server is restricted to *Windows*, the *iTask* WDL should stay platform independent. Because of this paths are represented in a platform independent way, described in Section 4.6.1. Then tasks dealing with the file system (Section 4.6.2) and for calling external processes (Section 4.6.3) are discussed.

4.6.1 Platform Independent Paths

A type for representing paths in a platform independent way is taken from the *Clean* library:

```
:: Path    = RelativePath [PathStep]
           | AbsolutePath DiskName [PathStep]

:: PathStep     = PathUp | PathDown String
:: DiskName   :== String
```

There are absolute and relative paths. Both consists of a sequence of steps, which are going the path up and down. If going down a name of a directory or file has to be given. Absolute paths also have a disk name, but it is only used for platforms using disk names and ignored otherwise.

There are situations in which one might want to append a number of path steps to a path not caring whether it is an absolute or relative one. For instance one might want to append a file name to the path of a folder the file is located in. To make this more convenient the following operator is defined as contribution of this thesis:

```
(+<) infixr 5 :: Path [PathStep] → Path
```

Further there is a function for converting platform independent paths into a platform dependent string representation:

```
pathToPD_String :: Path *env → (String, *env) | FileSystem env
```

A unique state representing the file system has to be provided.

To make it more convenient to use in *iTask* for this thesis also a task version hiding the unique state is added:

```
pathToPDString :: Path → Task String
```

Examples

A number of relative example paths and their encoded string versions in *Windows* format are given:

```
RelativePath []                                           "."
RelativePath [PathDown "test.txt"]                        ".\test.txt"
RelativePath [PathDown "someDir", PathDown "test.txt"]    "someDir\test.txt"
RelativePath [PathUp, PathDown "test.txt"]                "..\test.txt"
RelativePath [PathDown "someDir"] +< [PathDown "test.txt"] "someDir\test.txt"
```

The absolute path `AbsolutePath "c" [PathDown "someDir", PathDown "test.txt"]` is encoded as *"c:\someDir\test.txt"* on *Windows* and as *"/someDir/test.txt"* on *UNIX* platforms.

4.6.2 File System Tasks

A small collection for reading and writing text files, checking whether a file exists or is a directory, to create a directory and to get the application path are implemented. The collection is not complete and more or less arbitrary. It is based on facilities needed for realising the IDE case study (Section 5.2).

```
readTextFile    :: Path         → Task String
writeTextFile   :: String Path  → Task Void
fileExists      :: Path         → Task Bool
isDirectory     :: Path         → Task Bool
createDirectory :: Path         → Task Void
getAppPath      ::                Task String
```

Most basic tasks of the *iTask* language cannot fail, because they do not depend on factors outside of the *iTask* system. The situation is different for the tasks presented above. There are many reasons for those functions to fail, for instance a file might not exist or the server process might have no permission to access it. For this reason a number of exceptions, which can be thrown by tasks, as discussed in Section 2.1.5, are used to indicate the different kinds of errors:

```
:: FileException       = FileException String FileProblem
:: FileProblem         = CannotOpen | CannotClose | IOError
:: DirectoryException  = CannotCreate
```

The first argument of `FileException` is the platform dependent string representation of the file's path.

Example

The following program first lets the user input the path of a text file, reads its content and shows it to the user:

```
readFile :: Task Void
readFile =
        enterInformation "Enter the path of a text file:"
    >>= readTextFile
    >>= showMessageAbout "This is the content of the file:"
```

The problem here is that the user can give a path to a non-existing file. This will result in an exception thrown by `readTextFile`. This exception should be caught and an error message should be generated:

```
readFileErr :: Task Void
readFileErr =
    try
        readFile
        (λ (FileException path _) → showMessageAbout "Cannot read file:" path)
```

4.6.3 Calling Processes

There are situations in which one wants to call other processes on the server, for instance a compiler. One way to do this is the following task:

```
callProcessBlocking :: Path [String] → Task Int
```

The first argument is a path to an executable. The second one is a list of command-line arguments. The result of the task is the return code of the process. As the name suggests the call is blocking. This means that the server process blocks and can therefore not give a response to the client until the external process finishes. Even worse because currently there is no threading on the server side, also requests of other clients cannot be handled. Therefore this task should only be used for processes that are certain to terminate within a very short time.

A better solution is provided by the following task:

```
callProcess :: message Path [String] → Task Int | html message
```

It does essentially the same thing as the previous one, but instead of waiting until the external process finishes it stores its process identifier and shows a message given as first argument to the user. For this a monitor node in the task tree is generated. Each time the task is rebuilt it is checked whether the external process terminated in the meantime. If this is not the case the message is shown again. If the process terminated the task finishes and gives the process return code as result. Actually this means that the user has to push the refresh button (or the client has to refresh automatically within a certain interval) until the process finishes. Currently it is not possible to send a notification to the client immediately after the process finishes.

Actually non-blocking calls can also be realised using RPC functionality available for the *iTask* system. This makes it also possible to communicate with web services. For this functionality several threads are required. It is worked around the fact that threading is not possible with *Clean* by using a daemon written in *Java* to handle RPC requests. This generates much overhead for the simple case of a process running on the server. Using a web service is more complex than that.

So at the moment the solution presented in this section is preferable since it is less complex and generates less overhead.

Of course, also calling an external process can fail. Therefore both tasks may throw the following exception:

```
:: CallException = CallFailed String
```

Example

The following task shuts down the *Windows* computer the *iTask* server is running on and forces all applications to quit immediately:

```
shutdownExample = callProcessBlocking
  (AbsolutePath "c" [PathDown "windows", PathDown "system32", PathDown "shutdown.exe"])
  ["/s", "/f", "/t 0"]
```

An example how a compiler is called using a non-blocking call is given in Section 5.2.3.

Summary

In this section it is shown how I/O functionality like dealing with files on the server and calling processes can be added to the *iTask* system. The fact that tasks can have arbitrary side-effects makes it possible to do so in a straightforward way. Those tasks are also a good example of how errors can be structured using ADTs and used as exceptions to handle errors.

Chapter 5

Case Studies

In this section first it is described how a multi-document text editor can be implemented in Section 5.1. This is an example of a simple but commonly used MDI application. Then in Section 5.2 the implementation of a prototype, showing concepts needed for realising an IDE for *Clean*, is discussed.

5.1 Text Editor

A text editor is a good example, because it is a MDI application, but has a relatively simple user interface. Also this is a core component of an IDE. The purpose is to show that the extended *iTask* language is powerful enough to describe such an application. The text editor is based on the MDI combinator (Section 4.4).

In this section first the requirements for the text editor are discussed in Section 5.1.1. Then in Section 5.1.2 it is described how the documents this application works on, which are text files, are represented. The datastructures for storing the global and the editor states are discussed in Section 5.1.3. In Section 4.4.2 actions for MDI applications are separated into global and editor actions. For the case of the text editor the concrete actions of both types are defined in Sections 5.1.4 and 5.1.5. How those actions are made available to the user by the menu system is shown in Section 5.1.6. Finally the implementation of the global (Section 5.1.7) and editor tasks (Section 5.1.8), as defined in Section 4.4.1, is discussed in detail.

5.1.1 Requirements

As basis there have to be files which can be opened and would be stored in a file system for normal offline applications. Files could be stored in the server's file system for the case of this *iTask* application, for example using the tasks defined in Section 4.6.2. However the solution to store files using the database capabilities of the *iTask* system, which makes it possible to store values of arbitrary type in a convenient and type-safe way, is preferred here.

Because the focus here is showing how the text editor's user interface is realised, the database for storing files is very simple. Files are not organised, for instance in folders. Also there is no mechanism for locking files opened by other instances of the application.

The application should fulfil the standard requirements one would expect for a simple text editor application, like the possibility to open multiple files in separate windows or start working on a new file, to save files and to close them again. There should also be the possibility to quit the entire application. An important requirement is that no unsaved data is lost without asking the user to save it. Also the same file should not be opened more than once. An attempt to open an already opened file should result in focussing the window editing it.

The main requirement for editors is that there should be the possibility to edit the text file in a multi-line textfield. Also additional possibilities to manipulate the content of the file like replacing all occurrences of a string should be available. Finally the user should get the possibility to view some statistics about a file's content.

5.1.2 Text Files

Here a very simple database is used to store text files permanently. A text file is defined as:

```
:: FileName :== String
:: TextFile =   { fileId    :: (DBRef TextFile)
                , name      :: FileName
                , content   :: Note
                }
```

Each file has an identifier, a name and a content which is of type `Note`, because it is displayed as multi-line input-field in the user interface in this way. It is simple to make a permanent database for text files by using existing functionality of *iTask* and defining an instance of `DB` for this type:

```
instance DB TextFile where
    databaseId          = mkDBid "TextFiles"
    getItemId file      = file.fileId
    setItemId id file   = {file & fileId = id}
```

Here `"TextFiles"` is an identifier for the database which is used for all instances of workflows run by all users. So files are stored permanently in a database accessible by all workflow instances. There should be some kind of locking in case two users try to edit the same file, but this is left out for simplicity reasons here. It is possible to access the database of text files using the tasks discussed in Section A.1.2.

Upon this a more high level interface for accessing text files is built:

```
storeFile       :: FileName Note    → Task TextFile
setFileContent  :: Note TextFile    → Task TextFile
getFile         :: (DBRef TextFile) → Task TextFile
getAllFileNames :: Task [(FileName, Hidden (DBRef TextFile))]
```

First there is a task to store a file with a given name and content (`storeFile`) and to update the content of an existing file (`setFileContent`). Then an existing file can be retrieved from the database using `getFile`, which is more convenient to use than `dbReadItem`, because it assumes that the file exists and does not return a `Maybe`. Finally a list of all file names together with their identifiers can read from the database (`getAllFileNames`). The identifier is hidden because if the list is visualised for the user only the names should be visible without the identifiers.

5.1.3 Global State & Editor States

As discussed in Section 4.4.3 an MDI application has a global state and a number of editor states. The global state for this application is very simple. It is just an integer counting the number of new documents to be able to number them. (They are called "New Text Document 1", "New Text Document 2", ...) Consequently the first parameter given to the MDI combinator is just `0`.

Each editor has a state of the following type:

```
:: EditorState  = EditorState Note EditorFile
:: EditorFile   = NewFile Int | OpenedFile TextFile
```

The state consists of the current content of the edited file of type `Note`. Additionally information about the file which is edited is given. The edited file can either be a new file with a number or an existing opened file.

Some auxiliary functions are defined on the editor state:

```
hasUnsavedData  :: EditorState  → Bool
getFileName     :: EditorFile   → String
```

The first task checks if there is a difference between the current content of the editor and the content stored in the file. The second one returns the name of existing files. For new files it returns `"New Text Document"` followed by its number.

5.1.4 Global Actions

On the global level there are actions for opening a new or existing file, showing the about dialog and quitting the entire application. Those actions have in common that they do not refer to a specific edited file. They also have a meaning if no text files are opened at all. They correspond to the global actions defined in Section 4.4.2. So the list of group actions is given to the MDI combinator:

```
groupActions :: (DBid Int) (MDITasks EditorState Bool) → [GroupAction GAction Void Int]        1
groupActions gid mdiTasks=:{createEditor, iterateEditors} =                                     2
    [ GroupAction       ActionNew                                                               3
       (GExtend [newFile gid createEditor])                         GroupAlways                4
                                                                                                5
    , GroupAction       ActionOpen                                                              6
       (GExtend [openDialog mdiTasks <<@ GBFloating])               GroupAlways                7
                                                                                                8
    , GroupActionParam  actionOpenFile                                                          9
       (λfid → GExtend [open (DBRef (toInt fid)) mdiTasks False])   GroupAlways                10
                                                                                                11
    , GroupAction       ActionShowAbout                                                         12
       (GExtend [about <<@ GBAlwaysFloating])                       GroupAlways                13
                                                                                                14
    , GroupAction       ActionQuit                                                              15
       (GExtend [quit iterateEditors <<@ GBModal])                  GroupAlways                16
    ]                                                                                           17
                                                                                                18
actionOpenFile :== "openFile"                                                                   19
```

All actions can always be triggered (`GroupAlways`). The first action creates an editor editing a new file (lines 3, 4). The added task needs the reference to the global state since it has to derive the number of the new file and needs the MDI task to generate editors. Then there is an action to generate a dialog to open a file (lines 6, 7). The next action is a parametrised one, which is used by a menu showing recently opened files to directly open a file with a given identifier (lines 9, 10). Because a parameter is encoded as string it has to be converted into a file identifier in a somehow circuitously way. The third parameter `False` prevents that the file is added to the recently opened menu a second time. Then there are actions for showing a simple about dialog (lines 12, 13) and quitting the application (line 15, 16). The latter task makes use of `iterateEditors` to save unsaved data. The task for quitting the application has modal behaviour. Because of this all dialogs asking the user to save unsaved work are modal. All tasks are discussed in detail in Section 5.1.7.

5.1.5 Editor Actions

Then there are actions defined as editor actions in Section 4.4.2 which work on a specific edited file. They are saving and closing the file and opening a dialog for statistics or to replace text.

But there are also actions working on a specific file **and** a specific subtask related to that file. They are used for subtasks showing statistics or replacing text to close them and to trigger the "replace all"-action.

This can be modelled by using nested groups. For each edited file a new group is added as editor. Here the file specific actions are defined. Each of those groups always has an editor for the content of the file as child. Also there can be an arbitrary number of statistics and replace-dialogs, which define their own button actions:

```
 1  textEditorFile :: EditorStateRef → Task Void
 2  textEditorFile eid = dynamicGroupA [editorWindow eid <<@ GBFloating] (actions eid)
 3  where
 4      actions :: EditorStateRef → [GroupAction GAction Void EditorState]
 5      actions eid =
 6      [ GroupAction ActionSave
 7          (GExtend [save eid])                     (SharedPredicate eid noNewFile)
 8
 9      , GroupAction ActionSaveAs
10          (GExtend [saveAs eid <<@ GBModal])        GroupAlways
11
12      , GroupAction ActionReplace
13          (GExtend [replaceT eid  <<@ GBFloating])   (SharedPredicate eid contNotEmpty)
14
15      , GroupAction ActionStats
16          (GExtend [statistics eid  <<@ GBFloating])  GroupAlways
17
18      , GroupAction ActionClose
19          (GExtend [close eid <<@ GBModal])         GroupAlways
20      ]
21
22      noNewFile :: (SharedValue EditorState) → Bool
23      noNewFile SharedDeleted = abort "editor state deleted"
24      noNewFile (SharedValue (EditorState _ file)) = case file of
25          (OpenedFile _)  = True
26          _               = False
27
28      contNotEmpty :: (SharedValue EditorState) → Bool
29      contNotEmpty SharedDeleted = abort "editor state deleted"
30      contNotEmpty (SharedValue (EditorState (Note cont) _)) = cont ≠ ""
31
32  :: EditorStateRef :== DBid EditorState
```

First there is a task `editorWindow` initially added to the group (line 2). It lets the user edit the file's content using a textfield. It never finishes directly since it includes an interaction task with no actions. The only way to stop it is to stop the entire group and therefore closing the file. For the entire group there are actions for saving an already existing file (lines 6, 7) and for saving the editor's content as new file (lines 9, 10). The former should only be triggered if an existing files is opened which is checked by the predicate `noNewFile` (lines 22 – 26) on the editor's state. The case that the state is deleted should never occur and aborts the program. Further there are actions opening dialogs for replacing text (lines 12, 13) and showing statistics (lines 15, 16). An arbitrary number of such tasks can be added to the group. A predicate to check that the content is not empty (lines 28 – 30) is used to prevent starting a task replacing text for files with empty content. Finally the file can be closed (lines 18, 19). The reason why this action not just generates a `GStop` is that first the user might be given to possibility to save unsaved modifications of the file. All tasks are discussed in detail in Section 5.1.8.

In principle all subtasks inherit the group actions of the group above, but if they are rendered as floating windows only the group actions of the file specific group are used. So in the menu of an editor only file specific actions such as closing the file, but no global actions like quitting the application are available. Global action can only be triggered in the top menu bar of the group. In this way it is avoided that global action are repeated for each editor and mixed up with file specific ones. `ExcludeGroupActions` annotations are used for the statistics and replace-dialogs to avoid that all actions are repeated in the menus of the dialogs.

5.1.6 Menus

Both global and editor actions have to be made available to users. For this the menu system is used. There is only one menu structure defined for the entire application:

```
setMenus
    [ Menu "File"    [ MenuItem "New"             ActionNew        (hotkey N)
                     , MenuItem "Open..."         ActionOpen       (hotkey O)
                     , MenuName recOpenedMenu     (SubMenu "Recently Opened" [])
                     , MenuSeparator
                     , MenuItem "Save"            ActionSave       (hotkey S)
                     , MenuItem "Save As..."      ActionSaveAs     (hotkey A)
                     , MenuSeparator
                     , MenuItem "Close"           ActionClose      (hotkey C)
                     , MenuItem "Quit"            ActionQuit       (hotkey Q)
                     ]
    , Menu "Edit"    [ MenuItem "Replace..."      ActionReplace    (hotkey R)
                     ]
    , Menu "Tools"   [ MenuItem "Statistics..."   ActionStats      (hotkey T)
                     ]
    , Menu "Help"    [ MenuItem "About"           ActionShowAbout Nothing
                     ]
    ]

recOpenedMenu  :== "recOpened"

hotkey :: Key → Maybe Hotkey
hotkey key = Just {ctrl = True, alt = False, shift = True, key = key}
```

The sub-menu showing recently opened files has a name to make it more convenient to change it dynamically. For the hotkeys the control and shift key both have to be pressed together with a letter. This avoids conflicts with the browser's hotkeys. Most of them only use the control key.

The structure is used to automatically build appropriate menus for the different menu bars, as indicated in Figures 5.1 and 5.2. The menu bar of the application's main group includes global actions and the menu of a single editor window includes editor actions, while both menus are based on the same structure.

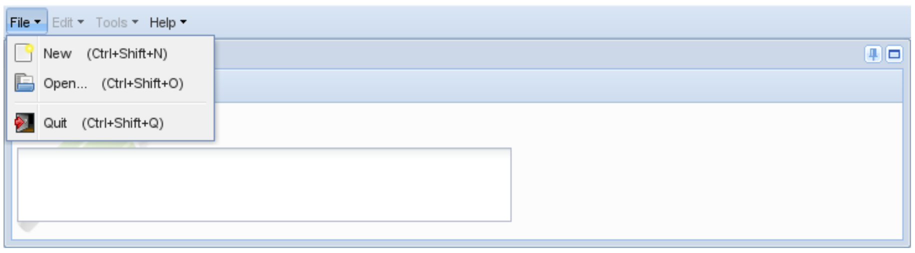


Figure 5.1: The Text Editor's Global File Menu

As can be seen in Figure 5.3 the menu of a pinned editor shows both global and editor actions.

5.1.7 Global Tasks

In this section the global tasks not dealing with a specific opened file are discussed.

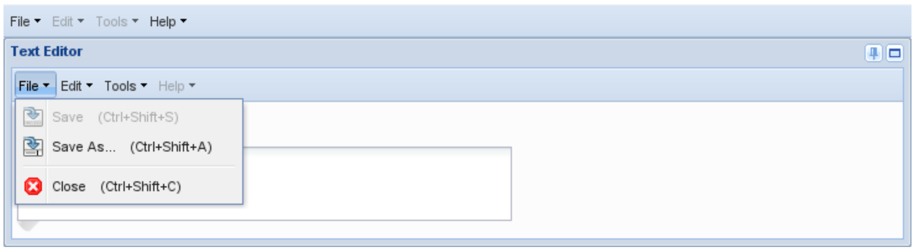


Figure 5.2: A File Menu of One Single Editor

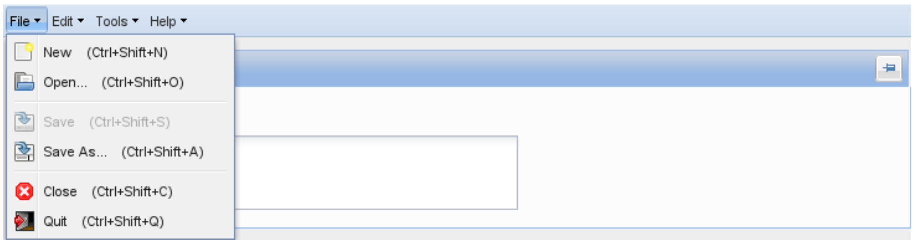


Figure 5.3: A Pinned Editor's File Menu

New File

```
newFile :: (DBid Int) (MDICreateEditor EditorState) → Task GAction                1
newFile gid createEditor =                                                         2
                 modifyDB gid inc                                                  3
    >>= λnewNum.  createEditor (EditorState (Note "") (NewFile newNum)) textEditorFile   4
```

For creating an editor for a new file first the number stored in the global state is incremented and given to the next task (line 3). Then a new editor is created using the task provided by the MDI combinator (line 4). The initial editor state represents an empty new file with the number just retrieved from the global state. The editor task `textEditorFile` is discussed in Section 5.1.5.

Open File

The task for creating an editor for an existing file is more sophisticated. It is either used by the open dialog, discussed in the next section, or by the menu for recently opened files.

```
open :: (DBRef TextFile) (MDITasks EditorState a) Bool → Task GAction
open fid {createEditor, existsEditor} addToRecOpened =
                existsEditor isEditingOpenendFile
    >>= λmbEid. case mbEid of
        Nothing =
                        getFile fid
            >>= λfile.  if addToRecOpened
                            (addToRecentlyOpened file.TextFile.name fid)
                            (return Void)
            >>|         return (GExtend [editor file])
        Just eid = return (GFocus eid)
where
    isEditingOpenendFile :: EditorState → Bool
    isEditingOpenendFile (EditorState _ file) = case file of
        NewFile _       = False
        OpenedFile file = (fid==file.fileId)

    addToRecentlyOpened :: String (DBRef TextFile) → Task Void
    addToRecentlyOpened name (DBRef id) =
                    getMenuItem recOpenedMenu
        >>= λitem.  case item of
          Just (SubMenu label entries)  = setMenuItem recOpenedMenu (newSubMenu label entries)
          _                             = stop
    where
        newSubMenu label entries =
            SubMenu                                                                        26
               label                                                                       27
               (take 5 [MenuItem name (ActionParam actionOpenFile (toString id)) Nothing:entries]) 28
                                                                                           29
    editor :: TextFile → Task GAction                                                      30
    editor file =                                                                          31
        GBFloating @>> createEditor                                                        32
            (EditorState file.TextFile.content (OpenedFile file))                          33
            textEditorFile                                                                 34
```

The task gets the file identifier, the MDI tasks and a flag indicating if the file should be added to the menu of recently opened files.

First it is checked if the file is already opened using `existsEditor` and the predicate defined in lines 13 – 16. If this is the case the editor window editing this file, which is tagged with the editor state's identifier, is focussed (line 11). Otherwise the file is retrieved from the database (line 6), possibly added to the menu of recently opened files (lines 7 – 9) and an editor task is added (line 10).

The task for adding a file to the menu (lines 18 – 28) first retrieves the submenu from the menu structure using the name that is given to it (line 20). The the submenu is dynamically changed and replaced by a new submenu (line 22). In this new submenu (lines 25 – 28) an entry for the new file is added at the top. The list only shows the last five opened files, which is achieved by applying `take` 5 to it.

The new editor (lines 30 – 34) is created using the MDI task for creating editors. The initial state consists of the file's current content and the file itself. The editor task is the group discussed in Section 5.1.5.

Open Dialog

```
openDialog :: (MDITasks EditorState a) → Task GAction                            1
openDialog mdiTasks =                                                            2
                getAllFileNames                                                  3
    >>= λfiles. if (isEmpty files)                                               4
        (       showMessageAbout "Open File" "No files to open!"                 5
            >>| continue                                                         6
        )                                                                        7
        (       enterChoiceA "Open File" buttons files                           8
            >>= λ(action,(name, Hidden fid)). case action of                     9
                ActionOk    =   open fid mdiTasks (Just name)                    10
                _           =   continue                                         11
        )                                                                        12
where                                                                            13
    buttons = [ButtonAction (ActionCancel, Always), ButtonAction (ActionOk, IfValid)] 14
```

To create an open dialog giving the user the possibility to choose a file to open first the list of all stored files is retrieved from the database (line 3). If there are no files to open the user gets a message about this (lines 5 – 7). The task `continue` is just an abbreviation for `return GContiue`. Otherwise the user can choose a file (line 8) and press an okay or cancel button (line 14). If the action is cancelled the tasks ends without further effect (line 11). Otherwise the file is opened using `open` (line 10).

An example how the task looks like if there are three documents in the database is given in Figure 5.4.

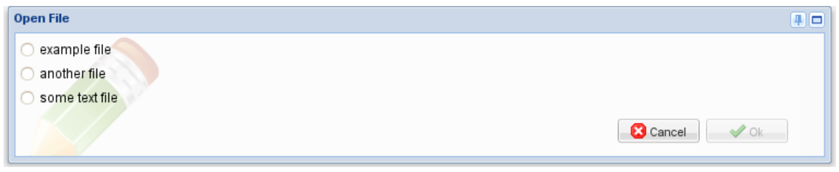


Figure 5.4: The Text Editor's Open dialog

About Dialog

```
about :: Task GAction
about =
        showMessageAbout "About" "iTextEditor August 2010"
    >>| continue
```

This task generates a dialog showing some information about the application.

Quitting the Application

There is a task for quitting the application. It makes sure that no unsaved data is lost:

```
quit :: (MDIIterateEditors EditorState Bool) → Task GAction                1
quit iterateEditors =                                                      2
                    iterateEditors False checkForUnsavedData               3
    >>= λcancel.    if cancel continue (return GStop)                      4
where                                                                      5
    checkForUnsavedData :: Bool String → Task Bool                         6
    checkForUnsavedData True  editor = return True                         7
    checkForUnsavedData False editor = requestClosingFile editor           8
```

This task iterates over all editors using the corresponding MDI task (line 3). For each editor a request to close it is done (line 8) using `requestClosingFile` which is conceptually an editor task and therefore described in Section 5.1.8. This task gives the user the possibility to cancel the action. In this case for all other editors no closing request is done (line 7) and the application is not quitted (line 4).

5.1.8 Editor Tasks

In this section tasks dealing with specific opened files are discussed.

The Title Listener

The title listener is a view on the editor state which gives the name of the file and additionally adds an asterisk in front of it if it contains unsaved data:

```
titleListener :: View EditorState
titleListener = listener
    { listenerFrom = λst=:(EditorState _ file) →
            if (hasUnsavedData st) "*" ""
            ⧺
            getFileName file
    }
```

Main Editor Window

```
editorWindow :: EditorStateRef → Task GAction                                          1
editorWindow eid =                                                                     2
        updateShared "Text Editor" [] eid [titleListener, mainEditor] <<@ Tag eid      3
    >>| continue                                                                       4
where                                                                                  5
    mainEditor = editor                                                                6
        { editorFrom    = λ(EditorState cont _)          → cont                        7
        , editorTo      = λnewCont (EditorState _ file) → EditorState newCont file     8
        }                                                                              9
```

The main editor window gives two views on the editor state. First it shows the name of the file using the title listener, then it defines an editor editing the content (lines 6 – 9). The interaction task is tagged with the reference to the state (line 3) to make it possible to focus the window in which the task is running. The task has no actions which means that the only way to stop it is to stop the group it is running in. Consequently as long as a file is opened exactly one main editor window exists for it.

How this task looks like is shown in Figures 5.2 and 5.3.

Saving Files

The first task is used to save an existing file or call `saveAs` for new files:

```
save :: EditorStateRef → Task GAction                                                  1
save eid =                                                                             2
                  readDB eid                                                           3
    >>= λeditor.   case editor of                                                      4
        EditorState txt (OpenedFile file) =                                            5
                      setFileContent txt file                                          6
            >>= λfile.  writeDB eid (EditorState file.TextFile.content (OpenedFile file))   7
            >>|         continue                                                       8
        _ = saveAs eid                                                                 9
```

In the case the editor is editing an existing file (line 5) the current content is written to the database (line 6) The modified file is also put in the editor state (line 7). If the editor is editing a new file the task for saving a new file is used (line 9) which makes this task more robust.

For saving a new file this task is used:

```
saveAs :: EditorStateRef → Task GAction                                                1
saveAs eid =                                                                           2
                     enterInformationA "Save As: enter name" buttons <<@ ExcludeGroupActions   3
    >>= λ(action,name).case action of                                                  4
        ActionOk =                                                                     5
                                       readDB eid                                      6
            >>= λ(EditorState txt _).       storeFile name txt                         7
            >>= λfile=:{TextFile|content}.  writeDB                                    8
                                               eid                                     9
                                               (EditorState content (OpenedFile file))  10
            >>|                             continue                                   11
        _ = continue                                                                   12
where                                                                                  13
    buttons = [ButtonAction (ActionCancel, Always), ButtonAction (ActionOk, IfValid)]  14
```

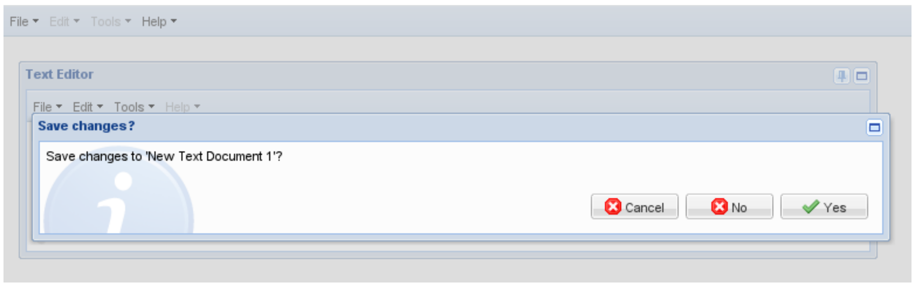
Here first the user has to enter a name for the file (line 3). If the okay button is pressed (line 5), the new file is stored (line 7) and the editor state is updated (lines 8 – 10).

Closing Files

```
close :: EditorStateRef → Task GAction                                                 1
close eid =                                                                            2
                  requestClosingFile eid                                               3
    >>= λcancel.   if cancel continue (return GStop)                                   4
                                                                                       5
requestClosingFile :: EditorStateRef → Task Bool                                       6
requestClosingFile eid =                                                               7
                                    readDB eid                                         8
    >>= λstate=:(EditorState _ file). if (hasUnsavedData state)                        9
        (                   ExcludeGroupActions @>>                                    10
                            showMessageAboutA "Save changes?" buttons (question file)  11
            >>= λaction.    case action of                                             12
                                ActionCancel    = return True                          13
                                ActionNo        = return False                         14
                                ActionYes       = save eid >>| return False            15
        )                                                                              16
        (return False)                                                                 17
where                                                                                  18
    buttons =   [ ButtonAction (ActionCancel, Always)                                  19
                , ButtonAction (ActionNo, Always)                                      20
                , ButtonAction (ActionYes, Always)                                     21
                ]                                                                      22
                                                                                       23
    question file = "Save changes to '" +++ getFileName file +++ "'?"                  24
```

The task is splitted up in two parts because `requestClosingFile` is also used if the entire application is quitted. The result of `requestClosingFile` indicates if the user cancelled the action. If the file has no unsaved data nothing happens and the user has no possibility to cancel the action (line 17). If there is unsaved data the user is asked whether the file should be saved (line 11). The user can cancel the action (line 13), discard the changes (line 14) or save them (line 15). For saving the file save is used which is implemented robust enough to work for existing but also for new files.

How the modal dialog to request closing a file looks like is shown in Figure 5.5.



Save changes to 'New Text Document 1'?

Figure 5.5: The Modal Dialog Used to Request Saving Changes

Replacing Text

```
:: Replace =    { searchFor     :: String                                                         1
                , replaceWith   :: String                                                         2
                }                                                                                 3
                                                                                                  4
derive class iTask Replace                                                                        5
                                                                                                  6
replaceT :: EditorStateRef → Task GAction                                                         7
replaceT eid = replaceT‘ {searchFor = "", replaceWith = ""}                                       8
where                                                                                             9
    replaceT‘ :: Replace → Task GAction                                                          10
    replaceT‘ repl =                                                                             11
                                ExcludeGroupActions @>>                                          12
                                updateInformationA "Replace" buttons repl                        13
        >>= λ(action, repl).    case action of                                                   14
                                    ActionReplaceAll =                                           15
                                            modifyDB eid (dbReplaceFunc repl)                    16
                                        >>| replaceT‘ repl                                       17
                                    _ = continue                                                 18
                                                                                                 19
    buttons = [ButtonAction (ActionClose, Always), ButtonAction (ActionReplaceAll, IfValid)]     20
                                                                                                 21
    dbReplaceFunc repl (EditorState (Note txt) file) =                                           22
        EditorState (Note (replaceSubString repl.searchFor repl.replaceWith txt)) file           23
                                                                                                 24
ActionReplaceAll :== ActionLabel "Replace all"                                                   25
```

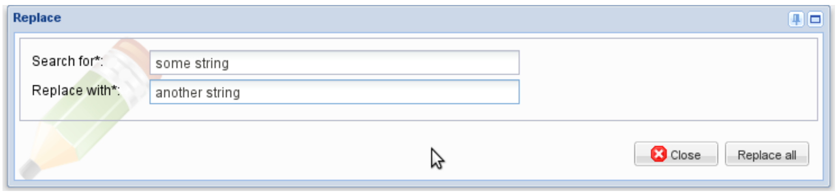
The behaviour of this task is that the user can type in a string which is searched for and another string by which all occurrences of this string is replaced. This information is represented by a value of the type Replace. After the text has been modified the task should not end but still should give the user the possibility to replace another string. The fields should show the last values the user typed in.

To achieve this the task replaceT‘ is used which expects an initial replace-value as input and asks the user to update it (lines 12, 13). As long as the user wants to replace text (line 15) the editor state is updated accordingly (lines 16, 22, 23) and the task calls itself recursively (line 17). The entire task stops if the user presses the cancel button (line 18). The user does not notice that a new task is started recursively, because the new task after the text has been modified generates the same user interface the user saw before. The task replaceT itself just calls replaceT‘ with empty fields as initial value (line 8).

In Figure 5.6 is it depicted how this task will be presented to the user.

Statistics

```
:: TextStatistics = { lines         :: Int
                    , words         :: Int
```



Search for*: some string
Replace with*: another string

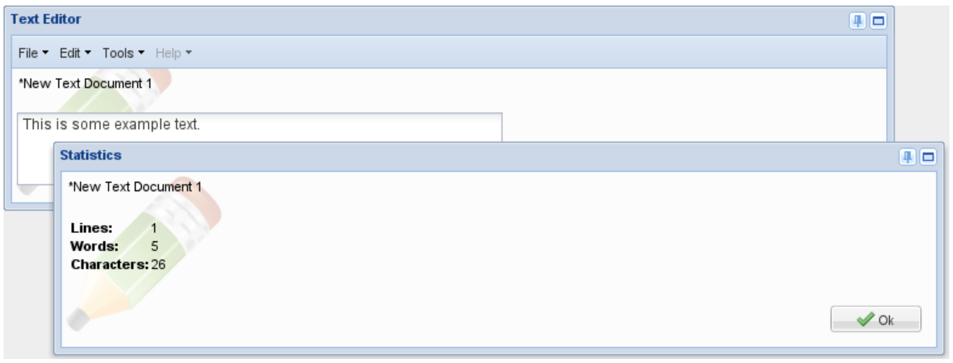
Figure 5.6: The Dialog to Replace Substrings in Edited Files

```
                    , characters    :: Int                                          3
                    }                                                               4
                                                                                    5
derive class iTask TextStatistics                                                   6
                                                                                    7
statistics :: EditorStateRef  → Task GAction                                        8
statistics eid =                                                                    9
        ExcludeGroupActions @>>                                                    10
        updateShared "Statistics" buttons eid [titleListener, statsListener]       11
    >>| continue                                                                   12
where                                                                              13
buttons = [ButtonAction (ActionOk, Always)]                                        14
                                                                                   15
statsListener = listener {listenerFrom =  λ(EditorState (Note text) _) →           16
    let txt = trim text                                                            17
    in                                                                             18
        { lines      = length (split "\n" txt)                                     19
        , words      = length (split " " (replaceSubString "\n" " " txt))          20
        , characters = textSize txt                                                21
        }                                                                          22
    }                                                                              23
```

Two views on the editor's state are given. The first one is a title listener to indicate for which file statistics are shown. The second listener (lines 16 – 23) uses some string functions to calculate the number of lines, words and characters and presents it to the user by putting it into a value of type `TextStatistics`.

The shared data mechanism of the *iTask* system automatically updates the view instantaneously each time the content of the file changes either because the user typed in something in the main editor or because strings are replaced by a replace task.

In Figure 5.7 an example of a text document with some text filled in together with the corresponding statistics is given.



*New Text Document 1

This is some example text.

*New Text Document 1

Lines: 1
Words: 5
Characters: 26

Figure 5.7: An Example for Text File Statistics

Summary

In this section the implementation of a multi-file text editor is discussed in detail. This shows that this kind of MDI application can be implemented using the *iTask* paradigm. Actually the *iTask* paradigm turned out to be very suited for this kind of application.

Using the menu system and nested groups makes it possible to structure possible user actions in a declarative way. Different menu bars for the different contexts are created automatically based on a single menu structure. Tasks accessing one file like the main editor, the replace and the statistics dialog can be implemented in such a way that they are independent. Changes made by one task are automatically reflected in the other ones, without one task being aware of the other ones. Finally the interface provided by the MDI combinator turned out to be very convenient to check if files are already opened or to request saving changed data.

5.2 A Clean IDE

With the MDI text editor we already showed that it is possible to realise essential functionality needed for an IDE. However an IDE is a greater challenge because more sophisticated features, for instance syntax highlighting, I/O like calling a compiler and also configuration issues, are involved.

An IDE also requires a complex program logic for instance to determine which files belong to a project or which library modules are used. Also functionality like searching up definitions in a project requires more than simple string comparison. For the case of the *Clean* IDE this is already implemented in the old IDE[1] also written in *Clean*. Those issues have nothing to do with the *iTask* paradigm and are therefore not handled in this thesis.

Only a collection of issues essential for realising an IDE are implemented in a small prototype which is able to compile a project with only one source code file using the standard environment. Actually we know how to realise handling of multiple files, but keeping track of which files are needed for a project is a lot of administrative work. Although functions for doing this are available in the code of the old IDE, it is not that easy to retrieve this functionality from existing code. The prototype developed in this project forms the basis for a complete IDE and shows that it is possible to type in syntax highlighted code, to compile this code on the server-side and send it to the client, to configure the application and to deal with errors using exceptions.

First the global application state of this prototype is discussed (Section 5.2.1). Then in Section 5.2.2 it is discussed how syntax highlighting can be realised. This is followed by a discussion of how projects can be stored and compiled in Section 5.2.3. The prototype requires some GUI constructions, like a configuration wizard, which can be generalised to GUI patterns which are discussed in Section 5.2.4.

5.2.1 Application State

This is the datastructure used as global application state:

```
:: AppState = { srcEditorContent  :: String
              , ideConfig         :: IDEConfig
              , syntaxHighlColors :: SyntaxHighlighterColors
              }
```

First it contains the editor's current content as plain string. The field `ideConfig` is mainly used to store and compile the code and discussed in Section 5.2.3. The colours used for syntax highlighting are determined by the field `syntaxHighlColors` described in Section 5.2.2.

[1] http://wiki.clean.cs.ru.nl/Download_Clean

5.2.2 Syntax Highlighting

Syntax highlighting for *Clean* programs can be done using the source code type (Section 4.5.2). Here an alternative solution using a formatted text (Section 4.5.1) with highlighted syntax as view on a plain string is used. Since the syntax highlighting is done on the server this solution is more flexible.

Because the colours are determined on the server-side they are configurable. The following type is used to give colours to the different constructs of *Clean* code:

```
:: SyntaxHighlighterColors =  { keywords              :: Color
                              , typeDefinitions       :: Color
                              , singleLineComments    :: Color
                              , multiLineComments     :: Color
                              , strings               :: Color
                              , characters            :: Color
                              , numbers               :: Color
                              }
```

This can directly be updated by the user (using `updateInformation`) as depicted in Figure 5.8.

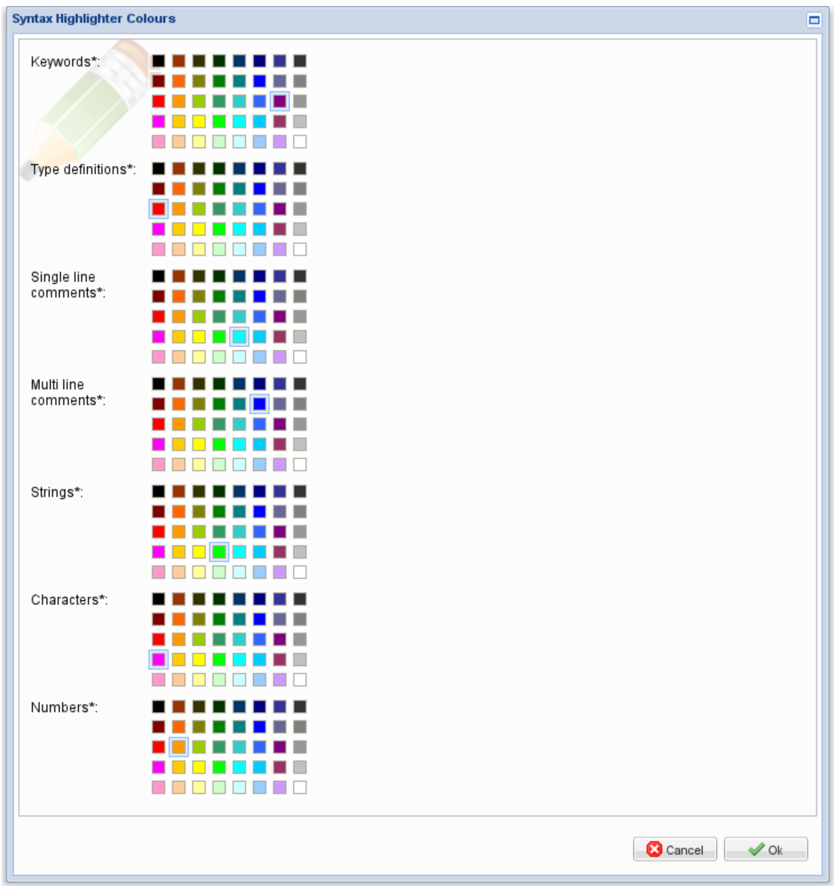


Figure 5.8: The Dialog Used to Choose the Colours for Syntax Highlighting

What is needed now is a function for converting source code in plain text to formatted text representing the code with highlighted syntax using given colours:

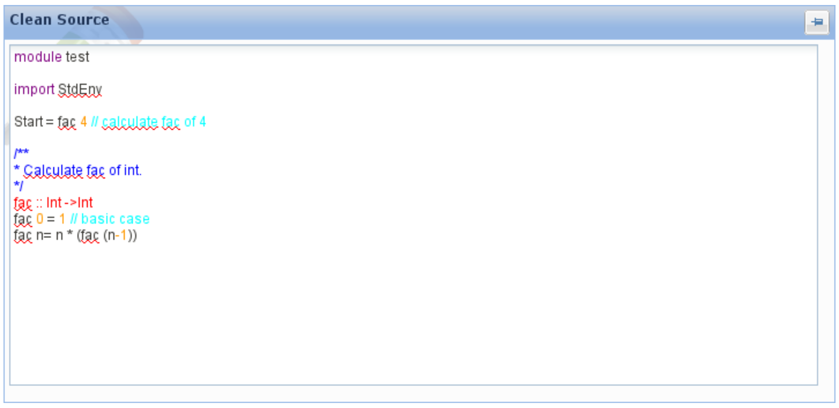
```
highlightSyntax :: String SyntaxHighlighterColors → FormattedText
```

The implementation of this function is not discussed here.

The editor using this function is defined as:

```
srcEditorView :: View AppState
srcEditorView = editor
    { editorFrom    = λstate →
                      highlightSyntax state.srcEditorContent state.syntaxHighlColors
    , editorTo      = λft state →
                      {state & srcEditorContent = toUnformattedString ft True}
    }
```

The `editorFrom` function retrieves the current content of the editor and the colours from the application state and feeds them to the syntax highlighter function. With a formatted text control without controls this looks like shown in Figure 5.9. Converting the formatted text back to plain text, which is then put back into the application state, is done using `toUnformattedString`. Cursor markers are kept inside the string to leave the cursor on the same position before and after syntax highlighting is performed.



```
module test

import StdEnv

Start = fac 4 // calculate fac of 4

/**
* Calculate fac of int.
*/
fac :: Int ->Int
fac 0 = 1 // basic case
fac n= n * (fac (n-1))
```

Figure 5.9: Example Code Syntax Highlighted Using Formatted Text

Because the colours are computed by the server, the formatted text should send an event to the server not only if it is unfocussed, but more often to colour for instance a new keyword instantaneously. An event could be generated at each keystroke, but this would be a waste of bandwidth. So a timer is used to send an event if there has been no new keystroke for about a half second.

But sending events to the server while the user is working on the code leads to timing problems. If between the event and the response from the server the user types in something, this is discarded at the moment the newly highlighted source from the server arrives and replaces the current one. Blocking user inputs between a request and its response would solve the problem but would make the application much less user-friendly. A more complex solution is to discard the response from the server if the content changed after the event and send a new request. So the content is only updated if there has been no user input for a period given by the timer plus the time the answer from the server arrives. In practice this could be a usable solution since normally source code is not entered without some pauses in between.

However implementing this in a stable way such that no user input is lost, has turned out to be not that easy. Also there are the already discussed problems with the formatted text control (Section 4.5.1). Consequently using the source code type is the more stable and interactive but less flexible way at the moment. A flexible and fast way would be to compile the syntax highlighter function, written in *Clean*, in such a way that it runs on the client.

5.2.3 Storing & Compiling Projects

In general there are several options how source code is compiled in a server/client application. The first choice is whether the compilation is done on the server or on the client. This choice is easy here because there is no general way to call a compiler on the client-side if it runs inside a web browser. On the server the *iTask* process can easily be given permissions to run a compiler.

Also there has to be the possibility to run the compiled program, again with the choice on which side this is done. Doing so on the server might not be a good idea, because there has to be some way to interact with the program from the client-side. Also the process running on the server must have very limited privileges. It is not a good idea to let clients submit arbitrary code which can access the entire server. But also with limited privileges it very difficult to avoid security problems.

Sending the executable to the client gives the problem of platform dependency. The client, written in *JavaScript*, is platform independent, but a *Windows* executable received by the browser can only be executed on *Windows* platforms. A solution would be to detect the operating system of the client and use cross compilers to generate an executable for the platform the client is running on. The problem is that there is only a *Windows* version of the most recent *Clean* system.

Another options to solve the problem in a platform independent way would be to translate the program into some platform independent language instead of machine code. For instance *Clean* code could be translated into *JavaScript* which then runs into the browser. A problem here is doing I/O. A virtual console could be used to run console applications inside the browser, but other operations like writing to files cannot be done. Still giving this possibility as additional option would be an elegant solution for a restricted class of applications. Realising this is an issue for further research.

The pragmatic solution it is chosen for in this thesis, is to generate a *Windows* executable on the server-side and send it to the client. The user can then download and run it on a *Windows* machine. It is not a very elegant and user-friendly solution, but it is functional enough to compile and run any application written in *Clean*.

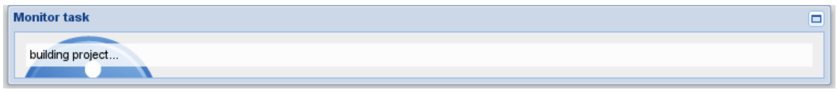
Compiling a *Clean* program is not that trivial, because the compiler, the code generator and the linker are separate executables. Also for being efficient one has to check which parts of the program have to be recompiled. Fortunately this is already solved by the old IDE and there is the possibility to give a project as argument and let it batch-build by the IDE executable. In the configuration there is a path where projects are stored and also a path to the old IDE executable:

```
:: IDEConfig = { oldIDEPath    :: Path
               , projectsPath  :: Path
               }
```

For generating a project some functionality has been taken out of the old IDE's code. A very simple project with one file in which the standard environment can be used is written into the file system of the server. Then this project can be compiled using this task:

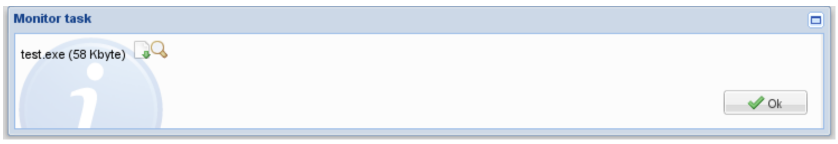
```
compileToExe :: (DBid AppState) → Task Document
compileToExe sid
    ♯ compileToExe‘ = try compileToExe‘ handleCallException
    ♯ compileToExe‘ = try compileToExe‘ handleReadLogException
    ♯ compileToExe‘ = try compileToExe‘ handleStringExceptions
    = compileToExe‘
where
     compileToExe‘ =
                        getConfig sid
        >>= λconfig.    getAppPath
        >>= λappPath.   pathToPDString config.projectsPath
        >>= λprjPath.   callProcess
                            "building project..."
                            config.oldIDEPath
                        ["--batch-build \"" ++ appPath ++ prjPath ++ "\\test\\test.prj\""]  15
    >>= λret.       case ret of                                                           16
                        0    =                  importDocument (prjPath ++ "\\test\\test.exe") 17
                                >>=             return                                   18
                        _   =                   readTextFile    (   config.projectsPath   19
                                                                    +<                   20
                                                                    [ PathDown "test"     21
                                                                    , PathDown "test.log" 22
                                                                    ]                    23
                                                                )                        24
                                >>= λlog.       throw   (CompilerErrors                  25
                                                            (filter ((≠) "") (split "\n" log)) 26
                                                        )                                27
                                                                                         28
handleCallException (CallFailed path) =                                                  29
    throw (CannotRunCompiler ("Error creating process '" ++ path ++ "'"))                30
handleReadLogException (FileException path _) =                                         31
    throw (CannotRunCompiler ("Unable to retrieve compiler errors from '" ++ path ++ "'")) 32
handleStringExceptions str =                                                             33
    throw (CannotRunCompiler str)                                                        34
```

The task assumes that there is a test project at some hard coded path, compiles it and returns the generated executable as *iTask* document. If one would not care about possible errors only lines 9 – 18 would be needed. First the configuration is retrieved from the application state using some auxiliary function (`getConfig`). Additionally the application's path is retrieved and the path for storing projects is converted into a platform dependent string representation (lines 10, 11). Then the old IDE is called to compile the project (lines 12 – 15). What the user gets to see while the process is running is shown in Figure 5.10. If there are no errors the process returns 0 after compilation. The generated executable is imported as document using `importDocument` (Section A.1.4). The document is then returned as result of the task (line 18). The resulting document can be downloaded by the user as depicted in Figure 5.11.



building project...

Figure 5.10: GUI Shown while the User is Waiting for the Compilation to Finish



test.exe (58 Kbyte)

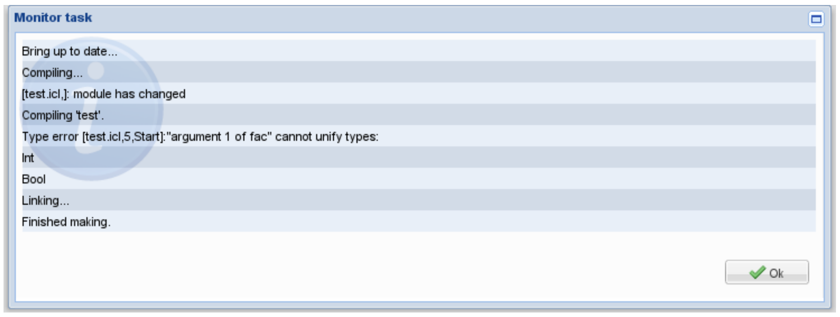
Figure 5.11: A Compiled Executable Provided as *iTask* Document

During compilation the compiler might find errors in the code. Actually this will occur more often than that the code is correct. In this case errors can be read from a log file generated by the compiler (lines 19 – 24). The content of the file consisting of several lines is splitted into a list of strings and empty lines are filtered out. After this those messages are thrown as exception (lines 25 – 27). This exception can be caught by the caller of `compileToExe` and shown to the user (Figure 5.12).

The type of the exception thrown by `compileToExe` is:

```
:: CompilerException = CannotRunCompiler String | CompilerErrors [String]
```

The first kind of exception that can occur (`CannotRunCompiler`) means that the compiler could not even be run properly. There are several places were this can go wrong. Trying to call a process can cause a `CallFailed` exception, trying to read in the error log can cause a `FileException` and also the task for importing a document can throw an exception which is just a string.



```
Bring up to date...
Compiling...
[test.icl,]: module has changed
Compiling 'test'.
Type error [test.icl,5,Start]:"argument 1 of fac" cannot unify types:
Int
Bool
Linking...
Finished making.
```

Figure 5.12: An Example of Compiler Error Messages

In lines 3 – 6 of `compileToExe` the actual task is surrounded by three `try` tasks, each responsible for catching a different kind of exception. Those exceptions are converted into proper `CannotRunCompiler` exceptions (lines 29 – 34) before they are thrown out of the function.

5.2.4 GUI Patterns

While developing GUI applications there are several repeating patterns. Similar to the idea of the MDI combinator (Section 4.4) higher-order tasks can be used to implement a certain behaviour once in a general way. It would be possible to make a collection of general GUI patterns. Here two GUI tasks used for the IDE are presented, serving as example.

Editing Options

Editing options is similar to having a view on a shared state. From some state including all options of the entire application a small portion is extracted, which is then modified by the user. In contrast to a view the state is only changed after the user confirms the change. If the user cancels the action the state is not changed at all. A task for doing this is:

```
editOptions ::
        description state (state → opts) (opts state → state)
    →
        Task state
    | html description & iTask state & iTask opts
```

The second and third argument corresponds to the get and putback functions of lenses (Section 4.2.9). The user can edit the value of type `opts` retrieved from the state and has the choice to confirm or cancel the modification by pressing a button. The possibly modified state is the result of the task.

For editing options inside a shared variable it would be more convenient to use a task which immediately changes the value of that variable like:

```
editOptions ::
        description (DBid state) (state → opts) (opts state → state)
    →
        Task Void
    | html description & iTask state & SharedVariable state & iTask opts
```

The reason why the first version of the combinator is used is that also options stored in the project file, influencing how the project is compiled, can be edited. The first version provides a more general solution, which works no matter how the state is stored. In the application two task are build upon this: one for editing options stored in the application state and one for editing options stored in the project file.

Of course, this task is very simple and consists only of a few lines of code, but still it is very convenient to have such a combinator if in the application options have to be edited several times. Especially it would be nice to have such a task in a library inside the *iTask* system.

Example An example of a dialog generated using this task can be seen in Figure 5.8.

Wizards

A wizard is a sequence of steps the user has to perform to achieve a certain goal. They guide the user through a process step-by-step. This is especially user friendly for processes which are complex and not performed regularly. For instance wizards are often used for configuration purposes or to guide the user through the process of generating special documents like letters.

At each step the user has buttons to proceed or to go back to the previous step in order to change information previously entered. The last step shows a button to finish the wizard. Also the user can cancel the wizard at each point. This common behaviour is generalised using this higher-order combinator:

```
wizard :: description [WizardStep state] state → Task (Maybe state)
    | html description & iTask state & SharedVariable state
```

The wizard uses one variable shared among all steps. Its initial value is given as third parameter and its new values is given as result if the user does not cancel the wizard. The wizard gets are list of steps which are carried out sequentially and defined as:

```
:: WizardStep state = ViewOnState String [View state]
                    | CustomTask (state WizardAction → Task (state,WizardAction))

:: WizardAction = GotoNext | GotoPrevious
```

A step using the constructor `ViewOnState` provides an explanatory text, views on the shared state and buttons for going to the next or previous step and to cancel the wizard. Using views on a shared state has the advantage that no data is lost if the user goes through the steps using the buttons.

In some situation such a standard step is not sufficient. For instance for the case that one wants to verify a certain condition like checking if a path is valid or if one wants to read some value from another database. In this case a custom task can be provided which can arbitrarily interact which the user or do some other I/O. It modifies the state and returns it together with an action telling whether to go back to the previous or go on with the next step.

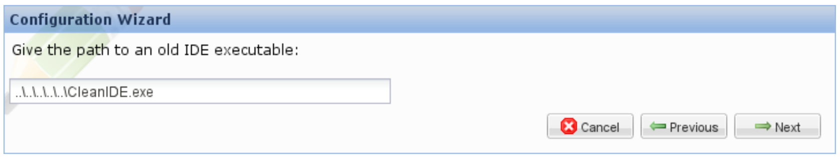
Also information about the last action is needed. One may only want to check for conditions about information if they were entered by the previous step. If the user goes back, the task may just want to skip itself by immediately returning a `GotoPrevious`.

Example In the IDE a wizard is used for generating the initial configuration, which includes the path where projects are stored and the path to the old IDE executable. Here only the two steps for entering the path to the old IDE are described.

The shared state of the wizard is of the already discussed type `IDEConfig`. First a view to edit the path is used:

```
ViewOnState "Give the path to an old IDE executable:"
   [editor { editorFrom    = λconfig        → config.oldIDEPath
           , editorTo      = λpath config   → {config & oldIDEPath = path}
           }
   ]
```

This step does not require much explanation. It just gives a view on the corresponding field of the state. As can be seen in Figure 5.13 buttons are automatically added by the wizard combinator.


Configuration Wizard

Give the path to an old IDE executable:

..\..\..\..\..\CleanIDE.exe

Figure 5.13: An Example of a Wizard Step

After this it is checked whether a file at this path is existing by this custom task:

```
CustomTask  (λconfig prevAction → case prevAction of                          1
    GotoPrevious = return (config, GotoPrevious)                              2
    GotoNext =                                                                3
                    fileExists config.oldIDEPath                              4
        >>= λok.    if ok                                                     5
                    (return (config, GotoNext))                               6
                    (       showMessageAbout "Error"    ( "’" +++ config.oldIDEPath +++   7
                                                          "’ does not exist!"  8
                                                        )                     9
                        >>| return (config, GotoPrevious)                     10
                    )                                                         11
    )                                                                         12
```

If the previous step was not the step for entering the path, the previous step was the step after this check. Therefore the previous action was to go back. In this case the path does not has to be checked again and the wizard should go to the previous step which is the step for entering the path (line 2).

Otherwise it is checked whether the file given as IDE executable exists (line 4). If this is the case it is continued with the next step immediately (line 6). In this case the user does not even notice the existence of this task. If the file does not exist an error message is shown to the user (lines 7 – 9) and the wizard goes back to the previous step such that the user can correct the path (line 10).

Summary

In this section it is shown that essential parts of an IDE like compilation and syntax highlighting can be realised. Doing compilation in a more platform independent and user friendly way is still an open issue. Also GUI patterns like wizards and option dialogs can be realised in a general way.

However to extend this to an entire *Clean* IDE comparable to the old one would still require much more work. The reason is that there are difficult issues like determining which modules belong to the project, handling environments and searching up definitions. This complex program logic is already implemented in the old IDE, but nevertheless it would require much work to get it out of the old code. Those issues have nothing to do with the GUI paradigm and are therefore left out.

Chapter 6

Evaluation

The extended *iTask* language has the intended capabilities discussed in Section 3.1. It is possible to use menus to give commands in a structured way (Section 4.1) and to let the user work on parallel tasks shown in windows (Section 4.3). Also there is an elegant way to realise views using shared data (Section 4.2). This can be used as basis for realising MDI applications (Section 4.4), which also shows that higher-order combinators can help implementing common behaviour. Also more sophisticated controls like formatted text (Section 4.5.1), source code (Section 4.5.2) and colours (Section 4.5.3) can be integrated into the *iTask* system. Finally the case studies (Chapter 5) show that it is possible to develop applications like a multi-document text editor and suggest that it would be feasible to implement a more complex application like an IDE. Here it is also shown that, like in modern commonly used languages, exceptions can be used to handle errors in a way that is separated from the actual program logic. Also repeatedly occurring GUI patterns can be generalised using customary combinators (Section 5.2.4). This idea could be used to build a collection of commonly used GUI patterns to provide programmers with powerful existing functionality.

The constructions added to the language adhere to the general design principles (Section 3.2). Nearly all new features are added as extensions to the language, which can be used optionally but do not make the language more difficult to use for more simple cases. For instance setting menus and creating shared views is done by using new tasks, which can be ignored if this functionality is not needed. There are only few changes which makes rewriting existing code necessary, but most changes makes old functionality more convenient to use and the code more readable, like an improved list of button actions which denotes the conditions for allowing certain actions in a more readable way than using two lists. Other examples of new tasks developed for this thesis which makes old functionally easier to use are tasks dealing with stored values, like automatically creating a unique identifier and setting an initial value with `createDB`.

The way how views on a shared state can be realised is declarative in the sense that only process and data are specified. Instead of monitoring the shared state and redrawing all views manually, the programmer just defines a relation between the shared state and the view using two functions. So the programmer just specifies that there are tasks updating shared data (process) and gives a functional relationship between the shared state and the views edited by the user (data). Creating windows is not declarative in the sense that annotations are used to define the behaviour of tasks. However if those annotations are left out one gets a declarative workflow description with the same functionality. The action list of tasks and groups is also not declarative because there is a distinction between buttons and menu actions. But also here if ignoring this information those action lists are declarative because they define which actions are in principle possible in a certain context and can be triggered in a certain situation (uniformly defines by the condition) which merely describes the process. Only defining the structure of menus is clearly not declarative since it has nothing to do with the process or the processed data. For this reason it is clearly separated from the action lists. Also at least the static part of the menu structure only has to be defined once at the beginning of an application. Therefore the part of the workflow not dealing with process or data is kept minimal.

All new features do not destroy the characteristic of everything being a task. Menus are just another way of generating actions and do not change the structure of the language essentially. Windows are modelled as a group of tasks running in parallel. Since the possibilities how tasks can influence each other are very limited, this has the drawback that for instance it is not possible to easily set the stacking order of windows or to close a certain one. It is only possible to dynamically add new tasks and stop all tasks in a group. Nested groups can be a way to close only a subset of the tasks running in parallel. Still no process running in a parent group can close such a group, the tasks doing this has to run in the group which is about to be closed. The possibility to focus a window was added because it is an essential operation but actually it does not fit well into the structure of a group. Also group actions are special in the sense that this is the only construction of the language where tasks are not started because another tasks finishes. Group actions behave similar as callback functions in the sense that they have an effect in the case some menu event occurs. But still the effect of them is restricted to adding new tasks or stopping the group they are attached to and cannot arbitrarily interact with other tasks. Also shared data extends the way tasks running in parallel can interact and makes it therefore more difficult to reason about workflows. Still the interaction is restricted and well-defined. Tasks do not directly influence or are even aware of other tasks using the same state, but only work on a data model.

Programmer are therefore forced to use views on a shared state to implement applications with interaction between different windows. This avoids dependencies between different tasks. For instance a dialog for replacing all occurrences of a string inside a text, does not change the text area inside another window, but works on the shared state including the text. So this task will work even if there is no editor showing the text at the moment or the representation of the text is changed later. This also has the advantage that the programmer does not have to deal with many different references to other components of the user interface like in other libraries. Most of the time one reference to a shared state or a part of it is sufficient.

Everything being a task also has other advantages. The entire state is stored on and synchronised with the server. It is possible to pause the work at any moment and continue working on another computer. Also the entire application or parts of it can be assigned to other users and have a priority and a deadline. This means that realising multi-user applications is essentially not more difficult than implementing a single-user one. Assigning the tasks working on the same data-model to different users tuns a single into a multi-user application. Of course, in practise additional measures might have to be taken to prevent conflicts if the automatic merging mechanism is not sufficient.

Additionally an abstract semantics can be given to workflows [16]. Having a language with defined semantics has the advantage that this might help to reason about the language in general and about programs written in this language in particular. Still there are tasks having side-effects for which it would be difficult to define a formal semantics, like reading and writing external files. But such side-effects never change the task structure and have no influence on tasks not accessing those files. Also shared variables behave like global variables which makes reasoning about them difficult. But still for a restricted but functional set of the language a semantics can be defined and help to reason about some properties of (parts of) applications written in the *iTask* language.

Summarised implementing applications using *iTask* workflows gives the programmer a highly abstract and declarative way of working. Programmers do not have to deal with time consuming issues like putting a complex user interface together but can concentrate on modelling the flow of data and relationships between different views on this data. We expect a low learning curve for implementing application in this way, because there are only few concepts which are powerful enough to support building complex applications. Also this way of programming is suited for rapid prototyping techniques. Changes can instantaneously be made and shown to a customer. Another advantage of the high level of abstraction is that also a high level of platform independency could be achieved because the system has much freedom of how the interface will look like and can therefore adapt it to the client currently used.

The high level of abstraction also has its drawbacks. The programmer has less control over how the user interface will look like. Also the way tasks can interact is restricted. It is impossible to freely arrange, move or modify parts of the GUI. Additionally the programmer is forced to use certain methods in the implementation, like the usage of shared data, and has therefore less freedom.

Chapter 7

Related Work

In contrast to existing functional libraries for generating user interfaces, which deal with widgets which are composed in some way to build the user interface, the *iTask* system works on a higher level of abstraction. It deals with data and the process manipulating it. The resulting user interface is automatically generated using generic programming techniques. Another unique feature of *iTask* is that it is based on a workflow semantics, which is a very natural way of defining and separating different portions of work a user has to do on a high level.

There are functional libraries which have a more imperative taste in the sense that the user explicitly has to create user interface elements and update them if the state of the application changes. Events generated by user actions are handled using callback function like in most imperative libraries. Examples are *Object I/O* [1, 2] and *wxHaskell* [18]. The former uses *Clean*'s *uniqueness type system* [27, 4] to do I/O while the latter uses *Haskell*'s monadic I/O approach [22, 30], which is similar to the monadic *iTask* language. For storing the state of the application in *Object I/O* each process has a *program state* which can be accessed by all callback functions. Additionally a *polymorphic local state* can be defined for a set of components. Finally there are complex message passing mechanisms to realise communication between multiple *interactive processes*. In *iTask* sharing data among a subset of tasks inside a process, the entire process or between multiple processes can be done using the same concept of shared data. In *wxHaskell* the concept of *mutable values* is used which is similar to *iTask*'s shared variables. However those mutable variables cannot be used to communicate between processes and there is no possibility to automatically generate views only by giving a functional relation between the model and the view.

Haggis [12] also lets programmers create widgets explicitly but gives them a more compositional view on the user interface. Each component is treated as virtual I/O device. Components are repeatedly combined together to build up the entire application. Also a separation between the user interface and the application, which means between the representation and the actual value or interaction with the user, is made. This ensures a higher level of abstraction for the implementation of the program logic. No callback functions are used to handle events but one can wait for a message to be generated by a component. For instance mutable variables are treated that way. One can for example wait for a variable to change. Also buttons generate a message if they are clicked. Concurrency is used to make it possible to compose parts of the user interface waiting for messages at the same time.

A more functional approach of defining user interfaces is *Fudgets* [6, 7]. Here *fudgets*, which are *stream processors* and pass messages, are hierarchically combined to build up the application. There are no mutable variables. Sharing data has to be realised by routing messages between components.

Fruit [9, 10] emphasises being build on a formal model even more. The main building blocks here are *signals* which are continuous time-varying values and *signal transformers* using pure functions for mapping signals to other signals. A GUI application is modelled as a signal transformer from a signal including all user inputs to a signal representing a picture. This purely functional approach makes it possible to define a complete semantics of the language, in contrast to *iTask* where tasks can have arbitrary side-effects. However the cost of the formal model used by *Fruit* is that it is very cumbersome to define I/O other than turning the user input into a picture. All I/O operations explicitly have to be added to the input and output signal.

An example where generic programming techniques are used for generating forms is the *iData toolkit* [24]. It supports the creation of interactive web applications consisting of interconnected forms. A mechanism for providing views similar to the approach discussed in this thesis is used. It has been shown that a complex application like a *conference management system* which also uses destructively updated shared data can be realised using this approach [23]. However the *iData toolkit* is not based on workflow semantics. In contrast to *iTask* where the system automatically keeps track of the control flow, with *iData* the programmer has to keep track of the application state manually.

Two more recent approaches for defining web applications that are also based on functional languages are *Links* [8] and *Hop* [26]. As *iData* and the *iTask* system their goal is to implement web applications using a single framework, instead of using different languages for the client, the program logic on the server and accessing the database. For *iTask* similar capabilities for client-side processing are available using client-side interpreter technology [13]. *Links* and *Hop* both do not work on the same level of abstraction as *iTask* but let the programmer deal with *HTML* code directly.

Chapter 8

Conclusions & Future Work

The *iTask* system has been extended to deal with essential features needed for implementing modern user interfaces. It is possible to structure commands using menus, to organise work in different windows and to modify a shared data model by different parallel tasks. Also it is shown that specialised types can be added to allow for abstract representations of more complex user interface components. Further we showed that it is possible to build complex office-like applications using the extended workflow language. The extended system is still based on the semantics of workflows and stays abstract and declarative.

The proposed paradigm for implementing graphical user interfaces has a higher level of abstraction than existing solutions with the price that it gives the programmer less control over how the user interface looks like. We think that the paradigm is highly suited for using rapid prototyping techniques and has a low learning curve, because it is based only on few core concepts. Still it is very powerful because all the power of functional programming can be used. Being embedded in a workflow system provides features like dealing with multiple users for free. Finally the program logic is based on a language for which abstract formal semantics can be defined, which might help reasoning about applications.

One direction for future work is to do more case studies. The work of this project could be continued and an entire IDE could be implemented to show that this complex kind of application can be realised. Also other kinds of application, for instance programs allowing the user to edit large amounts of data like spreadsheet applications, are interesting candidates to test and possibly improve the system. An interesting question for multi-user applications is for which situations the merging algorithm is good enough and when the programmer manually has to avoid conflicts. Maybe also more sophisticated mechanism for dealing with conflicts automatically might be needed. Adding a versioning system storing the entire history of the state would also be interesting in this context.

It has been shown how higher-order tasks can be used to generalise common behaviour. It would be very helpful if programmers were already equipped with a collection of solutions for common problems. Research in this direction first would have to identify common GUI patterns and then show how to realise them in a generalised way.

At the moment there is only one client designed for running in a web browser, but in principle the design of the system is suited for supporting several very different clients. A question related to applications is if it is possible to write a complex application, which itself is not aware on which client it is running at the moment, but can be used for instance in a web browser, on a smart-phone and on a client usable for blind persons. An important point here is to evaluate the usability compared to applications designed especially for those clients.

Although this project does not deal with layouts for good reasons in some situations it is desirable to influence it in some way. The question here is to what extent this is possible without destroying the advantages of generically generated forms.

A last area of research would be to define a formal semantics for the extended language. This could possibly lead to the insight that some features should be realised in an even more declarative way. The hope is that this can help reasoning about and testing an application implemented using the *iTask* WDL.

Bibliography

[1] Peter Achten and Marinus J. Plasmeijer. Interactive functional objects in Clean. In *Implementation of Functional Languages*, pages 304–321, 1997.

[2] Peter Achten and Martin Wierich. A tutorial to the Clean Object I/O library - version 1.2. `ftp://ftp.cs.ru.nl/pub/Clean/supported/ObjectIO.1.2/doc/tutorial.pdf`, February 2000.

[3] Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications.* PhD thesis, Radboud University Nijmegen, 2005.

[4] Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 189–206, London, UK, 1995. Springer-Verlag.

[5] 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.

[6] 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.

[7] Magnus Carlsson and Thomas Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces.* PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden, March 1998.

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects '06*, volume 4709, CWI, Amsterdam, The Netherlands, November 2006. Springer-Verlag.

[9] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.

[10] Antony Alexander Courtney. *Modeling user interfaces in a functional language.* PhD thesis, Yale University, New Haven, CT, USA, 2004. Director-Hudak, Paul.

[11] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[12] Sigbjørn 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.

[13] Jan Martin Jansen. *Functional Web Applications – Implementation and Use of Client Side Interpreters.* PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, July 2010. ISBN 978-90-9025436-4.

[14] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. iEditors: Extending iTask with interactive plug-ins. In S-B Scholz, editor, *Proceedings Implementation and Application of Functional Languages, 20th International Symposium, IFL 2008*, pages 170–186, Hatfield, Hertfordshire, UK, 10-12 September 2008. University of Hertfordshire, Technical Report No. 474.

[15] 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 Engineering Research and Practice '08*, pages 231–237, Las Vegas, NV, USA, 14-17, July 2008. CSREA Press.

[16] Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In *Scholz, S.-B. (ed.), IFL'08 : Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, pages 53–64. Hertfordshire, UK : University of Hertfordshire, September 2008.

[17] 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.

[18] 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.

[19] Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for end-users. In *Proceedings 21st Symposium on Implementation and Application of Functional Languages*, September 2009.

[20] Steffen Michels. Asynchronous update of webforms using generic programming techniques. `http://www.steffen-michels.de/articles/rdr2.pdf`, 2009. Written for the course "R&D: Research 2".

[21] Steffen Michels. Modeling a software development process using iTasks. `http://www.steffen-michels.de/articles/rdr3.pdf`, 2010. Written for the course "R&D: Research 3".

[22] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[23] Rinus Plasmeijer and Peter Achten. A conference management system based on the iData toolkit. In *Implementation and application of functional languages : 18th international symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006 ; revised selected papers*, pages 108–125. Springer Verlag, 2006.

[24] Rinus Plasmeijer and Peter Achten. iData for the world wide web - programming interconnected web forms. In *In Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), volume 3945 of LNCS*, pages 24–26. Springer Verlag, 2006.

[25] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to iTasks: Defining interactive work flows for the web. In *Central European Functional Programming School, Revised Selected Lectures, CEFP 2007*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, June 23-30 2007. Springer.

[26] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the 11th symposium on Object-oriented programming systems, languages, and applications '06*, pages 975–985, Portland, Oregon, USA, October 2006.

[27] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Proceedings of the International Workshop on Graph Transformations in Computer Science*, pages 358–379, London, UK, 1994. Springer-Verlag.

[28] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[29] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers, volume 2670 of LNCS*, pages 101–117. Springer, 2002.

[30] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

Appendix A

iTask Language Overview

In this appendix an overview of the *iTask* WDL, without the extensions added in this thesis, is given. This includes basic tasks (Section A.1), task combinators (Section A.2) and the common domain of specialised types (Section A.3). This overview is not complete, but gives an impression of the most commonly used language constructions which might be relevant for implementing GUI applications.

To increase readability context restriction are left out in this chapter. If not defined otherwise, types represented by single-letter lower-case variables (`a`, `b`, ...) have the restriction `iTask`. The variables `question`, `message` and `instruction` have restriction `html`.

A.1 Basic Tasks

Basic tasks are the smallest amount of work that can be done. There are tasks for interacting with the user, for storing data or performing other kinds of I/O.

A.1.1 Interaction Tasks

Interaction tasks are tasks requiring user interaction. First there are tasks to simply show information to the user:

Shows a message to the user. The user can end the task after reading the message.
```
showMessage           :: message              → Task Void
```
Shows a message to the user. The user cannot end the task after reading the message.
```
showStickyMessage     :: message              → Task Void
```
Shows an instruction to the user. The user can dismiss the instruction.
```
showInstruction       :: String instruction   → Task Void
```

The task `showStickyMessage` never finishes. Therefore is should only be used in parallel with other tasks, such that the task finishes because of the parallel construction it is part of does so.

Then there are tasks letting the user giving information to the system, like entering data or making choices:

Asks the user to enter information.
```
enterInformation      :: question             → Task a
```
Asks the user to update predefined information.
```
updateInformation     :: question a           → Task a
```
Asks the user to confirm or decline a question.
```
requestConfirmation   :: question             → Task Bool
```

Asks the user to select one item from a list of options.
```
enterChoice            :: question [a]         → Task a
```
Asks the user to select one item from a list of options with one option already pre-selected.
```
updateChoice           :: question [a] Int     → Task a
```
Asks the user to select one or more items from a list of options.
```
enterMultipleChoice    :: question [a]         → Task [a]
```
Asks the user to select one or more items from a list of options with some options already pre-selected.
```
updateMultipleChoice   :: question [a] [Int]   → Task [a]
```

For all interaction tasks there is also a version giving additional context information to the user. The context is of arbitrary type. A *HTML* representation of it is generically generated, as explained in Section 2.3.3.

```
enterInformationAbout       :: question b             → Task a
updateInformationAbout      :: question b a           → Task a
requestConfirmationAbout    :: question a             → Task Bool
enterChoiceAbout            :: question b [a]         → Task a
updateChoiceAbout           :: question b [a] Int     → Task a
enterMultipleChoiceAbout    :: question b [a]         → Task [a]
updateMultipleChoiceAbout   :: question b [a] [Int]   → Task [a]
showMessageAbout            :: message a              → Task Void
showStickyMessageAbout      :: message a              → Task Void
showInstructionAbout        :: String instruction b   → Task Void
```

A.1.2 Store Tasks

There are tasks to store a single value and also tasks for storing a collection of multiple values in a type-safe way.

Storing a Single Value

The following tasks are used to handle a database of one value of arbitrary type:

Creates a database reference from a string uniquely identifying the database.
```
mkDBid  :: String      → (DBid a)
```
Reads from the database. Returns a default value if no value is stored.
```
readDB  :: (DBid a)    → Task a
```
Writes to the database.
```
writeDB :: (DBid a) a  → Task a
```

A reference to such a database is a string with added type information:

```
::DBid a :== String
```

Storing a Collection of Values

To store a collection of multiple values of a type an instance of `DB` has to be provided for this type:

```
class DB a where
    databaseId  :: DBid [a]
    getItemId   :: a → DBRef a
    setItemId   :: (DBRef a) a → a

:: DBRef a = DBRef Int
```

First the database has a unique identifier (`databaseId`). Then in each item an identifier, which is an integer with type information (`DBRef`), has to be stored. The programmer has to provide a function for retrieving this identifier from a given value (`getItemId`). Finally a function for setting the identifier (`setItemId`) has to be provided.

Once this is defined for a type the following tasks can be used to access the database of multiple values of this type:

Determines if two values have the same identifier.
```
eqItemId      :: a a         → Bool
```
Reads the entire collection of stored values.
```
dbReadAll     ::               Task [a]
```
Writes the entire collection of stored values.
```
dbWriteAll    :: [a]         → Task Void
```
Creates a new entry in the database.
```
dbCreateItem  :: a           → Task a
```
Reads the entry with given identifier.
```
dbReadItem    :: (DBRef a)   → Task (Maybe a)
```
Updates an entry.
```
dbUpdateItem  :: a           → Task a
```
Deletes an entry from the collection.
```
dbDeleteItem  :: (DBRef a)   → Task Void
```

A.1.3 Date & Time Tasks

Tasks for retrieving the current date and time are simple examples of tasks retrieving information from outside of the *iTask* system and therefore performing I/O with the operating system:

Returns the current time.
```
getCurrentTime      :: Task Time
```
Returns the current date.
```
getCurrentDate      :: Task Date
```
Returns the current date and time.
```
getCurrentDateTime  :: Task DateTime
```

The types `Time`, `Date` and `DateTime` are included in the common domain which is discussed in Section A.3.

A workflow can also be paused for a given amount of time:

The task completes at the specified time.
```
waitForTime        :: Time → Task Void
```
The task completes at the specified date.
```
waitForDate        :: Date → Task Void
```
The task completes after the specified amount of time has passed since the creation of the task.
```
waitForTimer       :: Time → Task Void
```

A.1.4 Import Tasks

Import tasks can be used to import text files, comma-separated values (CSV) files and arbitrary files as documents. For paths platform dependent strings are used here:

Imports a text file.
```
importTextFile      :: String                → Task String
```

Imports a CSV file. A comma (',') is used as field separator, double quotes ('"') may be used to enclose fields and the escape character is backslash ('\').
```
importCSVFile       :: String                → Task [[String]]
```

Imports a CSV file with custom field separator, quote and escape character.
```
importCSVFileWith   :: Char Char Char String → Task [[String]]
```

Imports an arbitrary file as document.
```
importDocument      :: String                → Task Document
```

A.1.5 System Tasks

This is a small collection of tasks which can be used for interacting with the *iTask* engine directly:

Returns the user currently logged in the iTask system.
```
getCurrentUser       ::  Task User
```

Retrieves the process id of the current process.
```
getCurrentProcessId ::  Task ProcessId
```

Gets the user the current task is assigned to.
```
getContextWorker    :: Task User
```

Gets the user the current task is managed by.
```
getContextManager   :: Task User
```

Compute a default value.
```
getDefaultValue     :: Task a
```

Gets a random integer.
```
getRandomInt        :: Task Int
```

There are also many more tasks for interacting with the *iTask* engine directly. For instance the process, data and user databases can be accessed. This makes it possible to implement tasks for managing the *iTask* system using the *iTask* WDL. For instance the root user gets to see workflows for adding or removing users. However this is not relevant for the topic of this thesis and is therefore not discussed in detail.

Another interesting task is about changing running workflows which is also outside of the scope of this thesis.

A.2 Combinators

Combinators are used to build complex tasks by combining more simple ones. The result of a combinator is a task again. In this way arbitrary complex workflows can be built.

A.2.1 Sequence Combinators

There are a some combinators for defining a sequence of tasks.

Combines two tasks sequentially. The first task is executed first. When it is finished the second task is executed with the result of the first task as parameter.
```
(>>=) infixl 1  :: (Task a) (a → Task b)          → Task b
```

Combines two tasks sequentially like `>>=`*, but the result of the first task is ignored.*
```
(>>|) infixl 1  :: (Task a) (Task b)              → Task b
```

Executes the list of tasks one after another. The result is a list of all single tasks' results.

```
sequence        :: String [Task a]                      → Task [a]
```

Combines two tasks sequentially like `>>=`*, but both have to return an optional values. The second task is only executed if the first one returns a value.*

```
(>>?) infixl 1  :: (Task (Maybe a)) (a → Task (Maybe b))   → Task (Maybe b)
```

A.2.2 Repetition Combinators

Those combinators repeat a task until some condition holds.

Repeats a task infinitely. The combined task never finishes.

```
forever         :: (Task a)                             → Task a
```

Repeats a task until a given predicate on its result holds.

```
(<!)  infixl 6  :: (Task a)  (a → .Bool)                → Task a
```

Repeats a task until a given predicate on its result holds. If the predicate does not hold additionally an errors message is shown.

```
(<|)  infixl 6  :: (Task a)  (a → (Bool, [HtmlTag]))    → Task a
```

Iterates a task until a given predicate holds. The repetition is initialized using the third parameter as initial value and continued using the first. The output of each cycle serves as input for the next one. As soon as the predicate holds for a produced value it is returned as result.

```
repeatTask      :: (a → Task a) (a → Bool) a            → Task a
```

Iterates a task until a given predicate holds. The repetition is initialized using the first parameter and continued using the second. The output of each cycle serves as input for the next one. As soon as the predicate holds for a produced value it is returned as result.

```
iterateUntil    :: (Task a) (a → Task a) (a → .Bool)    → Task a
```

A.2.3 Static Group Combinators

The core group combinator making it possible to dynamically add tasks is discussed in Section 4.3.5. There is also a collection of static combinators without the possibility to add tasks dynamically. They are derived from the core combinator.

Groups two tasks of the same type. The result of the task finishing first is taken as result.

```
(-||-) infixr 3 :: (Task a) (Task a)     → Task a
```

Groups two tasks of different type. The result of the task finishing first is taken as result.

```
eitherTask      :: (Task a) (Task b)     → Task (Either a b)
```

Groups two tasks of different type. The group finishes if both tasks are finished. The group's result is the result of the right task.

```
(||-)  infixr 3 :: (Task a) (Task b)     → Task b
```

Groups two tasks of different type. The group finishes if both tasks are finished. The group's result is the result of the left task.

```
(-||)  infixl 3 :: (Task a) (Task b)     → Task a
```

Groups two tasks of different type. The group finishes if both tasks are finished. The group's result is a tuple including the results of both tasks.

```
(-&&-) infixr 4 :: (Task a) (Task b)     → Task (a,b)
```

Groups an arbitrary number of tasks of the same type. The result of the task finishing first is taken as result.

```
anyTask         :: [Task a]              → Task a
```

Groups an arbitrary number of tasks of the same type. The group finishes if all tasks are finished. The group's result is a list including the results of all tasks.

```
allTasks        :: [Task a]              → Task [a]
```

A.2.4 Tuning Combinators

Tuning combinators are used to fine tune tasks or workflows. For this the following combinators are used to apply values of class `tune` to an arbitrary task:

```
(<<@) infixl 2 :: (Task a) b  → Task a | tune b
(@>>) infixr 2 :: b (Task a)  → Task a | tune b

class tune b :: b (Task a) → Task a
```

To give an impression a list of types for which an instance of `tune` is defined is given:

```
instance tune  ManagerProperties   // Set initial properties
instance tune  User                // Set initial worker
instance tune  (Subject s)         // Set initial subject
instance tune  TaskPriority        // Set initial priority
instance tune  Timestamp           // Set initial deadline
```

The types are not discussed in details here. They are used to tune workflows. They only have an effect if applied to the workflow's top-level task.

A.2.5 Lifting Combinators

To do I/O in *Clean* the uniqueness type system [27, 4] is used to manipulate a unique value of type `World`. This makes it possible to do arbitrary I/O in a pure functional setting. The *iTask* WDL is based on a monadic combinator language which hides this unique state.

It is possible to use an arbitrary I/O function working on the unique world by lifting it to the task domain:

```
appWorld :: (*World → *World)       → Task Void
accWorld :: (*World → *(a,*World)) → Task a       | iTask a
```

There are two variants. The first one only changes the value of the world. The second one additionally returns a value.

A.3 Common Domain

The common domain is a collection of specialised types. Actually it extends the set of basic types with commonly used, more sophisticated ones. For most of them a special widget is used to edit them.

First there are string with a special meaning:

```
:: EmailAddress = EmailAddress String
:: URL          = URL String
:: PhoneNr      = PhoneNr String
:: Note         = Note String
:: Password     = Password String
```

This adds information about the intended use of those strings in the workflow description. Also the system can visualise and validate this data in a different way. For instance a `Note` is represented by a multi-line textarea instead of a single-line textfield. For a `Password` the GUI does not print the characters in plain text.

The following types are used to represent dates, times and a combination of both:

```
:: Date =   { day   :: Int
            , mon   :: Int
            , year  :: Int
            }
```

```
:: Time =   { hour  :: Int
            , min   :: Int
            , sec   :: Int
            }

:: DateTime = DateTime Date Time
```

How a `Date` is represented in the GUI is depicted in Figure 2.1.

An amount of money can be represented by this type which stores the currency together with the amount in cents:

```
:: Currency = EUR Int
            | GBP Int
            | USD Int
            | JPY Int
```

Finally buttons can be added to the GUI. It is not possible to trigger actions using such buttons. The user can only press and unpress them. The value of type `ButtonState` indicates the current state:

```
:: FormButton = { label :: String
                , icon  :: String
                , state :: ButtonState
                }

:: ButtonState = NotPressed | Pressed
```

Actually this type does not fit neatly into the concept since it changes the representation of what is actually a boolean value. So it does not really deal with the data processed by the workflow.