

An Introduction to Task Oriented Programming

Peter Achten, Pieter Koopman, and Rinus Plasmeijer^(✉)

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, Nijmegen, The Netherlands
`{p.achten,pieter,rinus}@cs.ru.nl`

Abstract. Task Oriented Programming (or shortly TOP) is a new programming paradigm. It is used for developing applications where human beings closely collaborate on the internet to accomplish a common goal. The tasks that need to be done to achieve this goal are described on a very high level of abstraction. This means that one does not need to worry about the technical realization to make the collaboration possible. The technical realization is generated fully automatically from the abstract description. TOP can therefore be seen as a model driven approach. The tasks described form a model from which the technical realization is generated.

This paper describes the *iTask* system which supports TOP as an Embedded Domain Specific Language (EDSL). The host language is the pure and lazy functional language *Clean*.

Based on the high level description of the tasks to do, the *iTask* system generates a web-service. This web-service offers a web interface to the end-users for doing their work, it coordinates the tasks being described, and it provides the end-users with up-to-date information about the status of the tasks being performed by others.

Tasks are typed, every task processes a value of a particular type. Tasks can be calculated dynamically. Tasks can be higher order: the result of a task may be a newly generated task which can be passed around and be assigned to some other worker later on. Tasks can be anything. Also the management of tasks can be expressed as a task. For example, commonly there will be many tasks assigned to someone. A task, predefined in the library for convenience, offers the tasks to do to the end-user much like an email application offers an interface to handle emails. This enables the end-user to freely choose which tasks to work on. However, one can define other ways for managing tasks.

A new aspect of the system is that tasks have become reactive: a task does not deliver one value when the task is done, but, while the work takes place, it constantly produces updated versions of the task value reflecting the progress of the work taken place. This current task value can be observed by others and may influence the things others can see or do.

1 The Task-Oriented Programming Paradigm

1.1 Introduction

These lecture notes are about *Task-Oriented Programming (TOP)*. TOP is a programming paradigm that has been developed to address the challenges software developers face when creating *interactive*, *distributed*, *multi-user* applications. *Interactive* applications provide their users with an optimal experience and usage of the application. Programming interactive components in an application is challenging because it requires deep understanding of GUI toolkits. Additionally, the program structure (for instance widget-based event handling with callback functions and state management) makes it hard to figure out what the application is doing. *Distributed* applications spread their computational activities on arbitrarily many computing devices, such as desktop computers, notebooks, tablets, smart phones, each running one operating system or another. The challenges that you face concern programming the operating systems of each device, keeping track of the distributed computations in order to coordinate these tasks correctly and effectively, and executing the required communication protocols. *Multi-user* applications serve users who work together in order to achieve common goals. A simple example of a common goal could be to write, or design something together. In this area the challenges concern keeping track of users, aiding them with their work, and making sure that they do not get in each other's way. More challenging examples are modern health care institutes, multi-national companies, command and control systems, where thousands of people do a job in collaboration with many others, and ICT plays an important role to connect the activities. We have written these lecture notes to show how contemporary, state-of-art programming language concepts can be used to rise to the challenge of creating applications in a structured way, using a carefully balanced mixture of the novel concept of *tasks* with the proven concepts of types, type systems, functional and type-indexed programming.

The Internet forms a natural habitat for the kind of applications that TOP has been designed for (Fig. 1) because its architecture makes TOP applications available on a wide range of equipment, such as desktop computers, notebooks, smart phones, and tablets. In addition, it is very natural for a TOP application to serve more than a single user. TOP applications can deploy web services, or provide these themselves. Under the hood the application uses a clients-server architecture. The client sides implement the front-end components of the application, running in web browsers or as apps on smart phones and tablets. The server side runs as a web service and basically implements the back-end coarse grain coordination and synchronization of the front-end components. During the operations, it can use other web services, rely on sensor data, use remote procedure calls, and synchronize data 'in the cloud' or back-end database systems.

Unless one can manage to separate the concerns in a well organized manner, programming this kind of applications is a white-water canoeing experience in which there is a myriad of rapids to be taken in the form of design issues, implementation details, operating system limitations, and environment requirements.

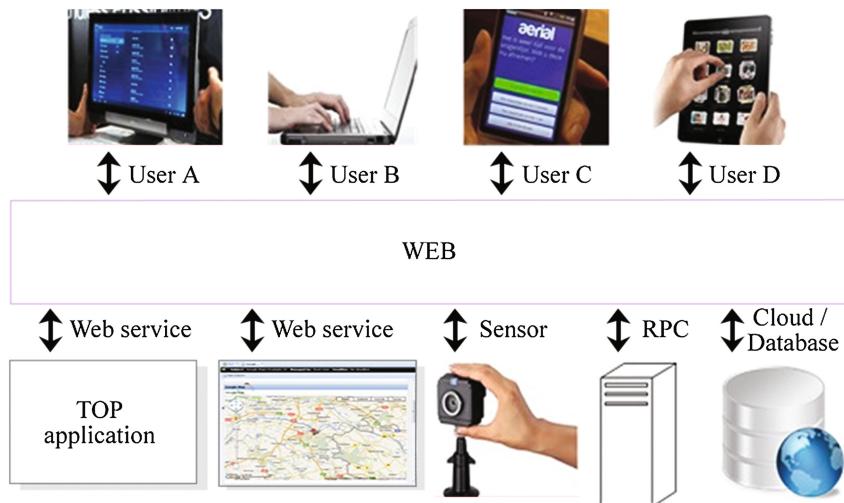


Fig. 1. The Internet habitat of TOP applications

TOP steers the programmer away from these rapids and guides to placid waters. It forces the programmer to think of the work that the intended processors (humans and computers) of your applications are required to do, as well as the structure of the information that is required to coordinate this work properly. TOP offers a *declarative* style of programming in which *what* takes precedence over *how*. A TOP program relates to work in a similar way as René Magritte's well known painting of a pipe relates to a real pipe (Fig. 2). In a TOP program *tasks* are specifications of *what* work must be performed by the users and computing machinery of an application. *How* the specification is executed is the concern of the TOP language implementation, taking the rapids. For instance, the task of obtaining information from users should require only a data model of the



Fig. 2. *La Trahison des Images* (*The Treachery of Images*), 1928–1929, by René Magritte – “This is not a pipe”

information; the TOP language implementation of this task handles the entire user interface management. Similarly, the task of coordinating tasks should require only the data model of the processed data; the TOP language implementation of this task handles all coordination and communication issues. Often data models need to be transformed from one format to another. It should be sufficient to specify the computation that is restricted on the proper domain and range and trust that the TOP language implementation knows when to invoke these computations on the proper data values without unexpected side-effects.

It should be clear that *types* play a pivotal role in TOP: they are used for modeling information and specify the domains and ranges of computations; the TOP language implementation uses them to generate and handle user interfaces and coordinate work implementations.

The TOP language that we have developed and use in this paper is *iTask*. Figure 3 gives a bird's-eye view of the main components of the *iTask* language. *iTask* is a *combinator language*. Combinator languages emphasize the use of *combinators* to construct programs. A combinator is a named *programming pattern* that in a very precise way states how a new piece of program is assembled from smaller pieces of programs. *iTask* is also an example of an *embedded language*. Embedded languages borrow key language aspects from an existing language, the *host language*. In this way they receive the benefits of known language constructs and, more importantly, do not have to re-invent the wheel. In the case of *iTask* the host language is the purely functional programming language *Clean*. Consequently, the combinators are expressed as functions, and the model types can be expressed with the rich type language of *Clean*. *iTask* extends its host language with a library that implements all type-indexed algorithms, web client handling, server side handling, and much more.

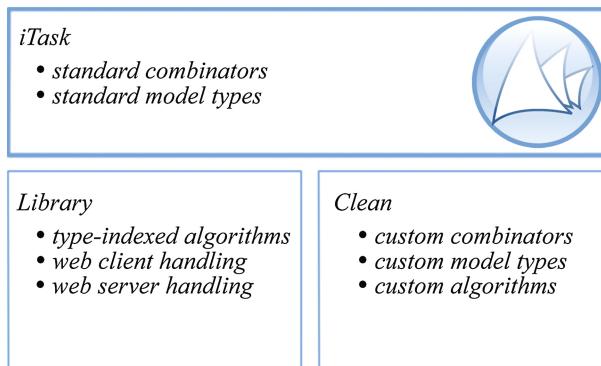


Fig. 3. The *iTask* language is embedded in the functional language *Clean*

TOP applications developed in *iTask* appear as a web service to the rest of the world and the *iTask* clients that connect with your application. Figure 4 shows how *iTask* applications fit in the Internet habitat. An *iTask* application acts as a

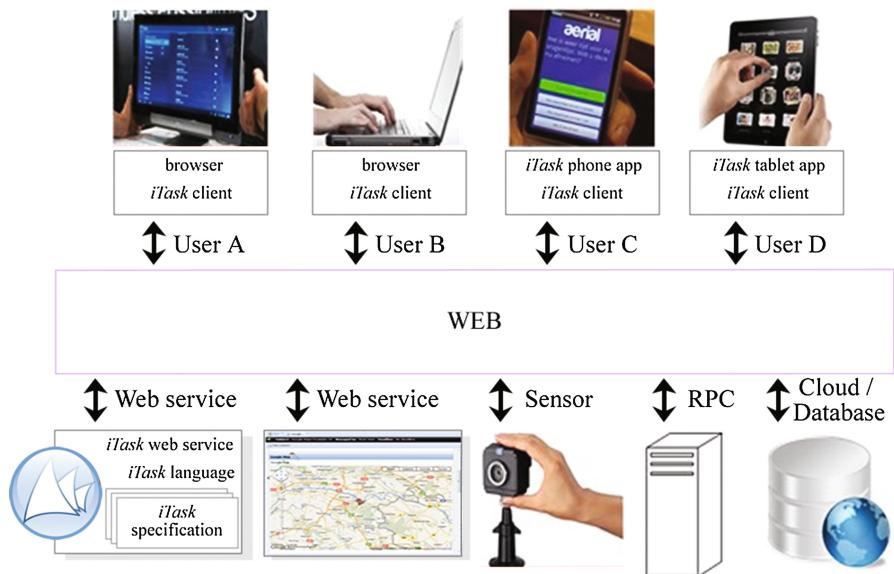


Fig. 4. *iTask* applications are Internet species

web service that can be used by other Internet applications. Users connect with the application via a standard web browser on a personal computer, or an app on a smart phone or tablet.

2 TOP Programming with iTasks

In this section we briefly explain how tasks and their signatures are denoted (Sect. 2.1), and how to set up the code examples in the *iTask* system (Sect. 2.2). The syntax of the host language *Clean* is very similar to *Haskell*. In Appendix A we give a brief overview of some *Clean* specific parts that we use in these lecture notes. Both the *Clean* programming environment and the *iTask* toolkit can be obtained at wiki.clean.cs.ru.nl.

2.1 Task Signatures

A task has two components: a description of the *work* that has to be performed, and the typed interface that determines the *type* of the task values that are communicated to the environment in which the work is performed.

Tasks abstract from activities within software systems, regardless whether these are executed by computer systems or humans, how long they will take, and what resources are consumed. For instance, a task can describe the job to interview a particular person without predetermining whether this must be done with a human interviewer or via an online questionnaire. As another example,

a task can describe the job that pieces of music must be played without predetermining whether a user starts to play guitar or let some music player application randomly pick and play a digitized music file from a play list. Work abstraction is a good thing, because it allows the context in which tasks operate not to trouble themselves with the way in which tasks are implemented.

Tasks do need to have up-to-date knowledge about each other's progress. This is where the type of a task enters the picture. Commonly, tasks process information, and often the environment would like to know what the current state is of a task. Other tasks can see how things are going by inspecting the current value of a task which may change over time. The current value of the information that is processed by a task is called its *task value*. The task value may change over time, but its type remains the same. For instance, during the interviewing task above, the task value might be the notes that are made by the human interviewer or the current state of the online questionnaire that is filled in by the interviewee. During the music playing task, the task value might be information about the current song that is played or the current recording of the digitized music that is played. In both examples, the task values change during the task, but their type remains constant.

Tasks are typed in the following way: if the type of the task value is T , then the corresponding task has type (Task T). So, a task with name t and task value type T has signature $t :: \text{Task } T$ (see signatures, page 240).

To describe what a task is about you need additional information. In these lecture notes we describe this in a functional style. A task is represented by a function which obtains the additional information via the function arguments. If we require n arguments of consecutive types $A_1 \dots A_n$ to describe a named task t of type (Task T), then this is specified by a *task function* with signature $t :: A_1 \dots A_n \rightarrow \text{Task } T$. Note that if $n = 0$, then t is the constant function that defines a task right away. Such a function has signature $t :: \text{Task } T$.

To give you a feeling how to read and write signatures of tasks, we show a few examples.

- A user who is writing a piece of text is performing a task with a task value that reflects the current content of that text. Let's name this task `write_text`. The text content can be modeled in different ways. As an example, you can choose a basic string representation, or a structure representation of the text that includes mark-up information, or a *pdf* document that tells exactly how the document should be rendered. Let us defer the decision how to represent the text exactly, and introduce some opaque type `Text`. We can define the signature of the task to write a piece of text as follows:

```
write_text :: Task Text
```

Observe that this task requires no further arguments.

- A task to interview a certain person, identified by a value of type `User`, may result in a `Questionnaire` document. Let's name this task `interview`. If we ignore the details of the user identification and questionnaire, then the signature of this task is:

```
interview :: User -> Task Questionnaire
```

This is an example of a task function with one argument, `User`.

- A computer that sorts a list of data is performing a computational task that ultimately returns a list with the sorted data. Let us name this task `sort_data`. This task requires as argument the list of data that must be sorted. For sorting, it suffices to know that the list elements possess an ordering relation, so this task should work for any element type, indicated with a type *variable* `a`, provided that an instance of the type class `Ord` for `a` is present. The signature of the sorting task function is specified by (see overloading, page 240):

```
sort_data :: [a] -> Task [a] | Ord, iTask a
```

In a signature, the occurring type class restrictions are enumerated after the `|` separator. The type of the task value must always be an instance of the `iTask` type class. For this reason, the type class restriction `iTask` is also included for values of type `a`.

- Assume that the task would be to start some given task argument at a given point in time. Hence, when performing such a task one first needs to wait until the given moment in time has passed, and then perform the given task argument. Let us name this task `wait_to_do`.

```
wait_to_do :: Time (Task a) -> Task a | iTask a
```

`Time` is a data type that models clock time. Note that `wait_to_do` is an example of a *higher-order* task function. A higher-order task function is a task function that has at least one argument that is itself a task(function).

These examples illustrate that the functional style of programming carries over to tasks in a natural way.

2.2 Modules and Kick-Start Wrapper Functions

We set up an infrastructure for the TOP examples that are presented in these lecture notes.

The host language *Clean* is a modular language. Modules collect task definitions, data types, and functions that are logically related (see modules, page 235).

We have assembled a couple of *kickstart* wrapper functions and put them in the module `TOPKickstart` that can be imported by a TOP main module. The kickstart wrapper functions are enumerated in Fig. 5. The `one-` wrapper functions are intended for a single user and the `multi-` wrapper functions assume the existence of a set of registered users. The `-App` wrapper functions support a single application only and the `-Apps` wrapper functions provide infrastructure to manage several applications.

The corresponding `TopKickstart.dcl` module is given in Fig. 6. Note that line 2 makes the entire `iTasks api` available to your TOP programs if you import `TopKickstart` yourself. The signatures of the four kickstart wrapper functions

	one user	multi-user
single application	oneTOPApp	multiTOPApp
multiple applications	oneTOPApps	multiTOPApps

Fig. 5. The four possible kickstart wrapper functions for *iTask* examples.

```

definition module TOPKickstart
import iTasks

oneTOPApp   :: (Task a)    !*World -> *World | iTask a
multiTOPApp :: (Task a)    !*World -> *World | iTask a
oneTOPApps  :: [BoxedTask] !*World -> *World
multiTOPApps :: [BoxedTask] !*World -> *World

:: BoxedTask = E.a: BoxedTask String String (Task a) & iTask a

```

Fig. 6. The kickstart module with four wrapper functions.

are given at lines 4–7. The `-App` wrapper functions expect a single task definition as argument. The type of this task, `(Task a)`, can be anything provided that it is an instance of the `iTask` type class. The `-Apps` wrapper functions are provided with arbitrarily many tasks. In order to properly model the fact that these tasks need not have to have identical types, these are encapsulated within the `BoxedTask` type. An explanation of this type can be found in Example 2 (see algebraic types, page 241).

When developing an application in these notes, we always tell which type of applications of Fig. 5 we are creating, and thus which kickstart wrapper function of Fig. 6 is required.

3 User Interaction

Having warmed up, we start our introduction on TOP with the means to interact with the user. The *type-indexed programming* foundation of TOP plays a crucial role. The information that must be displayed or received is *modeled* using the rich type language of *Clean*. The proper interactive tasks are created by *instantiating* the existing type-indexed task functions.

3.1 Displaying Information to the User

Many text books on programming languages start with a “*Hello, world*” program, a tradition initiated in the well known C programming book by Kernighan and Ritchie [1]. We follow this tradition.

EXAMPLE 1. ‘Hello, world’ as single application for a single user

We create a main module with the name `MyHelloWorldApp`:

```

1 module MyHelloWorldApp
2 import TOPKickstart
3
4 Start :: *World -> *World
5 Start world = oneTOPApp helloWorld world
6
7 helloWorld :: Task String
8 helloWorld = viewInformation "iTasks says:" [] "hello,world"

```

iTasks says:
hello, world

Just like its host language *Clean*, for an *iTask* program the main entry point is the `Start` function. Tasks, no matter how small, change the world. This is reflected in the type of the `Start` function (line 4). We use the `oneTOPApp` kickstart wrapper function (line 5) to create a single TOP application for a single user. The single TOP application is the task named `helloWorld`. The sole purpose of this task is to display the text “`hello,world`”. Because that is a value of type `String`, the type of the `helloWorld` task is `Task String` (line 7). At execution, an output similar to the one displayed to the right of the program should be produced (see side-effects, page 239).

The `hello, world` text in Example 1 is displayed with the task function `viewInformation`. Its signature is:

```
viewInformation :: d [ViewOption m] m -> Task m | descr d & iTasks m
```

It is an overloaded function due to the type class restrictions (`| descr d & iTasks m`). This task function has three arguments:

- The first argument has type `d` and is a descriptor to inform the user what she is looking at. The `descr` type class supports several data types as instances, of which in this section we use only two: the basic type `String` and the *iTask* type `Title`, which is defined as:

`:: Title = Title String`

- In both cases, a (typically short) text is displayed to give guidance to the user. They only differ in the way they are rendered. In case of a `String` value, the text is presented along the task rendering. In case of a `Title` value, the text is displayed more prominently in a small title bar above the task rendering.
- The second argument has type `[ViewOption m]` and can be used to fine-tune the visualization of the information. However, that does not concern us right now, so we use an empty list, denoted by `[]`.
- The third argument has type `m` and is the value that must be displayed. The `iTask` type class implements the type-indexed generation of tasks from types. Of course, this is only possible if the concrete type on which you apply this function is (made) available. How this is done, is explained in Sect. 3.3. For now you can assume that you can provide `viewInformation` with values of almost any conceivable data type.

Up until Sect. 6 we develop a number of very small tasks (in Sect. 6 we introduce multi-user applications). It is convenient to collect the small tasks using the kick start wrapper function `oneTOPApps`.

EXAMPLE 2. ‘Hello, world’ as multiple applications

We create a new main module, named `MyGettingStartedApps`.

```

module MyGettingStartedApps
import TOPKickstart

Start :: *World -> *World
Start world      = oneTOPApps apps world

apps           = [ BoxedTask (get_started +++ "Hello_world")
                  "Hello,_world,in,TOP"
                  helloWorld
                ]
where
  top          = "TOP/"
  get_started = top +++ "Getting_Started/"

helloWorld :: Task String
helloWorld = viewInformation "iTasks_says:" [] "hello,_world"

```

Within any (task) function definition, local definitions can be introduced after the keyword `where`. The scope of these definitions extends to the entire right hand side of the function body (`apps` in this example). Here, this facility is used to prepare for future extensions of this example, in which the text fragments `top` and `get_started` are shared.

We use the single user, multiple application kickstart wrapper function `oneTOPApps`, and provide it with only one boxed task, `helloWorld`, in lines 7–10. Figure 7 shows how this application is rendered within a browser. The first argument of this boxed task, the text “TOP/Getting Started/Hello_world”, is used to generate the task hierarchy that is depicted in the left-top area 1 in Fig. 7 (the function `+++` concatenates its two `String` arguments). The second argument, the text “Hello,_world,in,TOP”, is depicted in the description area at the left bottom 2 when the task is selected by the user. If it is started, then it appears in the task list of the user which is the right top area 3. In order to actually work on it, it can be opened, in which case its current state is rendered in the right bottom work area 4. In the work area it can be closed and reopened at any later time without harm. Deleting it in the task list area removes it permanently. ■

3.2 Getting Information from the User

The `viewInformation` task function displays information to the user. The dual task is getting information from the user. This task is called `updateInformation`.

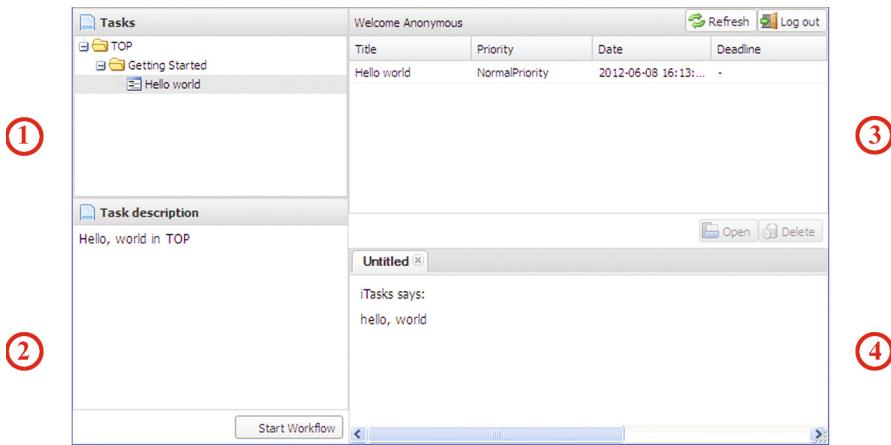


Fig. 7. ‘Hello, world’ as one task in a multi-application context.

```
updateInformation :: d [UpdateOption m m] m -> Task m | descr d & iTasks m
```

The first argument of this function has exactly the same purpose as the first argument of `viewInformation` and informs the user what she is supposed to do. The second argument is used for fine-tuning purposes, and we ignore it for the time being, and use the empty list `[]`. The third argument is the initial value that is rendered to the user in such a way that she can alter its content.

EXAMPLE 3. *What’s your name?*

We extend Example 2 with a task to ask for a user’s name.

```

module MyGettingStartedApps
import TOPKickstart

Start :: *World -> *World
Start world      = oneTOPApps apps world

apps           = [ BoxedTask (get_started +++ "Hello_world")
                  "Hello,_world,in,TOP"
                  helloWorld
                , BoxedTask (get_started +++ "Your_name?")
                  "Please_give_your_name"
                  giveName
                ]
where
  top          = "TOP/"
  get_started = top +++ "Getting_Started/"

helloWorld :: Task String
helloWorld = viewInformation "iTasks_says:" [] "hello,_world"
```

```

giveName :: Task String
giveName = updateInformation "iTasks_asks:" [] "Dr..Livingstone?"
```

20
21
22

The only modifications are lines 10–12 in which a new boxed task is included in the `apps` list, and lines 21–22 in which the new task `giveName` is defined. Figure 8 shows where the new boxed task can be selected by the user in the task hierarchy, and how the `giveName` task is rendered in the work area. ■

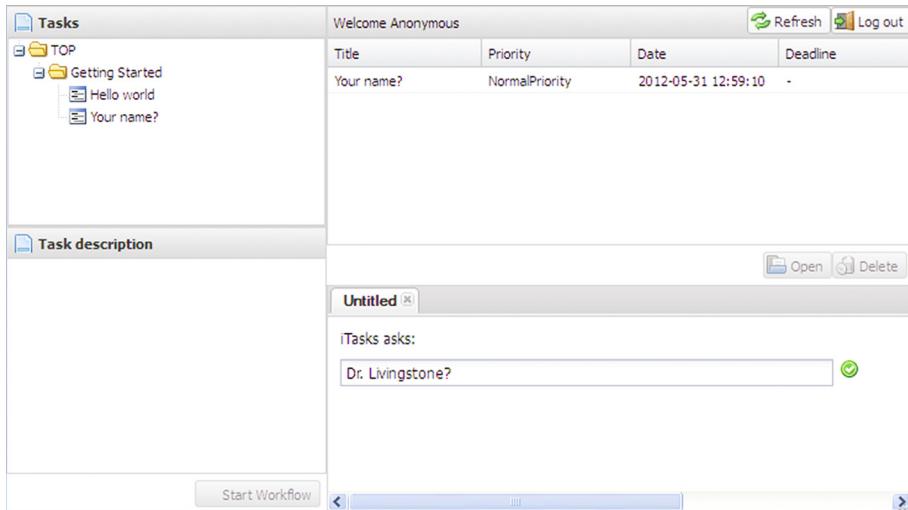


Fig. 8. Two tasks in a multi-application context.

Example 3 shows how to add a (boxed) task to the multi-application infrastructure that is created by the wrapper kickstart function `oneTOPApps`. In the remainder of these notes we restrict ourselves to discussing only the task functions that are added as boxed tasks.

3.3 Working with Data Models

Rendering and updating information by means of the functions `viewInformation` and `updateInformation` tasks works for the primitive types (booleans, integers, reals, characters, strings). Although this is useful, it is not very exciting either. Fortunately, the rendering mechanism also works for any custom defined type. The point of type-indexed programming is to encourage you to think in terms of data models and use generic functions instead of re-implementing similar tasks over and over again.

EXAMPLE 4. *Editing music tracks*

Suppose we own a collection of legally acquired digitized music and want to keep track of them in. For each piece of music we store the music storage medium (for instance *cd*, *dvd*, *blue ray*), the name of the album, name of performing artist, year of appearance, track number on album, track title, track duration, and tags. One way to model this is with the following types:

```

:: Track    = { medium :: Medium          1
              , album   :: Name           2
              , artist  :: Name           3
              , year    :: Year           4
              , track   :: TrackNr        5
              , title   :: Name           6
              , time    :: Time           7
              , tags    :: [Tag]          8
            }
:: Medium   = BlueRay | DVD | CD | MP3 | Cassette | LP | Single | Other String 10
:: Name     == String             11
:: Year     == Int               12
:: TrackNr  == Int               13
:: Tag      == String             14

```

In this definition, **Track** and **Medium** are *new* types. **Track** is a *record* type, which is a collection of *field names* (**medium**, **album**, and so on) that have types themselves (**Medium**, **Name**, and so on) (see record types, page 242). **Medium** is an example of an algebraic type which enumerates alternative data constructors (**BlueRay**, **DVD**, ..., **Other**) that may be parameterized (**Other** is parameterized with a **String**) (see algebraic types, page 241). **Name**, **Year**, **TrackNr**, and **Tag** merely introduces a *synonym type name* for another type (see synonym types, page 243). Although the type **Time** (line 7) is not a primitive type in the host language, it happens to be predefined in the *iTask* toolkit. Just like **Track**, it is a record type:

```

:: Time = { hour :: Int
          , min  :: Int
          , sec  :: Int
        }

```

but unlike **Track**, its rendering differs from the default scheme that the toolkit provides you with.

When defining a record value you need to enumerate each and every record field and provide it with a value. The order of record fields is irrelevant. Record fields are separated by a comma, and the entire enumeration is delimited by **{** and **}**. Similarly, when defining a list value you enumerate each and every element, separated by a comma and delimited by **[** and **]**. As an example, we define a value of type **Track**:

```

track = { medium = CD
         , album  = "Professor_Satchafunkilus_and_the_musterion_of_rock"

```

```

, artist = "Joe_Satriani"
, year   = 2008
, track  = 4
, title  = "Professor_Satchafunkilus"
, time   = {hour=0, min=4, sec=47}
, tags    = ["metal", "guitar", "rock", "instrumental", "guitar_hero"]
}

```

In order to make the TOP infrastructure available for a custom type t requires the declaration `derive class iTask t` in the specification. In our example, this concerns the new types `Track` and `Medium`:

```
derive class iTask Track, Medium
```

With the derived generic machinery available, `Track` values can be displayed in exactly the same way as done earlier with `String` values:

```

viewTrack :: Track -> Task Track
viewTrack x = viewInformation (Title "iTasks_says:") [] x

```

The `viewTrack` task function displays any track value that it is provided with. We can add `(viewTrack track)` to the list of boxed tasks in Example 2. Selecting this task gives the output as displayed in Fig. 9. The type-indexed algorithm recursively analyzes the structure of the value, guided by its type, and transforms the found components of its argument value into displays of those values and assembles them into one large form displaying the entire record value. Observe how the structure of the record type `Track` and the structure of the algebraic type `Medium` is rendered by the generic algorithm. Because `viewInformation` is a task that only displays its argument value, but does not alter it, the task value of `(viewTrack track)` is continuously the value `track`.

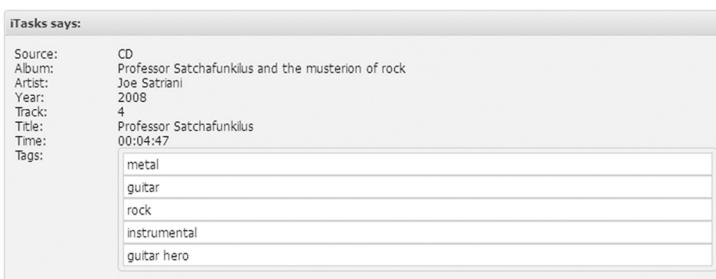


Fig. 9. The generated view of an example track.

The same principle of recursively analyzing the structure of a value is applied by `updateInformation`. In order to demonstrate this, we define this task function:

```
editTrack :: Track -> Task Track
editTrack x = updateInformation (Title "iTasks_says:") [] x
```

and add (`editTrack track`) to the collection of boxed tasks. The output is quite a different interactive element, as witnessed by Fig. 10. Instead of generating displays of component values, the algorithm now transforms them into interactive elements that can be viewed and edited by the user. The data constructors of the algebraic data type `Medium` can be selected with a menu, text entries are rendered as text input fields, numbers appear with increment and decrement facilities, and list elements can be edited, moved around in the list, deleted, and new list elements can be added. Initially, the task value of this editor task is `track`. With each user interaction, the task value is altered according to the input. It should be noted that the generated interactive elements are *type-safe*. An end-user can only type in values of appropriate type. For instance, entering the text "four" in the `track` field is rejected. In most cases it is not possible to enter illegal values. In other cases, illegal input is rejected, and replaced by the previous (legal) value.

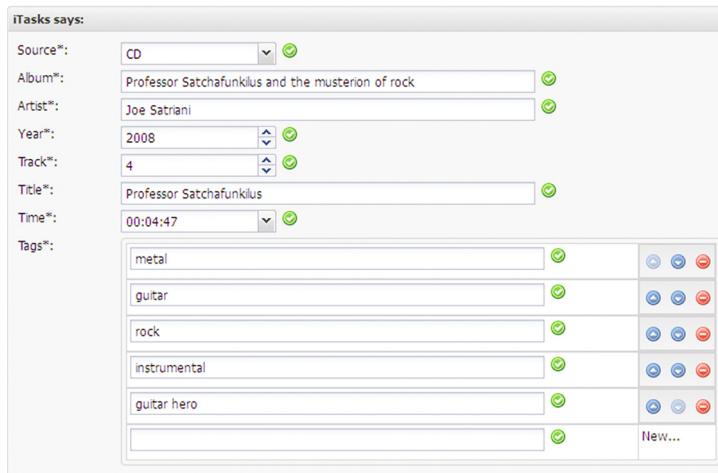


Fig. 10. The generated editor of an example track.

3.4 Working with Specialization

As mentioned in Example 4, the type `Time` is predefined in the *iTask* toolkit. Figures 9 and 10 illustrate that definitions have been provided to instruct the generic algorithm how to display and edit values of this type in a way that is *different from the default generic case*. This mechanism is called *specialization* and plays a significant role in the generic machinery that underlies task oriented programming because it allows to deviate from the general behavior.

There are many examples to be found within the *iTask* toolkit of specialized data model types. We do not wish to enumerate them all. For now we turn our attention to two dual types, `Display` and `EditTable` (Fig. 11), that interact nicely with the `viewInformation` and `updateInformation` tasks. If x is a model value of type t , then `(Display x)` is a model value of type `(Display t)` but it is rendered as a *display* of value x (hence, a user cannot alter its content). Basically, this is the same rendering as is normally provided by `viewInformation`. If x is a model value of type t , then `(EditTable x)` is a model value of type `(EditTable t)` but it is rendered as an *editor* of value x (hence, a user can alter its content). Basically, this is the same rendering as is normally provided by `updateInformation`. Using these values within data models allows one to specify very precisely what subcomponents can only be viewed by the user, and what subcomponents can be edited.

```
:: Display a = Display a
:: Editable a = Editable a

fromDisplay :: (Display a) -> a
toDisplay   :: a -> Display a

fromEditable :: (EditTable a) -> a
toEditable   :: a -> EditTable a
```

Fig. 11. Specialized model types for fine-tuning interaction.

3.5 Working with Types

Up until now, we have carefully provided the `viewInformation` and `updateInformation` task functions with concrete values. The type inference system of the host language *Clean* commonly can determine the type of the concrete value, and hence, it can be decided what instance of the type-driven algorithm should be used or generated. Commonly, a description of a task obtains sufficient information to infer the type of the model value for which either `viewInformation` or `updateInformation` need to be called.

Sometimes this is not possible, and one has to explicitly has to define the wanted type in the context such that the compiler can deduce the types for the tasks involved. Given a type, the proper instance can be determined. This is very useful in situations where it is not possible to conjure up a meaningful value of the desired type. In the case of interactive tasks, one sometimes does not want to specify an initial value but instead want to resort to a *blank* editor for values of that type, and let the user enter the proper information. This can be done with the following variant of `updateInformation` that omits the value to be altered:

```
enterInformation :: d [EnterOption m] -> Task m | descr d & iTask m
```

Just like before, the first argument is the descriptor to tell the user what is expected of her, and again we ignore the list of rendering options. The signature of this task function is actually quite odd: the `enterInformation` task function can generate a user-interface to produce a value of type `m`. This can only be done if it can be statically determined what concrete type `m` has. In this situation it becomes paramount to specify the type of the task value that is processed.

Earlier on, we used `(editTrack track)` to create an interactive task with the specific initial value `track` to allow users to alter this value. Alternatively, and more sensibly, we can specify the following interactive task:

```
inventTrack :: Task Track
inventTrack = enterInformation (Title "Invent_a_track") []
```

The generic algorithm, using the type information that a `Track` value needs to be analyzed, generates a blank interactive component (Fig. 12).

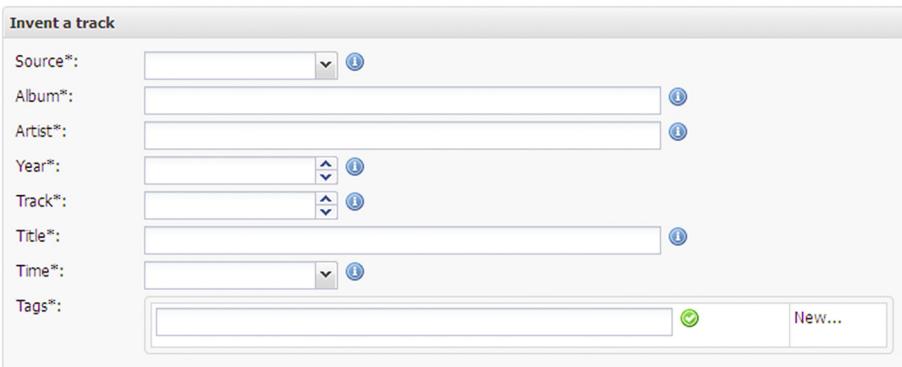


Fig. 12. The generated editor of a blank track.

The generic algorithm knows how to deal with lists, as is witnessed by creating views for *list-of-tags* in the examples. In exactly the same spirit, we can create viewers, editors, and inventors for *list-of-tracks* almost effortlessly:

```
viewTracks :: [Track] -> Task [Track]
viewTracks xs = viewInformation (Title "View_Tracks") [] xs

editTracks :: [Track] -> Task [Track]
editTracks xs = updateInformation (Title "Edit_Tracks") [] xs

inventTracks :: Task [Track]
inventTracks = enterInformation (Title "Invent_Tracks") []
```

The only thing that has changed is that the function signatures mention `[Track]` instead of `Track`. More interestingly, in case of `inventTracks`, the specified type dictates that an interactive element must be generated that handles a list of track values.

EXERCISE 1. “Hello, world”

Compile and run Example 1. Experiment with type class instances of the `descr` type class other than `String` and `Title`. Recompile and run to see the effect.

EXERCISE 2. “Hello, worlds”

Compile and run Example 2. Experiment with the arguments of the `BoxedTask` container by adding a few other tasks that display messages.

EXERCISE 3. Entering text

Add the following task to the collection of tasks of Example 2:

```
helloWorld2 :: Task String
helloWorld2 = updateInformation"iTasks_says:" □ "hello,world"
```

Recompile and run to see what the effect is.

EXERCISE 4. Your favorite collection

Design a data model for your favorite collection (for instance books, movies, friends, recipes) in a similar way as done in Example 4. Check what it looks like using `viewInformation`, `updateInformation`, and `enterInformation`.

EXERCISE 5. Editing your favorite collection

Create tasks to view and edit your favorite collection in the same way as explained on page 203 for collections of `Task` values with the functions `viewTracks`, `editTracks`, and `inventTracks`. Recompile and run to see the effect. □

4 Composition

In the previous section we have shown how a program exchanges information with the user using interactive tasks. The information is put away in the corresponding task value. Other tasks may need that information to proceed correctly. In TOP the composition of tasks is specified by means of task combinators. Combinators are functions that define how its argument tasks are combined into a new task. For reasons of readability, they are often specified as operators to allow an infix style of writing in the way we are used to when dealing with arithmetic expressions such as `+`, `-`, `*`, and `/`. In this section we introduce combinators for sequential and parallel composition, and show that this can be combined seamlessly with host language features such as choice and recursion.

4.1 Basic Tasks

The interactive task functions to view, update and enter information that are presented in Sect. 3 (`viewInformation`, `updateInformation`, and `enterInformation`) are all examples of *basic tasks*. A task(function) is *basic* if it cannot be dissected into other task(function)s. An example of a non-interactive basic task is the `return` task function (see strictness, page 244):

```
return :: !a -> Task a | iTask a
```

The sole purpose of `(return e)` is to evaluate expression e to a value x and stick to that value. It is a task which task value is always x . Despite its apparent simplistic form, the `return` task function is actually quite powerful: it allows one to introduce arbitrary computations in e to calculate a value for further processing.

EXAMPLE 5. *Sort track tags*

We can use `return` to define a task that makes sure that the tag list of a track is sorted (see record updates, page 243):

```
sortTagsOfTrack :: Track -> Task Track
sortTagsOfTrack x = return {x & tags = sort x.tags}
```

The function `sort :: [a] -> [a]` | `Ord a` is a library function of the host language that sorts a list of values, provided that the ordering operator $<$ (which is part of the `Ord` type class) is available for the element types. For `tags`, which are of primitive type `String`, the ordering operator has been defined. ■

4.2 Sequential Composition

Naïve sequential composition of tasks simply puts them in succession (see operators, page 236):

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

The combinator `>>|`, pronounced as *then*, is defined as a left-associative (`infixl`) operator of very low priority (1). In `(ta >>| tb)`, first task `ta` is evaluated. As soon as it is finished, evaluation proceeds with task `tb`. The types of the task values of `ta` and `tb` need not be identical. In addition, the type of the task value of the composite task is the same as `tb`'s task value type. Indeed, the task value of the composite task is the task value of `tb`.

As an example, we first ask the user to provide her name, and then greet her:

```
greet :: Task String
greet = giveName
      >>| helloWorld
```

We adopt the notational convention to write down the task function names below each other, as well as the task combinator functions.

The `greet` task is unsatisfactory, as it does bother the user to enter her name, but does not use that input to greet her properly. If we inspect the type of the naïve task combinator `>>|`, then we can tell that it is impossible for the second task argument to have access to the result value of the first task argument.

In most cases, follow-up tasks depend on task values produced by preceding tasks. If we express this dependency by means of a function, we obtain a non-naïve sequential combinator function, `>>=`, which is pronounced as *bind*.

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

In `(ta >>= tb)`, first task `ta` is evaluated. As soon as it is finished, its task value, `x` say, is applied to the second argument of bind, *which is now a task function instead of a simple task*, thus resulting in the computation `(tb x)`. The computation can use this value to decide what to do next, which is expressed by means of a task expression of type `(Task b)`. We can now create an improved version of the `greet` task (see lambda-abstractions, page 239):

```
greet :: Task String
greet =           giveName
  >>= \name -> viewInformation "iTTask_says:" [] ("Hello, " +++ name)
```

We extend the notational convention by putting also the task value names below each other, in the *lambda*-abstraction after the `>>=` task combinator. The example shows that the second argument of the bind combinator is a (very simple) computation that prefixes the `String` value `"Hello,"` to the given input of the user of the first task.

EXAMPLE 6. *Binding two tasks*

We bind `editTrack` and `viewTrack` and obtain a task that first allows the user to edit a track value, and when she confirms she is ready, displays the edited value.

```
editTask2 :: Track -> Task Track
editTask2 x =           editTrack x
  >>= \new -> viewTrack new
```

Note that the `editTask2` task function can also be written down slightly shorter because `viewTrack` is already a task function of a type that matches with the second argument of `>>=`:

```
// Alternative definition of editTask2:
editTask2 :: Track -> Task Track
editTask2 x =   editTrack x
  >>= viewTrack
```

The bind combinator `>>=` profits optimally of the fact that its second argument is a function that is applied to the information that is transferred from the first argument task to whatever task is computed by the function. This has the following advantages: (a) the information is available to all tasks that are created, and (b) we can compute what follow-up tasks to create, using the information and the full expressive power of the host language. We illustrate this with a number of examples.

EXAMPLE 7. *Availability of information*

Here is an alternative way of entering a track, by entering the fields in succession.

```
enterTrack :: Task Track
enterTrack
```

1

2

```

=           enterInformation "Select_medium:" []
3
>> \medium -> enterInformation "Enter_album:" []
4
>> \album  -> enterInformation "Enter_artist:" []
5
>> \artist -> enterInformation "Enter_year:" []
6
>> \year   -> enterInformation "Enter_track:" []
7
>> \track  -> enterInformation "Enter_title:" []
8
>> \title  -> enterInformation "Enter_time:" []
9
>> \time   -> enterInformation "Enter_tags:" []
10
>> \tags   -> return
11
        (newTrack medium album artist year track title time tags)
12
13
newTrack :: Medium Name Name Year TrackNr Name Time [Tag] -> Track
14
newTrack medium album artist year track title time tags
15
    = { medium = medium, album = album, artist = artist, year = year
16
        , track = track, title = title, time = time, tags = tags}
17

```

This example demonstrates two important aspects:

- the individual task values (`medium`, `album`, ...) that are retrieved during the execution can be used later in the sequence of tasks;
- the type-indexed character of the `enterInformation` task function is driven by the type of the `newTrack` function, which in turn is enforced by the type model of the `Track` record fields. In the first call of `enterInformation` it must yield a task value of type `Medium`, in the calls on lines 4, 5, and 8 the task value has type `String`, in the calls on lines 6 and 7 it is an `Int`, in line 9 it results in a `Time` task value, and finally, in line 10 it creates a task value of type `[String]`.

EXAMPLE 8. *Dependency of information*

In Example 7 the user can enter any number for the `year` field. It is much nicer to check for the earliest possible year depending on the value of the `medium` field of the track that is about to be entered. Suppose that we know of each music storage medium (except, of course, the `Other` case) when the first commercially available products were approximately available (see pattern matching, page 237):

```

firstYearPossible :: Medium -> Year
firstYearPossible BlueRay      = 2006
firstYearPossible DVD         = 1996
firstYearPossible MP3         = 1993
firstYearPossible CD          = 1981
firstYearPossible Musicassette = 1964
firstYearPossible Single      = 1949
firstYearPossible LP          = 1948
firstYearPossible other       = 0

```

Using this information, we construct a task that repeatedly asks the user to enter correct `year` values. The repetition is expressed recursively. Any entered value that appears earlier than deemed possible on that particular music storage medium is rejected.

```

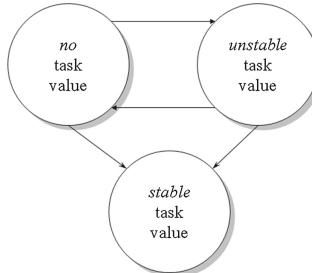
enterYear :: Medium -> Task Year
1
enterYear medium
2
= updateInformation "Enter_year:" [] first
3
>>= \year -> if (year >= first)
4
    (return year)
5
    ( viewInformation "Incorrect_year:" []
6
        ( medium +++> "s_were_not_available_before_" ++
7
            year +++> ".Please_enter_another_year."
8
        )
9
    )
10
>>| enterYear medium
11
)
12
where first = firstYearPossible medium

```

The predefined operator `++>` (line 7 and 8) converts its first argument to a `String` value and concatenates it with the second argument. A similar operator `<++>` is available in which the arguments are flipped. They can be used for any type of argument for which the generic `iTask` system has been generated. Also note the use of the naïve then combinator `>>|` on line 10: the task value of the messaging task is not relevant for asking the user again. ■

4.3 Intermezzo: Task Values

Now that we are getting in the business of composing tasks, we need to be more precise about tasks and task values. During execution, task values can change. A task can have no task value, e.g. which is initially the case for every `enterInformation` task function. A task value can be stable, e.g. which is the case with the `return` task. A task value may be unstable and varies over time, e.g. when the end-user changes information in response to an `updateInformation` function. It is entirely well possible that further processing of an unstable value eliminates the task value, for instance, when the end-user creates blank fields within the `updateInformation` task. Stable values, however, remain stable. The diagram below displays these possible transitions of task values.



Precisely these task values are available by means of the following two algebraic data types:

```

:: TaskValue a = NoValue | Value a Stability
:: Stability   = Unstable | Stable

```

Task combinator functions can inspect these task values and decide how they influence the composite behavior of tasks. This is also done by the `>>|` and `>>=` combinators. Both task combinator functions inspect the ‘stability’ of their first task argument’s task value during execution. As soon as that task produces a *stable* task value, the combinators make sure that the second task argument gets executed. If the first argument task has an unstable task value, then it is left to the user of the application to decide whether she is happy with that value. Hence, infrastructure is created to allow her to confirm that the current, unstable, value is fine to proceed with.

4.4 Parallel Composition

Alongside sequential composition is *parallel composition*, with which you express that tasks are available at the same time. We discuss two parallel task combinator functions that are often very useful. Because of their resemblance with the logical operators `&&` and `||`, their names are written as `-&&-` and `-||-` (pronounce as *and*, *or* respectively). Their signatures are:

```
(-&&-) infixr 4 :: (Task a) (Task b) -> Task (a, b) | iTask a & iTask b
(-||-) infixr 3 :: (Task a) (Task a) -> Task a | iTask a
```

The purpose of `-&&-` is to execute its argument tasks in parallel and assemble their individual task values into a pair. The types of the task values need not be of the same type, but this is of course allowed. The composite task only has a stable task value if both argument tasks have a stable task value. If either one of the argument tasks has no task value, then the composite task also does not possess one. In the other cases, the composite task has an unstable task value.

The purpose of `-||-` is to offer the user two alternative ways to produce a task value. For this reason, the types of its task arguments must be identical. The only situation in which the composite task does not have a task value is when both argument tasks have no task value. In any other case, the task value of the composite task is the task value of *the most recently changed or stable* task value.

EXAMPLE 9. *Entering an album with ‘and’*

Entering tracks individually is fine for albums with a small number of tracks, or for single purchases, but it is an inconvenient way of entering albums that have more than four tracks. We wish to enter the album information (with a task called `enterAlbumInfo`) separately from entering the track list (with a task called `enterTracklist`). We define the composite task `enterAlbum` that performs these tasks in parallel and combines their result with the pure computation `newAlbum`:

```
enterAlbum :: Task [Track]
enterAlbum
  =
    enterAlbumInfo -&&- enterTracklist
  >>= \ (info, tracks) -> return (newAlbum info tracks)
```

The distinct task value results of `enterAlbumInfo` and `enterTracklist` are called `info` and `tracks` respectively. The function `newAlbum` is a pure computation that creates a list of `Track` values (see list comprehensions, page 238):

```
newAlbum :: (Medium, Name, Name, Year) [(Name, Time, [Tag])] -> [Track]
newAlbum (medium, album, artist, year) tracks
  = [ newTrack medium album artist year nr song t tags
    \\ (song, t, tags) <- tracks & nr <- [1..]
  ]
```

The two tasks to enter the album information and the track list can proceed as described earlier. We choose sequential input for the album information, to allow the input of year values to be checked against the chosen medium value. The track list is entered as a list of track fields.

```
enterAlbumInfo :: Task (Medium, Name, Name, Year)
enterAlbumInfo
  = enterInformation "Select_medium:" []
  >>= \medium -> enterInformation "Enter_album:" []
  >>= \album -> enterInformation "Enter_artist:" []
  >>= \artist -> enterYear medium
  >>= \year -> return (medium, album, artist, year)

enterTracklist :: Task [(Name, Time, [Tag])]
enterTracklist
  = enterInformation "Enter_tracks:" []
```

EXERCISE 6. *Edit and view a track*

Add (`editTask2 track`) of Example 6 to your collection of top level tasks and compile and run your extended application. Manipulate the fields in the `editTrack` task and see when the bind combinator `>>=` allows you to enter the `viewTrack` task and when it prohibits you from doing that.

EXERCISE 7. *Edit and sort track tags*

Alter the `editTask2` task in such a way that before viewing the `new` track, the task first sorts the tag list of the `new` track using `sortTagsOfTrack` of Example 5. Hence, after editing a track, the user always sees a tag list in alphabetic order.

EXERCISE 8. *Edit and view a track list*

Create a recursive task of signature `enterTracks :: [Track] -> Task [Track]` that allows the user to enter tracks in succession. It displays the argument list of tracks, and appends new tracks to this list until the user decides that the list is complete. In that case, the accumulated track list is returned.

EXERCISE 9. *Compare ‘and’ with ‘or’*

Add the tasks `and` and `or` to your collection of top level tasks.

```

and =      updateInformation "A:" [] 42
          -&&- updateInformation "B:" [] 58
>>= \x -> viewInformation "C:" [] x

or =       updateInformation "A:" [] 42
          -||- updateInformation "B:" [] 58
>>= \x -> viewInformation "C:" [] x

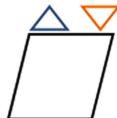
```

Compile and run your extended application. Explain the difference in behavior and return values.

5 Environment Interaction

In the previous section we have shown a number of ways to compose tasks. With these forms of composition communication between co-tasks is organized in a structured way. However, programs sometimes exhibit ad hoc communication patterns. This is often the case when interacting with the ‘external world’ and external tools need to be called, or persistent information is shared using the file system or databases.

In TOP, ad hoc communication between internal tasks and the external world is provided by means of *shared data sources*. A shared data source contains information which can be shared between different tasks or with the outside world, and can be read and written via a typed, abstract interface. *Shared data sources abstract over the way their content is accessed* in an analogous manner that tasks abstract over the way work is performed. We depict this in the following way:



The content \square of a shared data source can be (part of) the file system, a shared memory, a clock, a random stream, and so on. A shared data source can be read from via a typed interface \triangle and written to via another typed interface \triangledown . The read and write data types need not be the same. For instance, if the shared data source is a stopwatch, then the write type can represent stopwatch actions such as resetting, pausing, continuing, and so on, whereas its read type can represent elapsed time.

We explain how to get access to external resources in Sect. 5.1, and how to create local shared data sources in Sect. 5.2. Interactive tasks turn out to interact seamlessly with shared data sources. We integrate them in Sect. 5.3. Finally, we discuss two subjects that are concerned with the environment: basic file handling in Sect. 5.4 and basic time handling in Sect. 5.5.

5.1 Basic Environment Interaction

In this section we introduce the basic means to interact with external resources. We start with an example.

EXAMPLE 10. Limiting year input values

Time is an obvious external resource. Let us enhance the `enterYear` task of Example 8 (page 207) to also disallow inputs that exceed the current year. To obtain the current date we use the expression `(get currentDate)`, which is a task of type `(Task Date)`. `Date` is a predefined type:

```
:: Date = { day :: Int // 1..31
            , mon :: Int // 1..12
            , year :: Int }
```

We adopt the `enterYear` task to obtain the current date and use it to compare it with the user's input (see disambiguating records, page 242); (see guards, page 236):

```
enterYear :: Medium -> Task Year
1
enterYear medium
2
=           get currentDate
3
>>= \today -> updateInformation "Enter_year:" [] first
4
>>= \year -> if (year >= first && year <= today.Date.year)
5
      (return year)
6
      ( viewInformation "Incorrect_year:" []
        ( message year +++ ".Please_enter_another_year." ) )
7
      >>| enterYear medium
8
      )
9
)
10
where
11
  first       = firstYearPossible medium
12
  message year
13
  | year < first = medium +++> ("s_were_not_available_before" <+++ year)
14
  | otherwise     = "It_is_not_yet" <+++ year
15
```

In line 3 the current date is obtained from the environment. If the user input, provided in line 4, lies nicely between the two bounds, checked in line 5, then the input is returned. In the other case we provide the user with a matching message, defined by the function `message`, and start over again. ■

A shared data source that allows reading values of type `r` and writing values of type `w` is of type `ReadWriteShared r w`. For the time being, we consider this to be an opaque type with three access functions:

<code>get :: (ReadWriteShared r w) -> Task r iTask r</code>	
<code>set :: w (ReadWriteShared r w) -> Task w iTask w</code>	
<code>update :: (r->w) (ReadWriteShared r w) -> Task w iTask r & iTask w</code>	



The `get` and `set` access functions are task functions that read and write the shared data source. A frequently occurring pattern is to `get` a value `x` from a shared data source and immediately `set` it to `(f x)`. This can be shorthanded to `(update f)`.

In Example 10, `currentDate` is a shared data source that allows reading values of type `Date`, but it does not allow writing. This is expressed by using the trivial `Void` type (`:: Void = Void`) for its write interface type. Hence, `currentDate`

has type `ReadWriteShared Date Void`. For such *read only* shared data sources, a synonym type `ReadOnlyShared`, is introduced to express more clearly that you can only read values from such an entity. Analogously, for *write only* shared data source, the type synonym `WriteOnlyShared` is introduced. Finally, because often the read and write type interface is identical, the shorter type synonym `Shared` can be used.

```
:: Shared      rw ::= ReadWriteShared rw rw
:: ReadOnlyShared r ::= ReadWriteShared r Void
:: WriteOnlyShared w ::= ReadWriteShared Void w
```

The `currentDate` shared data source is an example of a globally available shared data source. One can imagine many such shared data sources, and in these lecture notes we encounter a few more. For now, we limit ourselves to three shared data sources that are concerned with time:

```
currentDate    :: ReadOnlyShared Date
currentTime    :: ReadOnlyShared Time
currentDateTime :: ReadOnlyShared DateTime

:: DateTime     = DateTime Date Time
```

Unsurprisingly, `currentTime` allows you to access the current time. `currentDateTime` is just a convenient way to get both the date and time in one go.

5.2 Ad Hoc Data Sharing

As explained above, the *iTask* toolkit provides you with a number of predefined shared data sources to ‘connect’ with the external world. You can also create shared data sources for internal purposes.

```
sharedStore :: !String !a -> Shared a | JSONEncode{!*!}, JSONDecode{!*!}, TC a
```

With `(sharedStore en ev)`, a shared data source is created which name is the result of evaluating `en`, and which initial content is the result of evaluating `ev`. The details of the classes `JSONEncode`, `JSONDecode`, and `TC` do not concern us right now: basically, they are available whenever you include `derive class iTask ...` for your model data types. The shared data source that you create with this function can be accessed with the `get`, `set`, and `update` functions of page 212.

EXAMPLE 11. A shared Track data source

We define a shared data source that can be used to manipulate a `Track` value:

```
trackStore :: Shared Track
trackStore = sharedStore "StoreTrack" track
```

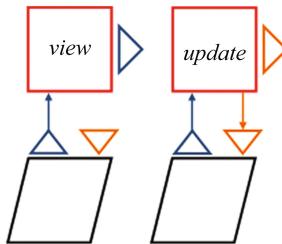
This creates a shared data source that is identified with the name “`StoreTrack`” and that has initial value `track` (page 199).

5.3 Interactive Tasks and Data Sharing

The interactive tasks `viewInformation` and `updateInformation` manipulate a model value. For your convenience, we repeat their signatures:

```
viewInformation  :: d [ViewOption m] m -> Task m | descr d & iTask m
updateInformation :: d [UpdateOption m m] m -> Task m | descr d & iTask m
```

These interactive tasks work *in isolation* on their task value, which is fine for many situations. However, work situations in which several interactive tasks view and update the *same information* require a more general version of these interactive tasks. Instead of editing the *current value* of the shared data source, they manipulate the *shared data source* directly.



In case of *viewing* the current value of a shared data source, its current value is *read* and *displayed*. In case of *updating* the current value of a shared data source, its current value is also read and displayed, but also *written* at each update.

Basically, this means that in the signatures above, the value type `m` must be replaced by an appropriate shared data source type. When doing this, we obtain the following, more general, interactive tasks:

```
viewSharedInformation  :: d [ViewOption r w] (ReadWriteShared r w) -> Task r
| descr d
& iTask r
updateSharedInformation :: d [UpdateOption r w] (ReadWriteShared r w) -> Task w
| descr d
& iTask r
& iTask w
```

These signatures show that the interactive tasks get ‘connected’ with a shared data source. For `viewSharedInformation`, this means that a task is created that displays the *current* value of the argument shared data source. Hence, whenever the shared data source obtains a new value, then this is displayed by the `viewSharedInformation` task. Because it *views* a value, its task value type corresponds with the *read* value type of the shared data source. The task always tries to show the current value of that can be read from the shared data source. Of course, when the shared data source is changed by someone, it may take some time before a task is informed that a change has happened.

The `updateSharedInformation` task also gets connected with a shared data source, but in addition to displaying the current value of the shared data source, it also allows updating its value. Every time this is done, all other ‘connected’

tasks refresh their displayed value as well. Because `updateSharedInformation` writes a value, its task value type corresponds with the `write` value type of the shared data source. Its task value is always the currently written value to the shared data source.

Viewing and updating tasks that are connected with shared data sources allows us to create intricate networks of interactive tasks (see Fig. 13).

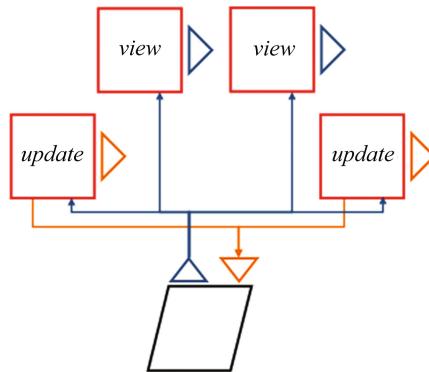


Fig. 13. Creating networks of interactive tasks via shared data sources

EXAMPLE 12. Update and view a shared data source

In this example we wish to create two tasks: one that allows the user to view and alter a `Track` value, and one that displays the result of these actions. This value is stored in the shared data source `trackStore` that was created in Example 11. Hence we need to combine two interactive tasks, one for viewing and one for updating a shared data source. We combine them with the ‘and’ operator `-&&-`:

```

editAndView :: Task (Track, Track)
editAndView
  = viewSharedInformation (Title "View_a_Track") [] trackStore
  -&&-
  updateSharedInformation (Title "Edit_a_Track") [] trackStore
  
```

The resulting task is depicted in Fig. 14. Any user action that is performed in the editing task is displayed in the viewing task. ■

Admittedly, in its current form Example 12 seems silly because the editing task already allows the user to view the current task value. However, if you imagine that the viewing task is performed by *another user*, then this is a sensible way of arranging work. In Sect. 6 we show how to distribute tasks to users. Nevertheless, also for a single user this pattern can make sense if only the viewing task processes the value of the shared data source to a more useful format and renders it accordingly. Up until now we have ignored the option list arguments of the interactive tasks. It is time to throw some light on this matter.

Viewing and updating tasks that are connected with a shared data source that reads its values as type `r` should be allowed to transform them to another

View a Track

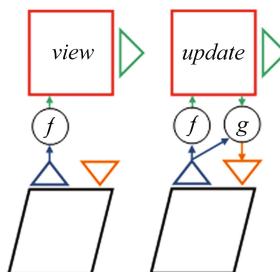
Source:	CD
Album:	Professor Satchafunklus and the musterion of rock
Artist:	Joe Satriani
Year:	2008
Track:	4
Title:	Professor Satchafunklus
Time:	00:04:47
Tags:	<ul style="list-style-type: none"> metal guitar rock instrumental guitar hero

Edit a Track

Source*:	CD	✓
Album*:	Professor Satchafunklus and the musterion of rock	✓
Artist*:	Joe Satriani	✓
Year*:	2008	✓
Track*:	4	✓
Title*:	Professor Satchafunklus	✓
Time*:	00:04:47	✓
Tags*:	<ul style="list-style-type: none"> metal guitar rock instrumental guitar hero 	<ul style="list-style-type: none"> (✓) (✓) (✓) (✓) (✓)

Fig. 14. Edit and view a shared track value.

domain of some type v using a function $f : r \rightarrow v$. The tasks then display and update values of the new domain. Hence, in case of updating tasks, the user creates a new value of type v that must be placed back into the shared data store that writes its values as some type w . In general, you need both the new value of type v and the current read value of the shared data source of type r . Hence, the new value to be stored in the shared data source is computed by a function $g : r v \rightarrow w$.



We want to associate f with the viewing task by means of a data constructor (`ViewWith f`) and the functions f and g with the updating task with data constructor (`UpdateWith f g`). The type definitions of these data constructors are:

```
:: ViewOption r = E.v: ViewWith (r -> v) & iTask v
:: UpdateOption r w = E.v: UpdateWith (r -> v) (r v -> w) & iTask v
```

Views need to be created of type v , so generic machinery for them has to be in place. This is enforced by the `iTask v` class constraint. The existential encapsulation `E.v` provides us with full freedom to choose any domain of our liking.

EXAMPLE 13. *Update and view a shared data source, revised*

We improve Example 12 by letting the viewing task only display a text message that informs the viewer what *album* of which *artist* is being edited. For this purpose, we add a viewing option to the viewing task (the rest of Example 12 remains unaltered):

```
editAndView :: Task (Track, Track)
editAndView
  = viewSharedInformation (Title "View_a_Track") [ViewWith view] trackStore
    -&&-
    updateSharedInformation (Title "Edit_a_Track") [] trackStore
where
  view :: Track -> String
  view track
  = "You_are_editing_album" +++ track.album +++ "\u2022by\u2022" +++ track.artist
```

The resulting view task is shown in Fig. 15. ■

5.4 File Interaction

Every so often, an application is required to access data that is stored in a format dictated externally. The application must read and write this data. Suppose that a file is stored at a location identified by the string value *filepath*. The task function (`importTextFile filepath`) obtains the entire content of that file as a string value, and (`exportTextFile filepath str`) replaces the entire current content of that file with *str*. The signatures of these task functions are:

```
:: FilePath ==> String

importTextFile :: FilePath -> Task String
exportTextFile :: FilePath String -> Task String
```

EXAMPLE 14. *Retrieving and storing tracks to file*

In this example we create two tasks: (a) a named task `importTracks` that imports tracks from a text file; and (b) a named task `exportTracks` that exports tracks to a text file. The format of the text file uses the *newline* character to

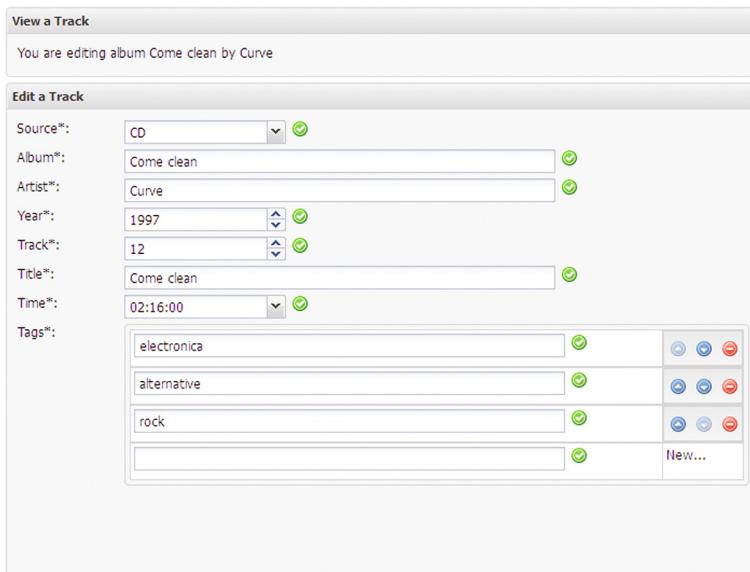


Fig. 15. Edit and view a shared track value, revised version.

separate entire entries, and uses the *tab* character to separate the fields within an entry. This is a fairly common format for simple databases.

We start with the *importing* task, naming it `importTracks`. Given a file location, it reads the file content and returns a list of `Track` values. It has signature:

```
importTracks :: FilePath -> Task [Track]
```

The function uses the `importTextFile` function to read in the entire contents of the text file. This provides us with a `String` value. We split the conversion of this value to a list of `Track` values into two steps: first, the entries and their fields are parsed (`tabSeparatedEntries`), and second, this list of fields is transformed to a list of track values (`toTrackList`). The signatures of these two functions are:

```
tabSeparatedEntries :: String -> [[String]]
toTrackList       :: [[String]] -> [Track]
```

With these two functions, `importTracks` can be defined:

```
importTracks :: FilePath -> Task [Track]
importTracks filepath
  =
    importTextFile filepath
  >>= \content -> return (toTrackList (tabSeparatedEntries content))
```

The two functions that still need to be implemented are pure computations:

```

tabSeparatedEntries :: String -> [[String]]          1
tabSeparatedEntries str
    = map (split "\t") (split "\n" str)              2
                                                3
toTrackList :: [[String]] -> [Track]                4
toTrackList entries
    = [ newTrack (fromString mdm) alb art (fromString yr) (fromString nr) title
        (fromString t)
        (split "," tags)                            5
    ] \\\ [mdm, alb, art, yr, nr, title, t, tags:_] <- entries
    ]                                              6
                                                7
                                                8
                                                9
                                                10
                                                11

```

The `split` function takes a separator string and source string and yields all source fragments that are separated by the separator string. The track tags are separated by a *comma* character, and hence, `split` can be used to create the list of tags (line 9). The `split` function is part of the `Text` module that needs to be imported explicitly. For converting textual representations of values to the values themselves, the host language *Clean* provides a type class `fromString`:

```
class fromString a :: !String -> a
```

For `Time` values, an instance is already provided in `iTask`. This is not the case for `Medium` values and `Int` values. Let us start with parsing `Medium` values:

```

fromString "BlueRay"      = BlueRay          1
fromString "DVD"          = DVD              2
fromString "MP3"          = MP3              3
fromString "CD"           = CD               4
fromString "Musicassette" = Musicassette   5
fromString "Single"       = Single           6
fromString "LP"           = LP               7
fromString other
| startsWith prefix other = Other postfix
where prefix             = "Other_"
  postfix                = other           8
fromString wrong          = abort ("unexpected input in fromString:" +++ wrong) 12

```

This is fairly straightforward: the only challenging bit concerns parsing `Other` values. The `startsWith` function from the `Text` module can be used to check whether the text starts with the `"Other_"` text (line 9), and, if this is the case, it can produce the correct value. In any other case, the text cannot be parsed, and a runtime error is generated (line 12).

For `Int` values, the situation is less complicated because the desired functionality is already available as the `String` instance of the `toInt` type class, which converts a `String` value to an `Int` value. Hence, the implementation of the `Int` instance of the `fromString` type class is trivial:

```
instance fromString Int where fromString str = toInt str
```

The *exporting* task basically needs to perform the inverse operations of the parts introduced above for the importing task. For that reason, we build this task in inverse order as well. Converting values to descriptions of these values is supported by the host language *Clean* with the type class `toString`:

```
class toString a :: !a -> String
```

For `Time` and `Int` values, instances are already available, but this is not the case for `Medium` values. However, because `Medium` is an instance of the `iTask` class, and hence can be serialized, its implementation is easy enough:

```
instance toString Medium where toString m = "" <+++ m
```

We proceed by defining the inverse operations of the functions `toTrackList` and call it `fromTrackList`, and `tabSeparatedEntries` and call it `tabSeparatedString`. Both functions use the inverse operation of `split`, which is called `join`. The `join` function takes a glue string and list of strings and concatenates the list elements, using the glue string between each element.

```
fromTrackList :: [Track] -> [[String]]  
fromTrackList tracks  
= [ [ toString medium, album, artist, toString year, toString track, title  
    , toString time, join "," tags]  
    \\ {medium, album, artist, year, track, title, time, tags} <- tracks  
]  
  
tabSeparatedString :: [[String]] -> String  
tabSeparatedString entries  
= join "\n" (map (join "\t") entries)
```

The tag list is joined with the *comma* character (line 4), and the fields and entries with the *tab* and *newline* character respectively (line 10). With these functions, we can define the exporting tracks task as follows:

```
exportTracks :: FilePath [Track] -> Task Void  
exportTracks filepath tracks  
= exportTextFile filepath (tabSeparatedString (fromTrackList tracks))  
>>| return Void
```

5.5 Time Interaction

In Sect. 5.1 we have shown how to obtain the current date and time. In many work situations it is important that tasks start at the right time, or are guaranteed to terminate within some specified time limit. For this purpose *iTask* offers a number of time related task functions:

```
waitForTime     :: !Time      -> Task Time  
waitForDate    :: !Date      -> Task Date  
waitForDateTime :: !DateTime -> Task DateTime  
waitForTimer   :: !Time      -> Task Time
```

The first three task functions wait until the specified time, date, or both has elapsed. Their task value result is identical to the argument value. The last task function waits the specified amount of time from the moment this task function is called. Its return value is the time when the timer went off.

EXAMPLE 15. *Extending tasks with a deadline*

In this example we create a new task combinator function that extends any given task t with a time limit d . The intended signature of this task combinator is:

```
deadline :: !(Task a) !Time -> Task (Maybe a)
```

The `Maybe` type represents an optional value and is defined as:

```
:: Maybe a = Nothing | Just a
```

Hence, no value is encoded as `Nothing`, and a value x as `(Just x)`.

The function `(deadline t d)` should execute task t . If t returns within time limit d with a result value x , then the combined task returns `(Just x)`. However, if t does not terminate within time limit d , then the combined task returns `Nothing`. Besides executing t this combinator executes a *timing task*. The first task that completes terminates the combined task. Hence, it makes sense to use the `-||-` task combinator (Sect. 4.4) for this purpose. It demands that its two task arguments have task values of the same type. If we let the timing task return `Nothing`, then all we need to do is make sure that the original task t returns `(Just x)` instead of just x . We create two wrapper functions for that purpose:

```
just :: !(Task a) -> Task (Maybe a) | iTask a
just    t = t >>= \x -> return (Just x)

nothing :: !(Task a) -> Task (Maybe b) | iTask a & iTask b
nothing t = t >>| return Nothing
```

`(just t)` executes t , and if it produces a stable task value x , it produces `(Just x)`. Similarly, `(nothing t)` also executes t , but after that produces a stable task value it is ignored, and instead only `Nothing` is returned.

The timing task can use `waitForTimer` task function:

```
timer :: !Time -> Task (Maybe a) | iTask a
timer d = nothing (waitForTimer d)
```

Hence, this is a task that waits the specified amount of time and then returns with `Nothing`. We can now implement the deadline task:

```
deadline :: !(Task a) !Time -> Task (Maybe a) | iTask a
deadline t d = (just t) -||- (timer d)
```

The argument task is executed, as well as the timer task. The first task that terminates determines the result of the combined task. ■

EXERCISE 10. Limiting year input values

In Example 10, the task function `enterYear` repeatedly asks the user for a year value until it lies within a given lower and upper bound value. It so happens that the *iTask* toolkit provides a data model for such kind of bounded values:

```
:: BoundedInt = { min :: Int    // the lower bound
                  , cur :: Int   // the current value ( $\min \leq \text{cur} \leq \max$ )
                  , max :: Int    // the upper bound
}
```

Use this type to define `enterYear` in such a way that it asks the user only once for a proper year value.

EXERCISE 11. Edit a track list and view information

In Exercise 8 you have created a task that allows the user to successively enter tracks. Enhance this task in a similar way as shown in Example 13. Display the *number of artists*, *number of albums*, *number of tracks*, and *total playing time*. ■

6 Collaboration

Up until this point we have discussed applications that serve a single user. We now extend this to serve arbitrarily many registered users. For this purpose we switch to the `multiTOPApp` or `multiTOPApps` kickstart wrapper functions (see Figs. 5 and 6). These wrapper functions add infrastructure to handle an arbitrary number of users. They use a custom defined module, `UserAdmin`. It is based on the core concept of a user. In this section, we use the functionality provided by the `UserAdmin` module.

When executing an application created by means of `multiTOPApp(s)`, the user is first asked to provide account information (see Fig. 16). This is used by the application to establish who it is serving. The infrastructure allows users to enter the application anonymously. It is up to the application whether or not this flaws the user experience. All applications maintain a shared data source containing information about the accounts and users that can be served by the application. The first thing to do is set up a collection of users (Sect. 6.1). As soon as a user base is available, an application can distribute its activities amongst the members of its user base (Sect. 6.2).

6.1 Employing Users

Employing users is actually not very different from adding tracks to a track list that is stored in a shared data source. Each application has a shared data source available that is called `userAccounts`. The involved type definitions are given in Fig. 17. Because `userAccounts` is a shared data source, it can be read with the task (`get userAccounts`) and written with the task (`set accounts userAccounts`) where `accounts` is a list of user account values. The

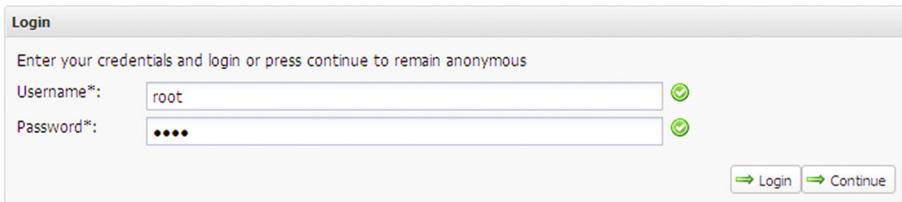


Fig. 16. Entering a multi-user application.

credentials consist of a user name and password. Note that the **Password** type is specialized within the *iTask* toolkit: when an editor is created for it it displays an edit box in which the user input is cloaked, as shown in Fig. 16. If a **title** is provided, then this is used by the application to address the user instead of her user name. Users can have different **roles** within an organization. Work can be assigned to users that have particular roles.

```
userAccounts :: Shared [UserAccount]

:: UserAccount = { credentials :: Credentials
                  , title      :: Maybe UserTitle
                  , roles      :: [Role]
                }
:: Credentials = { username   :: Username
                  , password   :: Password
                }
:: Password    = Password String
:: Username    = Username UserId
:: UserId      ::= String
:: UserTitle   ::= String
:: Role        ::= String
```

Fig. 17. Fragment of module **UserAdmin** concerning user accounts.

EXAMPLE 16. *The CLEAN company*

For illustration purposes, we introduce the fictitious CLEAN company. Its employees and their roles are displayed in Fig. 18. ■

EXERCISE 12. *Employ your users*

Create a main module that uses the kickstart wrapper function **multiTOPApps** to manage several top level tasks for multiple users. Fill its boxed task list with a task called **employ** that adds user accounts to the **userAccounts** shared data source. You can use any technique that has been discussed in the preceding sections or use the dedicated tasks in the **UserAdmin** module. Compile and run

Employee	Roles
	<i>Chris</i> team leader, sales, developer
	<i>Lucy</i> developer, system administrator
	<i>Emma</i> developer, system tester
	<i>Andrew</i> system tester, HMI
	<i>Nigel</i> finances, sales, project acquisition

Fig. 18. The CLEAN company employees and their roles.

to sign up all employees of the CLEAN company who are enumerated in Fig. 18. Choose passwords of your liking. Their user names are identical to their first names, so they do not require an additional title. ■

6.2 Distributing Work

Having a user base available, it is time to assign work to them. Before we explain how to do this, we first discuss the model types that are related with users and their properties. They are displayed in Fig. 19.

```

:: User          = AnonymousUser      SessionId
                  | AuthenticatedUser UserId [Role] (Maybe UserTitle)
:: UserConstraint = AnyUser
                  | UserWithId   UserId
                  | UserWithRole Role
  
```

Fig. 19. The user model types.

To an application, a user is either anonymous or belongs to the registered set of users. In the first case, a user is identified by means of the application's session which, for now, we consider to be opaque. Authenticated users are confirmed to be part of the collection of users that the application is allowed to serve. Their attributes originate from the user account details (Fig. 17). The UserConstraint model is used to define a subset of the authenticated users.

`AnyUser` imposes no constraint on this set, and hence, all users are eligible. In case of `(UserWithId uid)`, the user with username `(Username uid)` is selected. In case of `(UserWithRole role)`, any user that has `role` associated with her can be chosen.

`User`, `UserId`, and `UserConstraint` values can be used to indicate users to assign work to. If value `u` is of either of these types, then the task `(u @: t)` makes task `t` available to all users who belong to `u`. As soon as one of them decides to perform task `t`, it becomes unavailable to the other users. They receive some notification that task `t` is being executed by that user. The signature of operator `@:` is:

```
(@:) infix 3 :: user (Task a) -> Task a | iTask a & toUserConstraint user
instance toUserConstraint User
instance toUserConstraint UserId
instance toUserConstraint UserConstraint
```

EXAMPLE 17. Addressing the CLEAN company users

In the CLEAN company case, `AnyUser` refers to all employees. `(UserWithId "Lucy")` addresses Lucy. All sales persons, Chris and Nigel, are addressed with the value `(UserWithRole "sales")`. Hence, `(UserWithRole "sales") @: t` makes task `t` available to Chris and Nigel. The one who is the first to start on that task can finish it, and the other is informed that the job is being processed.

The user model types are ordinary types and therefore can also be used as values that are manipulated by the `iTask` type-driven functions. The task `(get currentUser)` can be used to find out which current `User` a task is serving.

EXAMPLE 18. Update and view a shared data source, distributed

We turn Example 12 into a distributed application. First, we assign the two sub tasks to two users:

```
editAndViewDistributed :: (user1, user2) -> Task (Track, Track)
                           | toUserConstraint user1 & toUserConstraint user2
editAndViewDistributed (user1, user2)
  = (user1 @: updateSharedInformation (Title "Edit_a_Track") [] trackStore)
    -&&-
  (user2 @: viewSharedInformation (Title "View_a_Track") [] trackStore)
```

Second, we determine who the current user is, and ask who is supposed to perform the view task while editing a track.

```
editAndViewTrack :: Task Track
editAndViewTrack
  =
    get currentUser
  >>= \me      -> updateInformation (Title "Enter_a_user_name") [] "user"
  >>= \you      -> editAndViewDistributed (me, you)
  >>= \_(track, _) -> return track
```

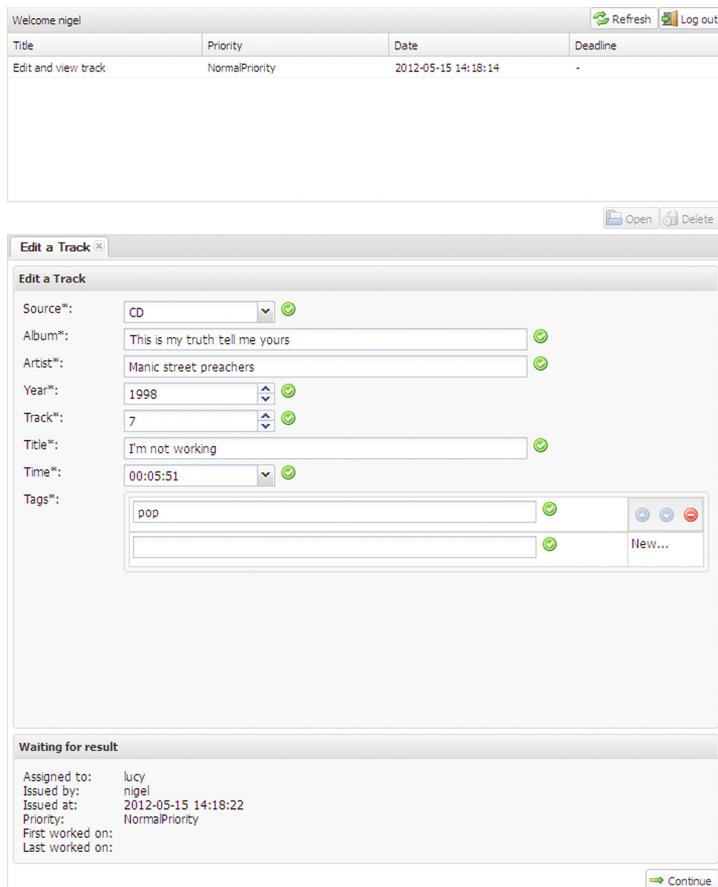


Fig. 20. Nigel, editing a track.

Suppose Nigel started the application and indicated Lucy to view his editing actions. Nigel can edit a track to his liking and tell that the sub task has been delegated to Lucy (Fig. 20). Lucy has received an extra task in her task list to follow Nigel's progress (Fig. 21). ■

EXAMPLE 19. *Making an appointment*

In this example, we create a task to make an appointment with a registered user.

```

appointment :: Task (Date, Time)                                1
appointment
=          get currentDate                                     2
>>= \today   -> get currentTime                           3
>>= \now    -> enterInformation (Title "Who_do_you_wish_to_meet?") [] 4
>>= \user   -> updateInformation (Title "When_to_meet?") [] [(today, now)] 5

```

```

>>= \options -> UserWithId user
    @:
        (updateInformation (Title "Select appropriate date-time pairs")
            [] (map toDisplay options)
        >>= return
    )
>>= \selected -> if (isEmpty selected)
    appointment
    (return (fromDisplay (hd selected)))

```

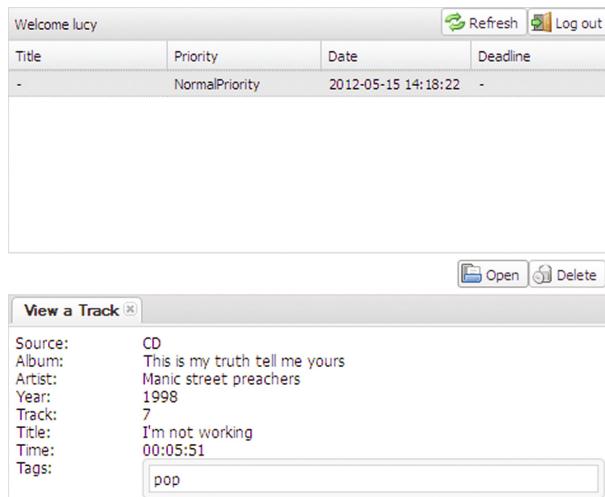


Fig. 21. Lucy, viewing Nigel's progress.

We obtain the current date and time (lines 3 and 4), ask the current user to choose a registered user (line 5) and create a number of possible date-time pairs (line 6). Hence, `user` and `options` are values of type `UserWithId` and `[(Date, Time)]` respectively. We ask `user` to select date-time pairs. Because we do not want her to alter these values, they are rendered as displays. She can only alter the *order* of suggested date-time pairs, and remove pairs. When done, the original user receives the selection as value `selected` (line 13). In case all options were inappropriate, the task starts all over again (line 14), otherwise the first pair is returned after stripping the `Display` data constructor of its model value. ■

EXERCISE 13. *Making an appointment*

Example 19 defines a task that never terminates in case the requested user consistently removes all suggested date-time pairs, or simply is inactive. Alter the example in such a way that the original user can decide to abandon this task. ■

7 Managing Work

Within an application, the various tasks need to keep track of each other's progress, and be able to change their course of action if necessary. In order to achieve this, an application needs to have means to detect and signal that its intended progress is hampered (Sects. 7.1 and 7.2) and it needs to adapt its behavior to handle new situations (Sect. 7.3).

7.1 Monitoring Work

In Sect. 6.2, we have introduced the task (`get currentUser`). Shared data sources such as `currentUser` prove to be a useful way for TOP applications to leave their trace by means of model data types that capture meta-information about their tasks. Before we discuss the model data types in detail, we first present the shared data sources that play a role in this context:

```
currentUser    :: ReadOnlyShared User
currentTopTask :: ReadOnlyShared TaskId
topLevelTasks  :: ReadOnlyShared (TaskList Void)
```

They are all *read-only* shared data sources because they are merely a shadow of the real tasks in progress. Similarly to `currentUser`, the named shared data source `currentTopTask` identifies *which task* is currently evaluated via an opaque value of type `TaskId`. For now, it suffices to know that a `TaskId` value uniquely identifies a task. The shared data source `topLevelTasks` gives access to all top level tasks that are being worked on. It basically gives you a list of meta-information values for each task that is being worked on. The `TaskId` value serves as key to find more information about a specific task. The meta-task information is fairly extensive, as displayed in Fig. 22. Right now, we are only interested in the `items` field that describes each and every top level task with a `(TaskListItem Void)` record value. It gives you its `TaskId` identification value and the current task value. The current task value may not seem very interesting for tasks of type `(Task Void)`, because it can only deliver `Void`. Still, one can tell whether or not this value is present, and, if so, whether or not it is stable. In the *iTask* system we can obtain a task list for parallel collections of tasks. The `TaskMeta` information is a list of key-value pairs that is used for layout purposes. This does not concern us right now. More interesting for keeping track of progress are the `ManagementMeta` and `ProgressMeta` model types. With this information, we can learn of a task's starting time, possible deadline, when it was last worked on, by whom it was issued, and so on.

EXAMPLE 20. Monitoring tasks

The easiest way to monitor the current tasks is by adding the following task to your application:

```
monitorTaskList :: Task Void
monitorTaskList
  =  viewSharedInformation (Title "TaskList:") [] topLevelTasks
```

```

>>| return Void

derive class iTask TaskList, TaskListId

:: TaskList      a = { listId          :: TaskListId a
                      , items           :: [TaskListItem a]
                      }
:: TaskListItem a = { taskId          :: TaskId
                      , value           :: TaskValue a           // SECTION 4.3
                      , taskMeta        :: TaskMeta
                      , managementMeta :: Maybe ManagementMeta
                      , progressMeta   :: Maybe ProgressMeta
                      }
:: TaskMeta      ::= [TaskAttribute]
:: TaskAttribute ::= (String, String)
:: ManagementMeta = { title          :: Maybe String
                      , worker         :: UserConstraint    // SECTION 6.2
                      , role           :: Maybe Role       // SECTION 6.1
                      , startAt        :: Maybe DateTime
                      , completeBefore :: Maybe DateTime
                      , notifyAt       :: Maybe DateTime
                      , priority        :: TaskPriority
                      }
:: ProgressMeta  = { issuedAt       :: DateTime
                      , issuedBy        :: User            // SECTION 6.2
                      , status          :: Stability
                      , firstEvent      :: Maybe DateTime
                      , latestEvent     :: Maybe DateTime
                      }
:: TaskPriority  = HighPriority
                  | NormalPriority
                  | LowPriority

```

Fig. 22. The model types that provide task meta-information.

Using the technique described in Sect. 5.3, it connects a display to the `topLevel` Tasks shared data source, thus allowing the end user to keep an up-to-date view of the set of top level tasks. As soon as the user chooses to continue, `monitorTaskList` terminates.

7.2 Monitoring Data

Interactive tasks can be connected with a shared data source. This is useful, as demonstrated by Example 20, because it keeps us up-to-date with the *current* value of the shared data source. However, sometimes we need to know *when* a shared data source is altered. In general, we want to impose a condition on the read value of a shared data source that acts as a trigger to continue evaluation. This can be done with the task function `wait`:

```
wait :: d (r -> Bool) (ReadWriteShared r w) -> Task r | descr d & iTask r
```

Just like interactive tasks, `wait` receives a description of its purpose that is displayed to the user. The predicate p of type $(r \rightarrow \text{Bool})$ is a condition on the current read value of the shared data source of type $(\text{ReadWriteShared } r \ w)$. As soon as the shared data source has a read value x for which $(p \ x)$ evaluates to `True`, then this also results in a stable task value x for the `wait` task.

EXAMPLE 21. Monitoring data

Consider this application of the `wait` task function:

```
waitForChange :: (ReadWriteShared r w) -> Task r | iTask r
waitForChange rws
  =
    get rws
  >>= \current -> wait (Title "Waiting_for_new_value:") ((!=!) current) rws
```

The task first reads the current value of the shared data source, and then monitors the shared data source until it has a different value. This difference is determined by the generic unequality operator \neq that is part of the `iTask` class. (This is also true for the generic equality operator \equiv .) ▀

7.3 Change Course of Action

In the previous two sub sections we have discussed how to monitor tasks and shared data sources. This can be used to *signal* deviating or unexpected behavior, and try to *respond* to these situations.

For *signalling*, TOP supports *exception handling* in a common *try-catch* style. We can use the following two task functions for this purpose:

```
throw :: !e -> Task a | iTask a & iTask e
try  :: (Task a) (e -> Task a) -> Task a | iTask a & iTask e
```

When a task encounters a situation that cannot be handled locally or sufficiently gracefully, it can *throw an exception value*, using the task function `(throw e)`, where e is an arbitrary expression that is completely reduced to a value. The expression can use the available local information to create some useful model value. As always, any type is valid, provided that the generic machinery has been made available for it. In `(try t r)`, task t is evaluated. If it throws no exceptional value then the task value of t is also the task value of `(try t r)`. However, if at some point within evaluation of t an exceptional value v is thrown, then evaluation of t is abandoned. If the type of the exceptional value v can be unified at run-time with the statically known type e of the exception handler r , then evaluation continues with `(r v)`. In that case this is also the result of `(try t r)`. If the two types cannot be unified (typically when an exception is raised for which this exception handler has not been designed) then `(try t r)` itself re-throws the very same exceptional value v , hoping that its context can handle the exception. Uncaught exceptions that escape all exception handlers are finally caught at the top-level, and only reported to the user.

EXAMPLE 22. *File import and export exceptions*

In Sect. 5.4 we have introduced the basic file import and export task functions `importTextFile` and `exportTextFile`. Both functions might throw an exception of type `FileNotFoundException`:

```
:: FileNotFoundException = FileNotFoundException !FilePath !FileError
:: FileError      = CannotOpen | CannotClose | IOError
```

Here, `FilePath` has the same role as in the argument of the two task functions and is supposed to point to a valid text file. The `FileError` values provide more detail about the nature of the exception. In case of `CannotOpen`, the indicated file could not be opened, either because it did not exist or because it was locked, perhaps by another task or application. In case of `CannotClose`, the indicated file could not be closed after reading the content. Other errors are report by means of `IOError`.

With these exceptions, we can enhance the tasks that were defined in Example 14, viz. `importTracks` and `exportTracks` that both assumed that everything is executed flawlessly. For `importTracks`, it makes sense to alter the task result type to a `Maybe` value that signals that the file was not read properly:

```
importTracks :: FilePath -> Task (Maybe [Track])
importTracks filepath
  = try (
      importTextFile filepath
      >>= \content -> return (Just (toTrackList (tabSeparatedEntries content)))
    ) handleFileError
where
  handleFileError :: FileNotFoundException -> Task (Maybe [Track])
  handleFileError _ = return Nothing
```

For `exportTracks`, we alter the task value type to a `Bool` to properly report success or failure:

```
exportTracks :: FilePath [Track] -> Task Bool
exportTracks filepath tracks
  = try (
      exportTextFile filepath (tabSeparatedString (fromTrackList tracks))
      >>| return True
    ) handleFileError
where
  handleFileError :: FileNotFoundException -> Task Bool
  handleFileError _ = return False
```

Note how both task functions introduce an exception handler with an explicit type. This is required by the signature of `try`, which needs to know for which type the event handler is defined.

For *responding*, TOP allows you to *terminate* currently running tasks as well as dynamically *create* new tasks.

```

removeTask      :: !TaskId !(ReadOnlyShared (TaskList a))
                -> Task Void | iTask a
appendTopLevelTaskFor :: !user !(Task a) -> Task TaskId | iTask a
                                         & toUserConstraint user

```

(`removeTask tid sds`) locates the task that is identified by `tid` within the given shared data source task list administration `sds` and stops and removes that task if found. Note that for argument `sds`, you can use the shared data source `topLevelTasks` that was defined in Sect. 7.1. (`appendTopLevelTaskFor u t`) dynamically creates a new task `t` for (any of the) user(s) `u`, in a similar way to the task assignment operator `@:` (Sect. 6.2). The difference is that `appendTopLevelTaskFor` only *spawns* `t` and returns with the stable task value that identifies the spawned task. In contrast, (`u @: t`) creates a *stub* in the current task that tells the current user that task `t` has been spawned for (any of the) user(s) `u`, and that you need to wait for its result task value.

EXAMPLE 23. *Birthday cake at the CLEAN company*

In the CLEAN company, it is a good habit to celebrate one's birthday with cake. We develop a task to select a time of day and invite everybody else for cake. We get to know our colleagues via the task (`get userAccounts`) (Fig. 17). From this list it is easy to obtain all names:

```

names :: [UserAccount] -> [UserId]
names accounts
= [uid \\ {credentials={username=Username uid}} <- accounts]

```

To invite everybody (except yourself) and announce your birthday, we need to obtain our identity with (`get currentUser`) (Sect. 7.1) and our name:

```

name :: User -> UserId
name (AuthenticatedUser id _) = id name
anonymous                      = "somebody"

```

The invitation displays the occasion and time.

```

cake :: UserId Time -> Task String
cake person time
= viewInformation "Birthday_cake" [] ("To_celebrate_" <++ person <++
                                         "s_birthday,_we_have_cake_at_" <++ time
                                         )

```

All that remains to be done is to put these parts in the right order:

birthdaycake :: Task [TaskId]	1
birthdaycake	2
=	3
get userAccounts	3
>> \accounts -> get currentUser	4
>> \me -> get currentTime	5
>> \now -> updateInformation (Title "When_to_eat_cake?") [] now	6
>> \time -> let colleagues = removeMembers (names accounts) [name me]	7

invite = cake (name me) time in	8
all (map (flip appendTopLevelTaskFor invite) colleagues)	9

The user accounts are obtained (line 3), the current user is determined (line 4), as well as the current time (line 5). We think of a suitable time to treat to cake (line 6) and exclude ourselves from the list of colleagues (line 7). The invitation task (line 8) is finally sent to every colleague (line 9). The `all` function is a task combinator function defined for the occasion: it receives a list of tasks, executes them all, and collects their resulting task values for further processing:

```
all :: [Task a] -> Task [a] | iTask a
all []      = return []
all [t:ts]  =          t
              >>= \v -> all ts
              >>= \vs -> return [v:vs]
```

EXERCISE 14. Improved user feedback

In Example 22, the exception handlers do not attempt to inform the user that anything has gone wrong. Define for both task functions better exception handlers that tell the user what exception has occurred, and remind her of the file path that was used.

EXERCISE 15. Remove birthday cake invitations

In Example 23, all users except the initiator receive an extra task. Of course it is polite to remove these tasks for these users after the event. Extend the `birthdaycake` task in such a way that the extra tasks are removed, using the `removeTask` function that has been presented in this section.

8 Related Work

The TOP paradigm emerged during continued work on the iTask system. In its first incarnation [2], *iTask1*, the notion of tasks was introduced for the specification of dedicated workflow management systems. In *iTask1* and its successor *iTask2* [3], a task is an opaque unit of work that, once completed, yields a result from which subsequent tasks can be computed. When deploying these systems for real-world applications, viz. in telecare [4] and modeling the dynamic task of coordinating Coast Guard Search and Rescue operations [5,6] it was observed that this concept of task is not adequate to express the coordination of tasks where teams constantly need to be informed about the progress made by others. The search for better abstraction has resulted in the TOP approach and task concept as introduced in these lecture notes.

Task-Oriented programming touches on two broad areas of research. First the programming of interactive multi-user (web) applications, and second the specification of tasks.

There are many languages, libraries and frameworks for programming multi-user web applications. Some of them are academic, and many more are in the

open-source and proprietary commercial software markets. Examples from the academic functional programming community include: the Haskell cgi library [7]; the Curry approach [8]; writing xml applications [9] in *SMLserver* [10]; WashCGI [11]; the Hop [12,13] web programming language; Links [14] and formlets [15]. All these solutions address the technical challenges of creating multi-user web applications. Naturally, these challenges also need to be addressed within the TOP approach. The principal difference between TOP and these web technologies is the emphasis on using tasks both as modeling and programming unit to abstract from these issues, including coordination of tasks that may or may not have a value.

Tasks are an ambiguous notion used in different fields, such as Workflow Management Systems (WFMS), human-computer interaction, and ergonomics. Although the *iTask1* system was influenced and partially motivated by the use of tasks in WFMSs [16], *iTask3* has evolved to the more general TOP approach of structuring software systems. As such, it is more similar in spirit to the WebWorkFlow project [17], which is an object oriented approach that breaks down the logic into separate clauses instead of functions. Cognitive Task Analysis methods [18] seek to understand how people accomplish tasks. Their results are useful in the design of software systems, but they are not software development methods. In Robotics the notion of task and even the “Task-Oriented Programming” moniker are also used. In this field it is used to indicate a level of autonomy at which robots are programmed. To the best of our knowledge, TOP as a paradigm for interactive multi-user systems, rooted in functional programming is a novel approach, distinct from other uses of the notion of tasks in the fields mentioned above.

9 Conclusions and Future Work

In this paper we introduced Task-Oriented Programming, a paradigm for programming interactive web-based multi-user applications in a domain specific language, embedded in a pure functional language.

The distinguishing feature of TOP is the ability to concisely describe and implement collaboration and complex interaction of tasks. This is achieved by four core concepts: (1) *Tasks observe intermediate values of other tasks* and react on these values before the other tasks are completely finished. (2) *Tasks running in parallel communicate via shared data sources*. Shared data sources enable useful lightweight communication between related tasks. By restricting the use of shared data sources we avoid an overly complex semantics. (3) *Tasks interact with users based on arbitrary typed data*, the interface required for this type is derived by type driven generic programming. (4) *Tasks are composed to more complex tasks using a small set of combinators*.

Commonly, web applications are *heterogeneous*, i.e.: they are constructed out of components that have been developed using different programming languages and programming tools. An advantage of the TOP approach is that even complex applications can be defined in one formalism.

TOP is embedded in Clean by offering a newly developed *iTask3* library. We have used TOP successfully for the development of a prototype implementation of a Search and Rescue decision support system for the Dutch Coast Guard. The coordination of such operations requires up-to-date information of subtasks, which is precisely suited for TOP. The *iTask* system has also successfully been used to investigate more efficient ways of working on Navy Vessels. The goal here is to get a significant reduction of crew members and systems. There are many application areas where the TOP approach can be of use. With industrial partners we want to investigate and validate the suitability of the TOP paradigm to handle complex real world distributed application areas in several domains.

Acknowledgements. The authors wish to thank the reviewers for their constructive feedback.

A Functional Programming in Clean

This section gives a brief overview of functional programming in *Clean* [19]. *Clean* is a pure lazy functional programming language. It has many similarities with *Haskell*.

A.1 Clean Nutshells

This section contains a set of brief overviews of topics in *Clean*. These overviews should be short enough to read while studying other parts of this paper without loosing the flow of those parts. The somewhat experienced functional programmer is introduced to particular syntax or language constructs in *Clean*.

Modules. A module with name M is represented physically by two text files that reside in the same directory: one with file name $M.dcl$ and one with file name $M.icl$.

The $M.icl$ file is the *implementation* module. It contains the (task) functions and data type definitions of the module. Its first line repeats its name:

implementation module M

An implementation module can always use its own definitions. By importing other modules, it can use the definitions that are made visible by those modules as well:

import M_1, M_2, \dots, M_n

The $M.dcl$ file is the *definition* module. It contains M 's interface to other modules. The first line of a definition module also gives its name:

definition module M

A definition module basically serves two purposes.

- It exports identifiers of its own implementation module by repeating their signature. Hence, identifiers which signatures are not repeated are *cloaked* for other modules.
- It acts as a serving-hatch for identifiers that are exported by other modules by importing their module names. In this way you can create libraries of large collections of related identifiers.

Operators. *Operators* are binary (two arguments) functions that can be written in *infix* style (between its arguments) instead of the normal *prefix* style (before its arguments). Operators are used to increase readability of your programs. With an operator declaration you associate two other attributes as well. The first attribute is the *fixity* which indicates in which direction the binding power works in case of operators with the same precedence. It is expressed by one of the keywords `infixl`, `infix`, and `infixr`. The second attribute is its *precedence* which indicates the binding power of the operator. It is expressed as an integer value between 0 and 9, in which a higher value indicates a stronger binding power.

The snapshot below of common operators as defined in the host language *Clean* illustrates this.

```
class (==) infix 4 a :: !a !a -> Bool
class (+)  infixl 6 a :: !a !a -> a
class (-)  infixl 6 a :: !a !a -> a
class (*)  infixl 7 a :: !a !a -> a
class (/)  infixl 7 a :: !a !a -> a
class (^)  infixr 8 a :: !a !a -> a
```

(These operators are *overloaded* to allow you to instantiate them for your own types.) Due to the lower precedence of `==`, the expression $x + y == y + x$ must be read as $(x + y) == (y + x)$. Due to the fixities, the expression $x - y - z$ must be read as $(x - y) - z$, and $x ^ y ^ z$ as $x ^ (y ^ z)$. In case of expressions that use operators of the same precedence but with conflicting fixities you must work out the correct order yourself using brackets ().

Guards. Pattern matching is an expressive way to perform case distinction in function alternatives, but it is limited to investigating the structure of function arguments. *Guards* extend this with conditional expressions. Here are two examples.

<pre>sign :: !Int -> Int sign 0 = 0 sign x x < 0 = -1 sign x = 1</pre>
--

<pre>instance < Date where < x y x.year < y.year = True x.year == y.year x.mon < y.mon = True x.mon == y.mon = x.day < y.day otherwise = False otherwise = False</pre>	1 2 3 4 5 6 7 8
---	--------------------------------------

In `sign`, the first alternative matches only if the argument evaluates to the value 0. In that case, `sign` results in the value 0. The second alternative imposes no pattern restrictions, but it does have a guard (`| x < 0`). Even though the pattern always matches, evaluation of the guard must result in `True` if the second alternative of `sign` is to be chosen. Therefor, the value -1 is returned only if the argument is a negative number. Finally, the last alternative has neither a pattern restriction nor a guarded restriction, and therefor matches all remaining cases, which concern the positive numbers. In those cases, the result is 1.

The implementation of `<` for `Date` values illustrates *nested* guards. In contrast with top-level guards, nested guards must be completed with `otherwise` to catch any remaining cases. The `otherwise` keyword can also be used in top-level guards, as is shown on the last line of the `<` function. The `<` function first checks the guard on line 3 and returns `True` if the first `year` field is smaller than the second `year` field. If the guard evaluates to `False`, then the second guard on line 4 is tested. In case of equal `year` field values, evaluation continues with the nested guards on lines 5–7 that inspect the `month` fields. If the first nested guard on line 5 evaluates to `True`, then the comparison also yields `True`. In case of a `False` result, the second nested guard on line 6 is tested. In case of equal `month` field values, the comparison of the `day` values provides the final answer. Finally, to complete the nested guards, the last case on line 7 concludes that the first argument is not smaller than the second, a conclusion that is shared by the last top-level guard on line 8.

Choice and Pattern Matching. In Example 8 the function `firstYearPossible` uses *pattern matching* to relate values of type `Medium` with year values. The `enterYear` function uses `if` to determine whether or not the user's input is valid. Unlike most programming languages, in which an *if-then-else* construct is supported in the language, it can be straightforwardly incorporated as a function in a lazy functional language, using pattern matching as well. Let's examine the type and implementation of `if`:

```
if :: !Bool a a -> a
if True then else = then
if _      _    else = else
```

The *type* tells you that the `Bool` argument is strict in `if`: it must always be evaluated in order to know whether its result is `True` or `False`. The *implementation* uses the evaluation strategy of the host language to make the choice effective. The `if` function has two alternatives, each indicated by repeating the function name and its arguments. Alternatives are examined in textual order, from top to bottom. Up until now the arguments of functions were only variables, but in fact they are *patterns*. A pattern *p* is one of the following.

- A *variable*, expressed by means of an identifier that starts with a lowercase character or simply the `_` wildcard symbol in case the variable is not used at all.
- A variable identifies and matches any computation without forcing evaluation.
- Within the same alternative, the variable identifiers must be different.

- A *constant* in the language, such as 0, `False`, 3.14, '\$', and "hello,world". To match successfully, the argument is evaluated fully to determine whether it has exactly the same constant value.
- A *composite* pattern, which is either a *tuple* (p_1, \dots, p_n) , a *data constructor* $(d\ p_1 \dots p_n)$ where n is the arity of d , a *record* $\{f_1=p_1, \dots, f_n=p_n\}$, or a *list* $[p_1, \dots, p_n]$ or $[p_1, \dots, p_n : p_{n+1}]$. Matching proceeds recursively to each part that is specified in the pattern. In case of records, only the mentioned record fields are matched. In case of lists, p_1 upto p_n are matched with the first n elements of the list, if present, and p_{n+1} with the remainder of the list.

Patterns control evaluation of arguments *until it is discovered that it either matches or not*. Only if all patterns in the same alternative match, computation proceeds with the corresponding right-hand side of that alternative; otherwise computation proceeds with the next alternative.

Hence, in the case of `if` its second argument is returned if the evaluation of the first argument results in `True`. If it results in `False` the second alternative is tried. Because it does not impose any restriction, and hence also causes no further evaluation, it matches, and the third argument is returned.

In `firstYearPossible` the data constructors are also matched from top to bottom. The last case always matches, and returns the value 0.

List Comprehensions. Lists are the workhorse of functional programming. *List comprehensions* allow you to concisely express list manipulations. Their simplest form is:

$[e \setminus\setminus p \leftarrow g]$

Generator g is an expression that is or yields a list. (Note that g can also evaluate to an array. In that case you need to use $\leftarrow:$ instead of \leftarrow to extract array elements.) From the generator, values are extracted from the front to the back. Each value is matched with the pattern p . If this succeeds, then the pattern variables in p are bound to the corresponding parts of the extracted value, and expression e , that typically uses these bound pattern variables, yields an element of the result list. If matching fails, then the next element of the generator is tried.

Besides the pattern p , elements can also be selected using a *guarded condition*:

$[e \setminus\setminus p \leftarrow g \mid c]$

Here, c is a boolean expression that can use any of the pattern variables that are introduced at generator patterns to its left. For each extracted value from the sequence for which the pattern match succeeds, the guarded condition is evaluated. Only if the condition also evaluates to `True`, a list element is added.

It is possible to use several pattern-generator pairs $p \leftarrow g$ in one list comprehension. They are combined either in *parallel* with the & symbol or as a *cartesian product* with the , symbol.

- In $p_1 \leftarrow g_1 \& p_2 \leftarrow g_2$, values are extracted from g_1 and g_2 at the same index positions and matched against p_1 and p_2 respectively. The shortest generator determines termination of this value-extraction process.

- In $p_1 \leftarrow g_1$, $p_2 \leftarrow g_2$, for each extracted value from g_1 that matches p_1 all values from g_2 are extracted and matched against p_2 .

Each and every one of the above ways to manipulate lists is already very expressive. However, they can be combined in arbitrary ways. This can be daunting at times, but once you get used to the expressive power, list comprehensions often prove to be the best tool for list processing tasks.

λ -Abstractions. *Lambda*-abstractions $\lambda x \rightarrow e$ allow you to introduce *anonymous* functions ‘on the spot’. They typically occur in situations where an ad hoc function is required, for which it does not make much sense to come up with a separate function definition. This frees you from thinking of a proper identifier and perhaps a type signature as well. The bind combinator $>>=$ is an excellent example of such a situation because in general you need to give a name x to the task value of the first task, and want to give an expression e that uses x . If you weren’t interested in x , you would have used the naïve then combinator $>>|$ instead.

Modelling Side-Effects. In a pure functional programming language all results must be explicit function results. This implies that a changed state should also be a function result. The type of the `Start` function in Example 1 is `*World -> *World`, this indicates that it changes the world. There are two things worth noting at this moment:

- The basic type `World` is *annotated* with the *uniqueness attribute* `*`. In a function type any argument can be annotated with this attribute. This enforces the property that whenever the function is evaluated, it has the *sole reference* to the corresponding argument value. This is useful because it allows the function implementation to *destructively update* that value without compromising the semantics of the functional programming language. This can only be done if the function body itself does not violate this uniqueness property. This is checked statically.
- The basic type `World` represents the ‘external’ environment of a program. If the `Start` function has an argument, the language assumes that it is of type `World`. The language provides no other means to create a value of type `World`, so if an application is to do any interaction with the external environment, it must have a `Start` function with a uniquely attributed `World` argument.

Incorporating side-effects safely in a functional language has received a lot of attention in the functional language research community. For lazy functional languages a host of techniques has been proposed. Well-known examples are *monads*, *continuations*, and *streams*. For eager functional languages, the situation is less complicated because in these languages programs exhibit an execution order that is more predictable.

Signatures. A signature $x :: t$ declares that identifier x has type t . An identifier x starts with a lowercase or uppercase letter and has no whitespace characters. The type t can be either of the following forms.

- It is one of the basic types, which are: `Bool`, `Int`, `Real`, `Char`, `String`, `File`, and `World`.
- It is a type variable. Their identifiers start with a lowercase character.
- It is a composite type, using one of the language type constructors `[]`, `{ }`, `(,)`, and `->`.
 - If t is a type, then `[t]` is the *list-of-t* type.
 - If t is a type, then `{t}` is the *array-of-t* type.
 - If t_1 and t_2 are types, then `(t1, t2)` is the *tuple-of-t₁-and-t₂* type. This generalizes to t_1 upto t_n with $2 \leq n \leq 32$, separating each type by `,`. Hence, `(t1, t2, t3)`, `(t1, t2, t3, t4)` and so on are also tuple types.
 - If t_1 and t_2 are types, then $t_1 -> t_2$ is the *function-of-t₁-to-t₂* type. This generalizes to $t_1 \dots t_n -> t_{n+1}$, where $t_1 \dots t_n$ are the argument types, and t_{n+1} is the result type. The function argument types are separated by whitespace characters. So, $t_1 \; t_2 -> t_3$, $t_1 \; t_2 \; t_3 -> t_4$ and so on are also function types.
- It is a custom defined type, using either an algebraic type or a record type. Their type names are easily recognized because they always start with an uppercase character. Examples of algebraic and record types can be found in Sect. 3.3.

Signatures can be *overloaded*, in which case they are extended with one or more *overloading constraints*, resulting in $x :: t \mid tc_1 \; a_1 \; \& \dots \& \; tc_n \; a_n$. A constraint $tc_i \; a_i$ is a pair of a type class tc_i and a type variable a_i that must occur in t . Note that $tc_1 \; a \; \& \; tc_2 \; a \; \& \dots \& \; tc_n \; a$ can be shorthanded to $tc_1, \; tc_2, \dots, tc_n \; a$.

Overloading. *Overloading* is a common and useful concept in programming languages that allows you to use the same identifier for different, yet related, values or computations. In the host language *Clean* overloading is introduced in an explicit way: if you wish to reuse a certain identifier x , then you declare it via a *type class*:

```
class x a1 ... an :: t
```

with the following properties:

- the *type variables* $a_1 \dots a_n$ ($n > 0$) must be different and start with a lowercase character;
- the *type scheme* t can be any type that uses the type variables a_i .

This declaration introduces the type class x with the single *type class member* x . It is possible to declare a type class x with several type class members $x_1 \dots x_k$:

```
class x a1 ... an
  where x1 :: t1
        :
        xk :: tk
```

It is customary, but not required, that in this case identifier x starts with an uppercase character. The identifiers x_i need to be different, and their types t_i can use any of the type variables a_i .

Type classes can be *instantiated* with concrete types. This must always be done for all of its type variables and all type class members. The general form of such an instantiation is:

```
instance x t'1 ... t'n | tc1 b1 & ... & tcm bm
  where ...
```

with the following properties:

- the types $t'_1 \dots t'_n$ are substituted for the type variables $a_1 \dots a_n$ of the type class x . They are not required to be different but they are not allowed to share type variables;
- the types t'_i can be overloaded themselves, in which case their type class constraints $tc_i b_i$ are enumerated after $|$ (which is absent in case of no constraints). The type variable b_i must occur in one of the types t'_i ;
- the **where** keyword is followed by implementations of all class member functions. Of course, these implementations must adhere to the types that result after substitution of the corresponding type schemes t_i .

Algebraic and \exists -Types. The `BoxedTask` type in Fig. 6 is an example of an *algebraic type* that is *existentially quantified*. Algebraic types allow you to introduce new constants in your program, and give them a type at the same time. The general format of an algebraic type declaration is:

```
:: t a1 ... am = d1 t11 ... t1c1 | ... | dn tn1 ... tncn
```

with the following properties:

- the type constructor t is an identifier that starts with an uppercase character;
- the type variables a_i ($0 \leq i \leq m$) must be different and start with a lowercase character;
- the data constructors d_i ($1 \leq i \leq n$) must be different and start with an uppercase character;
- the data constructors can have zero or more arguments. An argument is either one of the type variables a_i or a type that may use the type variables a_i .

From these properties it follows that all occurrences of type variables in data constructors (all right hand side declarations) must be accounted for in the type constructor (on the left hand side). With *existential quantification* it is possible to circumvent this: for each data constructor one can introduce type variables

that are known only locally to the data constructor. A data constructor can be enhanced with such local type variables in the following way:

$$\text{E. } b_1 \dots b_k : d_i t_{i1} \dots t_{ic_i} \& tc_1 x_1 \& \dots \& tc_l x_l$$

with the following properties:

- the type variables b_j ($0 \leq j \leq k$) must be different and start with a lowercase character;
- the arguments of the data constructor d_i can now also use any of the existentially quantified type variables b_i ;
- the pairs $tc\ x$ are type class constraints, in which tc indicates a type class and x is one of the existentially quantified type variables b_i .

From these properties it follows that it does not make sense to introduce an existentially quantified type variable in a data constructor without adding information how values of that type can be *used*. There are basically two ways of doing this. The first is to add functions of the same type that handle these encapsulated values (in a very similar way to methods in classes in object oriented programming). The second is to constrain the encapsulated type variables to type classes.

Record Types. Record types are useful to create named collections of data. The parts of such a collection can be referred to by means of a field name. The general format of a record type declaration is:

$$\text{:}\text{: } t\ a_1 \dots a_m = \{ r_1 :\text{: } t_1, \dots, r_n :\text{: } t_n \}$$

with the following properties:

- the type constructor t is an identifier that starts with an uppercase character;
- the type variables a_i ($0 \leq i \leq m$) must be different and start with a lowercase character;
- the pairs $r_i :\text{: } t_i$ ($1 \leq i \leq n$) determine the components of the record type. The field names r_i must be different and start with a lowercase character. The types t_i can use the type variables a_i .

Just like algebraic types, record types can also introduce existentially quantified type variables on the right-hand side of the record type. However, unlike algebraic types, their use can not be constrained by means of type classes. Hence, if you need to access these encapsulated values afterwards, you need to include function components within the record type definition.

Disambiguating Records. Within a program record field *names* are allowed to occur in several record types (the corresponding field *types* are allowed to be different). This helps you to choose proper field names, without worrying too much about their existence in other records. The consequence of this useful feature is that once in a while you need to explicit about the record value that is

created (in case of records with exactly the same set of record field names) and when using record field selectors (either in a pattern match or with the `.field` notation). Type constructor names are required to be unique within a program, hence they are used to disambiguate these cases.

- When creating a record value, you are obliged to give a value to each and every record field of that type. If a record has a field with a unique name, then it is clear which record type is intended. Only if two records have the same set of field names, you need to include the type constructor name t within the record value definition.
 $\dots \{ t \mid f_1 = e_1, \dots, f_n = e_n \} \dots$
- If a record pattern has at least one field with a unique name, then it is clear which record type is intended. The record pattern is disambiguated by including the type constructor name t in the pattern in an analogous way as described above when creating a record value, except that you do not need to mention all record fields and that the right hand sides of the fields are patterns rather than expressions:
 $\dots \{ t \mid f_1 = p_1, \dots, f_n = p_n \} \dots$
- If a record field selection $e.f$ uses a unique field name f , then it is clear which record type is intended. A record field selection can be disambiguated by including the type constructor name t as a field selector. Hence, $e.t.f$ states that field f of record type t must be used.

Record Updates. Record values are defined by enumerating each and every record field, along with a value. Example 5 shows that new record values can also be constructed from old record values. If r is a record (or an expression that yields a record value), then a new record value can be created by specifying only what record fields are different. The general format of such a *record update* is:

$$\{ r \& f_1 = e_1, \dots, f_n = e_n \}$$

This expression creates a new record value that is identical to r , except for the fields f_i that have values e_i ($0 < i \leq n$) respectively. A record field should occur at most once in this expression.

Synonym Types. Synonym types only introduce a new type constructor name for another type. The general formal of a type synonym declaration is:

$$:: t' a_1 \dots a_n ::= t$$

with the following properties:

- the type constructor t' is an identifier that starts with an uppercase character;
- the type variables a_i ($0 \leq i \leq n$) must be different and start with a lowercase character;
- the type t can be any type that uses the type variables a_i . However, a synonym type is not allowed to be recursive, either directly or indirectly.

Synonym types are useful for documentation purposes of your model types, as illustrated in Example 4. Although the name t' must be new, t' does not introduce a new type: it is completely exchangeable with any occurrence of t .

Strictness. In the signature of the basic task function `return` the first argument is provided with a *strictness annotation*, `!`. Recall that *iTask* is embedded in *Clean*, which is a *lazy* language. In a lazy language, computation is driven by *the need to produce a result*. As a simple example, consider the function `const` that does nothing but return its first argument:

```
const x y = x
```

There is absolutely no need for `const` to evaluate argument `y` to a value. However, argument `x` is returned by `const`, so its evaluation better produces a result or otherwise `const x y` won't produce a result either.

The more general, and more technical, way of phrasing this is the following. Suppose we have a function f that has a formal argument x . Let e be a diverging computation (it either takes infinitely long or aborts without producing a result). If $(f e)$ also diverges, then argument x is said to be strict in f . Note that this is a property of the function, and not of the argument. In case of `const`, it is no problem that argument `y` might be a diverging computation because it is not needed by `const` to compute its result. The consequence is that with respect to termination properties, it does not matter if strict function arguments are evaluated *before* the function is called. In many cases, this increases the performance of the application because you do not need to maintain suspended computations (due to lazy evaluation), but instead can evaluate them to a result and use that instead.

The strictness property of function arguments is expressed in the function signature by prefixing the argument that is strict in that function with the `!` annotation. In case of `const`, its signature is:

```
const :: !a b -> a
```

References

1. Kernighan, B., Ritchie, D.: The C Programming Language, 2nd edn. Prentice Hall, Englewood Cliffs (1988)
2. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, pp. 141–152. ACM Press (2007)
3. Lijnse, B., Plasmeijer, R.: iTasks 2: iTasks for end-users. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 36–54. Springer, Heidelberg (2010)
4. van der Heijden, M., Lijnse, B., Lucas, P.J.F., Heijdra, Y.F., Schermer, T.R.J.: Managing COPD exacerbations with telemedicine. In: Peleg, M., Lavrač, N., Combi, C. (eds.) AIME 2011. LNCS, vol. 6747, pp. 169–178. Springer, Heidelberg (2011)
5. Jansen, J., Lijnse, B., Plasmeijer, R.: Towards dynamic workflows for crisis management. In: French, S., Tomaszewski, B., Zobel, C. (eds.) Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management, ISCRAM 2010, Seattle, WA, USA, May 2010

6. Lijnse, B., Jansen, J., Nanne, R., Plasmeijer, R.: Capturing the Netherlands coast guard's SAR workflow with iTasks. In: Mendonca, D., Dugdale, J. (eds.) Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM 2011, Lisbon, Portugal. ISCRAM Association, May 2011
7. Meijer, E.: Server side web scripting in Haskell. *J. Funct. Program* **10**(1), 1–18 (2000)
8. Hanus, M.: High-level server side web scripting in Curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
9. Elsman, M., Larsen, K.F.: Typing XHTML web applications in ML. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 224–238. Springer, Heidelberg (2004)
10. Elsman, M., Hallenberg, N.: Web programming with SMLserver. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 74–91. Springer, Heidelberg (2002)
11. Thiemann, P.: WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In: Adsul, B., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, p. 192. Springer, Heidelberg (2002)
12. Serrano, M., Gallesio, E., Loitsch, F.: Hop, a language for programming the web 2.0. In: Proceedings of the 11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, Oregon, USA, 22–26 October 2006, pp. 975–985 (2006)
13. Loitsch, F., Serrano, M.: Hop client-side compilation. In: Proceedings of the 7th Symposium on Trends in Functional Programming, TFP 2007, New York, NY, USA, Interact, 2–4 April 2007, pp. 141–158 (2007)
14. Cooper, E., Lindley, S., Yallop, J.: Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
15. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: An idiom's guide to formlets. Technical report, The University of Edinburgh, UK (2007). <http://groups.inf.ed.ac.uk/links/papers/formlets-draft2007.pdf>
16. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology (2002)
17. Hemel, Z., Verhaaf, R., Visser, E.: WebWorkFlow: an object-oriented workflow modeling language for web applications. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 113–127. Springer, Heidelberg (2008)
18. Crandall, B., Klein, G., Hoffman, R.R.: Working Minds: A Practitioner's Guide to Cognitive Task Analysis. MIT Press, Cambridge (2006)
19. Plasmeijer, R., van Eekelen, M.: Clean language report (version 2.1) (2002). <http://clean.cs.ru.nl>