Task-Oriented Programming in a Pure Functional Language

Rinus Plasmeijer[1] Bas Lijnse[1,2] Steffen Michels[1]
Peter Achten[1] Pieter Koopman[1]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL, Nijmegen, The Netherlands

[2] Faculty of Military Sciences, Netherlands Defense Academy
P.O. Box 10000, 1780 CA, Den Helder, The Netherlands

{rinus, b.lijnse, s.michels, p.achten, pieter}@cs.ru.nl

Abstract

Task-Oriented Programming (TOP) is a novel programming paradigm for the construction of distributed systems where users work together on the internet. When multiple users collaborate, they need to interact with each other frequently. TOP supports the definition of tasks that react to the progress made by others. With TOP, complex multi-user interactions can be programmed in a declarative style just by defining the tasks that have to be accomplished, thus eliminating the need to worry about the implementation detail that commonly frustrates the development of applications for this domain. TOP builds on four core concepts: *tasks* that represent computations or work to do which have an observable value that may change over time, *data sharing* enabling tasks to observe each other while the work is in progress, *generic* type driven generation of *user interaction*, and special combinators for *sequential* and *parallel* task composition. The semantics of these core concepts is defined in this paper. As an example we present the iTask**3** framework, which embeds TOP in the functional programming language Clean.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.11 [*Software Engineering*]: Software Architectures—Languages; D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; H.5.3 [*Information Interfaces And Presentation*]: Group and Organization Interfaces—Computer-supported cooperative work; H.5.3 [*Information Interfaces And Presentation*]: Group and Organization Interfaces—Web-based interaction

Keywords Task-Oriented Programming; Clean;

1. Introduction

When humans and software systems collaborate to achieve a certain goal they interact with each other frequently and in various ways. Constructing software systems that support human tasks in a flexible way is hard. In order to do their work properly human beings need to be well informed about the progress made by others. We lack a formalism in which this aspect of work is specified at a high level of abstraction.

In this paper we introduce *Task-Oriented Programming (TOP)*, a novel programming paradigm to define interactive systems using *tasks* as the main abstraction. TOP provides advanced features for task collaboration. We choose tasks as *unit of application logic* for three reasons. First, they cover many phenomena that have to be dealt with when constructing systems in a natural and intuitive way. In daily life we use this notion to describe activities that have to be done by persons to achieve a certain goal. In computer systems, running processes are also commonly called tasks. On a programming language scale, a function, a remote procedure, a method, or a web service, can all be seen as tasks that can be executed. Second, in daily life it is common practice to split work into parallel and sequential sub-tasks and at the same time, during execution, not to be very strict about their termination behavior and production of results. Progress of work can be guaranteed even though some, or all, sub-tasks produce partial results. This contrasts with the usual concept of computational tasks that are interpreted as well-defined units of work that take some arguments, take some time to complete, and terminate with a result. Third, tasks abstract from the operational details of the work that they describe, assuming that the processor of the task knows how to perform it. The processor must deal with a plethora of issues: generate and handle interactive web pages, communicate with browsers, interact with web services in the cloud, interface with databases, and so on. Application logic is polluted with the management of side effects, the handling of complicated I/O like communication over the web, and the sharing of information with all users and system components. In this pandemonium of technical details one needs to read between the lines to figure out what a program intends to accomplish. Using tasks as abstraction prevents this. For these reasons, we conjecture and show that in the TOP paradigm specifying what the task *is* that needs to be done, and *how* it can be divided into simpler tasks is sufficient to create the desired application.

We present a foundation for Task-Oriented Programming in a pure functional language. We formalize the notion of tasks as abstract descriptions of interactive persistent units of work. Tasks produce typed, observable, results but have an abstract implementation. When observed by other tasks, a task can either have no (meaningful) value, have a value that is a temporary result that may change, or have a stable final result. We show how to program using this notion of tasks by defining a set of primitive tasks, a model for sharing data between tasks, and a set of operators for composing tasks. Because higher-order function composition provides powerful composition already, only a small set of operators is necessary. These are sequential composition, parallel composition, and the conversion of task results.

Most notably, we make the following contributions:

- We introduce *Task-Oriented Programming* as a paradigm for programming interactive multi-user systems composed of interacting tasks.
- We present *tasks* as abstract units of work with observable intermediate values and continuous access to shared information.
- We present combinators for composition and transformation of tasks and formally define their semantics.
- We demonstrate real-world TOP in Clean using the redesigned and extended iTask**3** framework.

The remainder of this paper is organized as follows: in Section 2 we informally explain the TOP paradigm by defining its concepts and a non-trivial example in Clean with the iTask**3** framework. In Section 3 we formalize the foundations of TOP component-wise: tasks and their evaluation, sharing information, user interaction, and sequential and parallel task composition. In Section 4 we reflect on the pragmatic issues that need to be dealt with in frameworks that facilitate real-world TOP programming. After a discussion of related work in Section 5, we conclude in Section 6 .

For readability, we use Clean$*$ (van Groningen et al., 2010) which is a dialect of Clean that adapts a number of Haskell language features. In this paper we deploy *curried function types* (Clean function types have arity), and the *unit type* `()`.

2. The TOP Paradigm

Task-Oriented Programming extends pure Functional Programming with a notion of *tasks* and operations for composing programs from tasks. Complex interactive multi-user systems are specified as decompositions of the tasks they aim to support.

2.1 TOP Concepts

Tasks: Tasks are abstract descriptions of interactive persistent units of work that have a typed value. When a task is *executed*, by a TOP framework, it has an opaque persistent state. Other tasks can observe the *current* value of a task in a carefully controlled way. When an executing task is observed, there are three possibilities:

1. **The task has no value observable for others:** This does not mean that no progress is made, but just means that no value of the right type can be produced that is ready for observation.
2. **The task has an unstable value:** When a task has an unstable value, it has a value of the correct type but this result may be different after handling an event. It is even possible that the next time the task is observed it has no value.
3. **The task has a stable value:** The task has a clear final result. This implies that if the task is observed again, it will always have the same value.

Tasks may be interactive. Such tasks process events and update their internal state. However, this event processing is abstracted from in Task-Oriented programs. The effects of events are only visible as changes in task results.

Many-to-many Communication with Shared Data: When multiple tasks are executed simultaneously, they may need to share data between them. How and where this data is stored however, is often completely irrelevant to the task. What matters is that the data is available and that it is shared. Thus, when one task modifies shared data, the other tasks can observe this change. In TOP we abstract from how and where data is stored and define *Shared Data Sources* (SDS) as typed abstract interfaces which can be read, written and updated atomically.

Generic Interaction: The smallest tasks into which an interactive system can be divided are single interactions, either between the system and its users or between the system and another system. Single interactions can be entering or updating some data, making a choice or just viewing some information. In TOP we abstract from how such interactions are realized unless it is essential to the task. A TOP framework *generates* user interfaces *generically* for any type of data used by tasks. This means that it is not necessary to design a user interface and program event handling just to enter or view some information. It is possible to specify interactions in more detail, but it is not needed to get a working program.

Task Composition: TOP introduces the notion of tasks as first-class values, but also leverages first-class functions from pure functional programming. This means that only a small carefully designed set of core combinator functions is needed from which complex patterns can be constructed.

1. **Sequential composition:** TOP uses *dynamic* sequential composition. Because task values are observable, sequential compositions are not defined by blindly executing one task after another. They are defined by composing an initial task with a set of functions that *compute* possible next steps from the observed value of the initial task.
2. **Parallel composition:** Parallel composition is defined as executing a set of tasks simultaneously. Tasks in a parallel set have read-only access to a shared data source that reflects the current values of all sibling tasks in the set. In this way tasks can monitor each other's progress and react accordingly.
3. **Value transformation:** Task domains can be converted by pure functions in order to combine tasks in a type consistent way.

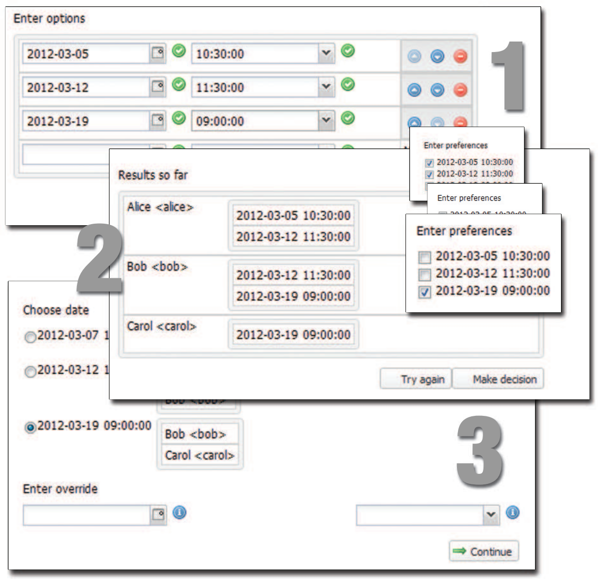
2.2 An Example of TOP in Clean

To illustrate Task-Oriented Programming in practice, we present a non-trivial example that uses the novel iTask**3** framework. In the example, one specific user, the *coordinator*, has to collaborate with an arbitrary number of users to find a meeting date and time. Figure 1 displays that this task consists of three sub-tasks. This figure consists of actual screenshots of the user interfaces generated by the iTask**3** framework.

In sub-task *one*, the coordinator creates a number of date-time pairs. While doing so, he or she can rearrange their order, insert new date-time pairs, or remove them. Once satisfied, the coordinator confirms the work by pressing the *Continue* button, and *steps* into sub-task *two*.

This sub-task *two* consists of a number of tasks running in parallel. The users (Alice, Bob, and Carol in this example) are all asked to make a selection of the proposed date-time pairs (the *Enter preferences* windows). Meanwhile, the coordinator can monitor and follow the selections being made (the *Results so far* window). At any time, the coordinator can either choose to restart the entire task all over again, by pressing the *Try again* button. He or she can also select a date-time pair that is suitable for (the majority of) all users by pressing the *Make decision* button. In the first case, they *step* into the plan meeting task afresh, and in the latter case, they *step* into sub-task *three*.

In sub-task *three* the system provides the coordinator with an overview of available users per date-time pair, thus helping him or her to make a good decision. The coordinator can also decide not to pick any of the candidate date-time pairs and override them with a



Enter options

2012-03-05	10:30:00
2012-03-12	11:30:00
2012-03-19	09:00:00

Enter preferences
☑ 2012-03-05 10:30:00
☑ 2012-03-12 11:30:00

Results so far

Alice <alice>	2012-03-05 10:30:00
	2012-03-12 11:30:00
Bob <bob>	2012-03-12 11:30:00
	2012-03-19 09:00:00
Carol <carol>	2012-03-19 09:00:00

Try again · Make decision

Enter preferences
☐ 2012-03-05 10:30:00
☐ 2012-03-12 11:30:00
☑ 2012-03-19 09:00:00

Choose date
○ 2012-03-07
○ 2012-03-12
● 2012-03-19 09:00:00 Bob <bob>, Carol <carol>

Enter override

Continue

Figure 1. Selecting possible dates for a meeting

proposed alternative. Once satisfied with a choice, the coordinator terminates the entire task by pressing *Continue*, and returns a *stable* date-time value.

In the remainder of this section we show how to specify this example in a Task-Oriented way. Figure 2 displays the complete specification. It contains TOP-notions explained in detail further on in this paper. The key point of this example is to show how Task-Oriented Programming aids to create a specification that closely matches the description that is shown above. The semantics of the used concepts are defined in Section 3.

The entire task of the coordinator is described by `planMeeting`. Its type (line 1) expresses that given a list of users, it is a task that produces a date-time pair. `User` and `DateTime` are predefined data types. `User` represents a registered user. `DateTime` is just a pair of `Date` (day-month-year triplet) and `Time` (hours-minutes-seconds triplet) which also happen to be predefined.

As discussed, the main structure of `planMeeting` consists of three subsequent sub-tasks (lines 2-4), which are glued together by means of the *step* combinator `>>*`. The second argument of `>>*` enumerates the potential subsequent *task steps* that can be stepped into while the first argument task is in progress. Hence, the first sub-task, `enterDateTimeOptions`, is followed by `askPreferences`, which in turn is followed by *either* `tryAgain` *or* `decide`. Entering user information (performed by `enterDateTimeOptions`, `select`, and `pick`) is an example of a task that may or may not have a *task value*. This depends on the input provided by the user. The potential task steps which can follow can observe the task value and define whether or not sufficient information is provided to step into the next task. In case of the transition from the first sub-task to the second sub-task, this requires an *action* from the coordinator (line 12). This is only sensible if the previous task has a task value, which is tested by the predicate `hasValue`. In that case, the current task value is retrieved (`getValue`) and used to step into the next sub-task, which is to ask all users to choose preferred date-time pairs.

The *observable* task value is accessible in the step combinator to determine the next task steps chosen. The task value and its access functions are straightforward: `hasValue` tests for the `Val` data constructor, and `getValue` returns that value if present:

```
:: Value a    = NoVal    | Val a Stability
```

```
planMeeting :: [User] → Task DateTime                                       1
planMeeting users =   enterDateTimeOptions                                  2
                  >>* [askPreferences users]                                3
                  >>* [tryAgain users, decide]                              4
                                                                            5
enterDateTimeOptions :: Task [DateTime]                                     6
enterDateTimeOptions = enterInformation "Enter options" []                  7
                                                                            8
askPreferences :: [User]                                                    9
               → TaskStep [DateTime] [(User,[DateTime])]                    10
askPreferences users                                                        11
  = OnAction (Action "Continue") hasValue (ask users o getValue)            12
                                                                            13
ask :: [User] → [DateTime] → Task [(User,[DateTime])]                       14
ask users options                                                           15
  = parallel "Collect possibilities"                                        16
    [ (Embedded, monitor)                                                   17
    :[(Detached (worker u),select u options) \\ u←users]                    18
    ]                                                                       19
    @ λanswers → [a \\ (_,Val a _)←answers]                                 20
                                                                            21
monitor :: ParallelTask a | iTask a                                         22
monitor all_results                                                         23
  = viewSharedInformation "Results so far" []                               24
       (mapRead tl (taskListState all_results))                             25
    @? λ_ → NoVal                                                           26
                                                                            27
select :: User → [DateTime] → ParallelTask (User,[DateTime])                28
select user options _                                                       29
  = enterMultipleChoice "Enter preferences" [] options                      30
    @ λchoice → (user,choice)                                               31
                                                                            32
tryAgain :: [User] → TaskStep [(User,[DateTime])] DateTime                  33
tryAgain users                                                              34
  = OnAction (Action "Try again") (const True)                              35
             (const (planMeeting users))                                    36
                                                                            37
decide :: TaskStep [(User,[DateTime])] DateTime                             38
decide                                                                      39
  = OnAction (Action "Make decision") hasValue (pick o getValue)            40
                                                                            41
pick :: [(User,[DateTime])] → Task DateTime                                 42
pick user_dates                                                             43
  =   (enterChoice "Choose date" [] (transpose user_dates) @ fst)           44
     -||-                                                                   45
      (enterInformation "Enter override" [])                                46
  >>* [OnAction (Action "Continue") hasValue (return o getValue)]           47
```

Figure 2. Complete task specification of the `planMeeting` example

```
:: Stability   = Unstable | Stable

hasValue :: Value a → Bool
hasValue (Val _ _) = True
hasValue _         = False

getValue :: Value a → a
getValue (Val a _) = a
```

Tasks with `Stable` values are terminated and can no longer produce a different task value. Hence *task values* are first-class citizens in Task-Oriented Programming. Two task transformer functions provide access: `@?` alters the task value of the preceding task, and `@` is similar, but only if a `Val` is present:

```
(@?) infixl 1 :: Task a → (Value a → Value b)
                                        → Task b | iTask a & iTask b
(@)  infixl 1 :: Task a → (a → b) → Task b | iTask a & iTask b
```

The second sub-task of the coordinator is to ask all users *in parallel* to make a selection of the created date-time pairs. In addition, the coordinator *constantly monitors* their progress. Parallel composition of tasks is defined with the `parallel` combinator. It is used explicitly in the `ask` task, and implicitly (by means of the derived parallel-or combinator `-||-` that provides a shorter notation for the common case of choice between two alternative tasks) in the `pick` task. Parallel composition is a core concept in Task-Oriented Programming. The second argument of `parallel` enumerates the sub-tasks that need to be evaluated in parallel. The progress is *shared* between all sub-tasks. Relevant to the example is the function `taskListState`, which transforms this shared state to share the current *task values*. This is used by the `monitor` task (lines 24-25) to create a view on the current task values of the users. The `monitor` task uses `@?` to explicitly state that its task value never contains a concrete value. These can be provided only by the `select` sub-tasks. They offer their user the means to make a multiple-choice of the provided date-time pairs, and use `@` to attach the user to identify who made that specific selection (lines 30-31).

Finally, the last sub-task can be stepped into when the coordinator either decides to start all over again (lines 34-36) or pick a value (lines 39-40). The first action step is always valid (`const True`, line 35) and the second action step only when the previous task actually has a value (line 40). The derived combinator `-||-` evaluates its two task arguments in parallel, and has a task value that is either stable (if one or both sub-tasks have one) or unstable (if one or both have one) or none. Hence, the action step can only occur when the coordinator has either selected one of the suggested date-time pairs or chosen to override them.

This example demonstrates how a TOP approach can lead to a concise specification in which tasks are glued together and overall progress can be achieved even though the tasks themselves might not terminate or consume too much time.

3. A Formal Foundation of TOP

In this section we introduce and semantically define the core concepts of Task-Oriented Programming. These are *task values*, *tasks* and their *evaluation* (Section 3.1), *many-to-many* communication (Section 3.2), *user-interaction* (Section 3.3), *sequential* task composition (Section 3.4), and *parallel* task composition (Section 3.5).

Except for Section 3.1, every section has the same structure: we first introduce the core concept and illustrate it by means of the iTask**3** system, and then formally define the operational semantics using *rewrite semantics*. The rewrite rules are specified in Clean$*$. Such a way of formal specification of semantics is somewhat unusual, but this approach has certain advantages over traditional ones (Koopman et al., 2009). The specification is well-defined, concise, compositional, executable, and can express even complicated language constructs as the ones introduced in this paper. Since we are dealing with constructs embedded in a functional language it is an advantage to describe their semantics as pure functions in a functional language as well. We have experimented with several alternative definitions which can easily introduce errors that remain overlooked. It is an advantage that the descriptions are checked by the compiler and that we have been able to test their correct working by applying it to concrete examples. Furthermore, the formal semantics is very suited and also used as blue print for the actual implementation and can serve as a reference implementation for implementations in other programming languages as well. In order to distinguish semantic definitions from iTask**3** API and code snippets, we display semantic definitions as *framed* verbatim text, and iTask**3** fragments as *unframed* verbatim text.

3.1 Tasks and their Evaluation

In this section we define *task results* and *task values* (Section 3.1.1), *tasks* (Section 3.1.2), their *evaluation* (Section 3.1.3), and a number of *task transformer functions* (Section 3.1.4).

3.1.1 Task Results and Task Values

A task of type `Task a` is a description of work which progress can be inspected by a *task value* of type `Value a` (Section 2.2). Tasks handle events. Events have a time stamp, for which we use an increasing counter, making it possible to determine the temporal order of events. The *task result* of handling an event may be a new task value. Semantically, we extend the task value with the time stamp of the event that caused the creation of that task value. Tasks that run into an exceptional situation have as task result an exception value instead of a task value. The domains of task results and task values capture these situations:

```
:: TaskResult a =      ValRes TimeStamp (Value a)
                |  ∃e: ExcRes e & iTask e
:: TimeStamp    :== Int
:: Value a      =   NoVal    | Val a Stability
:: Stability    =   Unstable | Stable
```

The task value of a task result can be in three different states: there can be no value at all (`NoVal`), there can be an `Unstable` value which may vary over time, or the value is `Stable` and fixed. To illustrate, consider the task of writing a paper p. At time t_0 you have no paper at all (`ValRes` t_0 `NoVal`). After a while, at time t_1 there may be a draft paper p_1, which is updated many times at subsequent time stamps $t_2 \ldots t_n$ with draft papers $p_2 \ldots p_n$ (`ValRes` t_i (`Val` p_i `Unstable`)). You may even start all over again (`ValRes` t_{n+1} `NoVal`). At a certain point in time, t_{n+k} say, when you decide that the paper is finished the task has result `ValRes` t_{n+k} (`Val` p_{n+k} `Stable`) meaning that the paper can no longer be altered.

Some tasks never produce a stable value. Examples are the interactive tasks (`enterInformation`, `viewSharedInformation`, `enterChoice`, `enterMultipleChoice`) that were used in Section 2.2: a user can create, change or delete a value as many times as wanted. Typical examples of tasks that produce a `Stable` value are ordinary functions, or system and web service calls. A task can raise an exception value (`ExcRes e`) in case it is known that it can no longer produce a meaningful value (for instance when a call to a web service turns out to be unavailable). Any value can be thrown as exception and inspected by an exception handler (Section 3.4), using existential quantification `∃e` and the type class context restriction `& iTask e`. Tasks with stable values or exception values have no visualization but memorize their task result forever. The other tasks require a visualization to support further interaction with the user.

3.1.2 Tasks

Semantically, we define a task to be a state transforming function that reacts to an event, rewrites itself to a reduct, and accumulates responses to users:

```
:: Task a   :== Event → *State → *(Reduct a, Reponses, *State)
:: Event      = RefreshEvent
              | EditEvent   TaskNo Dynamic   // Section 3.3.1
              | ActionEvent TaskNo Action    // Section 3.4.1
:: *State     = { taskNo    :: TaskNo
                , timeStamp :: TimeStamp     // Section 3.3.1
                , mem       :: [Dynamic]     // Section 3.2.1
                , world     :: *World
                }
:: Reduct a   = Reduct (TaskResult a) (Task a)
:: TaskNo   :== Int
:: Responses:== [(TaskNo, Response)]         // Section 3.3.1
```

We distinguish three sorts of events: a `RefreshEvent`, e.g. when an user wants to refresh a web page, an `EditEvent`, e.g. a new value that is committed intended for an interactive task (Section 3.3), and an `ActionEvent` which is used to tell the step combinator which task to do next (Section 3.4). The latter two cases identify the task that is required to handle the event. The interactive task and step task are provided with a fresh identification value and current time stamp, using the semantic function `newTask`:

```
newTask :: (TaskNo → TimeStamp → Task a) → Task a
newTask ta ev st=:{taskNo = no, timeStamp = t}
  = ta no t ev {st & taskNo = no+1}
```

Fresh task identification numbers are generated by keeping track of the latest assigned number in the `State`. The `State` extends the external environment of type `*World` with internal administration and is passed around in a single-threaded way which is enforced by the uniqueness attribute $*$.

The reduct contains both the *latest task result* and a *continuation* of type `Task a`, which is the remaining part of the work that still has to be done. This continuation can be further evaluated in the future when the next event arrives.

The responses collect all responses of all subtasks the task is composed of. They are used to update every client with the proper information about the latest state of affairs. A client can use this information to adjust the page in the browser or in an app.

In the remainder of this paper we define semantic task functions for the core basic tasks and task combinators, thus explaining how these elements rewrite to the next reduct.

3.1.3 Task Evaluation

A TOP application consists of one top level task, the main task, which has to be evaluated. The work continues until either an exception escapes handling, or the work at hand has obtained a stable task value.

```
evaluateTask :: Task a → *World → *(Maybe a, *World) | iTask a  1
evaluateTask ta world                                            2
# st      = {taskNo = 0, timeStamp = 0, mem = [], world = world} 3
# (ma,st) = rewrite ta st                                        4
= (ma,st.world )                                                 5
                                                                 6
rewrite :: Task a → *State → *(Maybe a, *State) | iTask a        7
rewrite ta st=:{world}                                           8
# (ev,world) = getNextEvent   world                              9
# (t, world) = getCurrentTime world                              10
# st         = {st & timeStamp = t, world = world}               11
# (Reduct res nta, rsp, st) = ta ev st                           12
= case res of                                                    13
   ValRes _ (Val a Stable) → (Just a, st)                        14
   ExcRes _ → (Nothing, st)                                      15
   _        → rewrite nta                                        16
                {st & world = informClients rsp st.world}        17
```

In Clean($*$), passing around multiple unique environments explicitly, such as `st` (`:: *State`) and `world` (`:: *World`), is syntactically supported by means of the non-recursive `#`-let definitions. The main task is recursively rewritten by the function `rewrite`. Rewriting is triggered by an event. We abstract from the behaviour of clients and just assume that they send events and handle responses. We assume that all events are collected in a queue. In `getNextEvent` (line 9) the next event is fetched from this queue. If there are no events, the system waits until there is one. The current time is stored in the state (lines 10-11) to ensure that all tasks which update their value in this rewrite round, will get the same time stamp. Hereafter (line 12), the main task `ta` is evaluated given the event and current state. Any sub-task defined in the main task is a task as well, and can be evaluated in the same way: just apply the corresponding task function to the current event and the current state. Rewriting stops when the main task has delivered a stable value (line 14), or an uncaught exception is raised (line 15). Otherwise, the main task is not finished yet, and the continuation task returned in the reduct defines the remaining work which has to be done. First the accumulated responses are sent to the clients (`informClients`, line 17) to inform them about the latest state-of-affairs. We abstract in the semantics from the way this is done. Rewriting continues with the continuation `nta` and the updated state.

3.1.4 Utility Functions for Converting Tasks

The semantic function `stable`, when applied to a time stamp `t` and value `va`, defines a task that has reached a stable value:

```
stable :: TimeStamp → a → Task a
stable t va _ st
= (Reduct (ValRes t (Val va Stable)) (stable t va),[],st)
```

Notice that the continuation of the task `stable t va` in the reduct is exactly the same function `stable t va`. It is a kind of fixed point task, which, whenever it is evaluated in some future, always returns the same reduct (value and continuation). With this semantic function, we can define the semantic function of the core task `return`:

```
return :: a → Task a
return va ev st=:{timeStamp = t} = stable t va ev st
```

Here, `return` has a similar role as the `return` function in a monadic setting: it lifts an arbitrary value `va` of type `a` to the task domain.

Raising an exception is similar, except that the task result is always an exception value:

```
throw :: e → Task e | iTask e
throw e _ st = (Reduct (ExcRes e) (throw e),[],st)
```

With operator `@?` and a function `f` of type `Value a → Value b` a task `ta` of type `Task a` can be converted to a task of type `Task b`:

```
(@?) infixl 1 :: Task a → (Value a → Value b)                    1
                              → Task b | iTask a & iTask b        2
(@?) ta f ev st                                                   3
= case ta ev st of                                                4
   (Reduct (ValRes t aval) nta,rsp,nst)                           5
   → case f aval of                                               6
       Val b Stable                                               7
           → stable t b ev nst                                    8
       bval → (Reduct (ValRes t bval) (nta @? f),rsp,nst)         9
   (Reduct (ExcRes e) _,_,nst)                                    10
   → throw e ev nst                                               11
                                                                  12
(@) infixl 1 :: Task a → (a → b) → Task b | iTask a & iTask b     13
(@) ta f = t @? λaval → case aval of                              14
                         NoVal   = NoVal                          15
                         Val a s = Val (f a) s                    16
```

First the task `ta` is evaluated (line 3). Exceptions raised by `ta` are simply propagated (lines 10-11). The resulting task value, if any, is converted by function `f`. If this results in a stable value, then the entire task becomes stable with the current time stamp (lines 7-8). Notice that this has as consequence that the original task `ta` is no longer needed. If the result is not stable, the original task may change its value over time, and we need to apply the conversion function to values produced in the future as well. Therefore, the current result `bval` of the conversion is stored in the reduct with the continuation `nta @? f` which takes care of the conversion of the new task values produced in the future (line 8). The derived operator `@` uses `@?` to transform task values only when a concrete value is present.

3.2 Many-to-many Communication

For collaborating tasks it is important to keep each other up-to-date with the latest developments while the work is going on. Hence we need to be able to share information between tasks and support many-to-many communication.

How and where this data is stored, is completely irrelevant to the tasks. What matters is that the data is available and that it is shared. To achieve this abstraction we use the concept of multi-purpose Shared Data Sources (SDS) (Michels and Plasmeijer, 2012). SDSs are typed, abstract interfaces which can be read, written and updated atomically.

A SDS can represent a shared file, a shared structured database, reveal the current users of a system, or it can be a physical entity, like the current time or temperature. In general, a SDS abstracts from any shared entity that holds a value that varies over time.

```
:: RWShared r w

:: ROShared r :== RWShared r  ()
:: WOShared w :== RWShared () w
:: Shared a   :== RWShared a  a
```

A SDS has abstract type `RWShared r w`. *Reading* its current value returns a value of type `r`, and *writing* is done with a new value of type `w`. Read-only shared objects (`ROShared r`) only support reading as type `r`, write-only shared objects (`WOShared w`) only support writing as type `w`, and `Shared a` objects demand that the read and write values have the same type `a`.

As an example, we show a few shares that are offered by the iTask**3** system to create SDSs:

```
sharedFile   :: Path → a → Shared a | iTask a
currentTime  :: ROShared  Time
currentUsers :: ROShared [User]
```

With (`sharedFile fname content`) a task is described that associates a file identified by `fname` with an initial value of type `a`. A task gains access to the current time and registered users with the tasks `currentTime` and `currentUsers`.

SDSs provide many-to-many communication both between tasks and other applications. We make a difference between external and internal SDSs. External SDSs are abstractions of external objects such as files and databases and can be accessed anywhere in the application. For the internal communication between tasks only, one can create a shared memory SDS of type `Shared a` which has a limited scope. A task `ta` can be parameterized with a freshly created shared memory SDS `sa` of type `Shared a` that has some initial value `va` using the combinator `withShared va (λsa → ta)`:

```
withShared :: a → (Shared a → Task b) → Task b | iTask a
```

In this way, a shared memory is created which can only be accessed by the sub-tasks defined within `ta`. For an example of its use, see Section 4.

To write a value to a SDS, one can connect a task `ta` with a SDS `s` using a function `f` with the combinator `ta @> (f,s)`:

```
(@>) infixl 1 :: Task a
             → (Value a → r → Maybe w, RWShared r w)
             → Task a | iTask a
```

This enforces `f` to be repeatedly applied to the current task value of `ta` (if any) and the currently read value of `s`, the result of which is the new value (if any) that is written to `s`. The combinators `withShared` and `@>` are defined in Section 3.2.1.

SDSs integrate smoothly with interactive tasks. For every basic interactive task (such as `enterChoice` and `enterMultipleChoice`) a *shared* version (such as `enterSharedChoice` and `enterSharedMultipleChoice`) is provided that expects a SDS instead of a common value. This is discussed in Section 3.3 in more detail. In this way tasks can monitor and alter SDSs.

3.2.1 Semantics of Memory Shared between Tasks

To explain the semantics of SDSs, we restrict ourselves to their use for offering shared memory between (parallel) tasks. These SDSs cannot be accessed by external applications. Hence the semantic definition does not need to handle concurrency and atomicity issues: there is only one `rewrite` function (Section 3.1.3) that handles rewriting of all tasks defined in an application.

Shared memory cells are stored in the `State`, in record field `mem` of type `[Dynamic]`. Each SDS memory cell can be used to store a value of arbitrary type, hence `mem` is modeled as a heterogeneous list using Clean's built-in dynamic types (Vervoort and Plasmeijer, 2003; van Weelden, 2007). Any shared value of any type can be stored in a value of type `Dynamic`, together with a representation of its type (using the function `serialize :: a → Dynamic | iTask a`). It can be fetched from this store any time later, using a dynamic type pattern match that guarantees that no type errors can occur at runtime (using the function `de_serialize :: Dynamic → a | iTask a`). We define a SDS creation function, and two functions to update a SDS:

```
:: RWShared r w = { get :: *State → *(r,*State)                      1
                  , set :: w → *State → *State                       2
                  }                                                  3
                                                                     4
createShared :: a → *State → *(Shared a,*State) | iTask a            5
createShared a st=:{mem}                                             6
= ({get = get,set = set},{st & mem = mem ++ [serialize a]})          7
where                                                                8
  idx           = length mem                                         9
  get   st=:{mem} = (de_serialize (mem!!idx),st)                     10
  set a st=:{mem} = {st & mem = updateAt idx (serialize a) mem}      11
                                                                     12
updateShared :: (r → w) → RWShared r w → *State → *(w,*State)        13
updateShared f sh_a st                                               14
# (rv,st)    = sh_a.get st                                           15
# wv         = f rv                                                  16
= (wv,sh_a.set wv st)                                                17
                                                                     18
updateMaybeShared :: (r → Maybe w) → RWShared r w → *State           19
                                        → *(Maybe w,*State)          20
updateMaybeShared f sh_rw st                                         21
# (readv,st) = sh_rw.get st                                          22
= case f readv of                                                    23
    Nothing  = (Nothing,st)                                          24
    Just wv  = (Just wv,sh_rw.set wv st)                             25
```

A SDS is represented by two access functions `get` and `set` that retrieve and store the required information from and to the state. Creating a shared value with `createShared` appends an initial serialized value to the list of memory locations (line 7), and returns two dedicated `get` and `set` functions that access this new memory location. The SDS update functions both obtain the current read value of the SDS argument (line 15 and 22). However, `updateShared` always updates the SDS with a new value, and `updateMaybeShared` does this only if the argument function actually produces a new value. With these internal functions, we can define `withShared` and `@>`:

```
withShared :: a → (Shared a → Task b) → Task b | iTask a
withShared va tfun ev st
# (sh_a,st) = createShared va st
= tfun sh_a ev st

(@>) infixl 1 :: Task a
             → (Value a → r → Maybe w, RWShared r w)
             → Task a | iTask a
(@>) ta (f,sh_rw) = update NoVal ta
where
 update otval ta ev st
 = case ta ev st of
   (Reduct (ExcRes e) nta, _, nst)
      → throw e ev nst
   (Reduct (ValRes ts ntval) nta,rsp,nst)
      → ( Reduct (ValRes ts ntval) (update ntval nta)
         , rsp
```

```
     , if (ntval==otval)                                        18
          nst                                                   19
          (snd (updateMaybeShared (f ntval) sh_rw nst))         20
     )                                                          21
```

withShared creates a fresh SDS for its argument task function and applies it to obtain the proper task. The combinator @> memorizes the previous task value (initially NoVal) and the current task continuation (initially the task argument ta) (line 9 and 16). As usual, at each event the current task continuation is evaluated (line 12). Exceptions are propagated (lines 13-14). The only difference is that if the new task value ntval is different from the memorized task value otval, then the SDS is updated using the argument function of @> and the local function updateSDS (line 18). This function only updates the SDS if a new value is computed (lines 23 and 25). In this way unnecessary updates of shared data is avoided. Because @> keeps checking the SDS using the most recent task value, this leads to reactive behavior: every time the watched task is changing its value, the shared memory also gets updated conditionally, as described above.

3.3 User Interaction

In Task-Oriented Programming user-interactions are defined as tasks that allow a user to enter and modify a visualized value of some type. Such an interactive task is called an *editor*. The type of the value to be edited plays a central role. By using type indexed generic functions (Alimarine, 2005; Hinze, 2000) this visualization is generated fully automatically for any (first order) type. This way one can focus on defining tasks, without having to deal with the complexities of web protocols and formats.

Interaction tasks follow a model-view pattern where the value of the task is the model and the visualization is the view. Events in the view are processed by the TOP framework to update the model. Conversely, when the model changes the view is updated automatically by the TOP framework.

Interaction tasks are all alike, yet different. In this section we define the semantics of one core editor task (Section 3.3.1). However, to improve readability TOP frameworks can offer a range of predefined interaction tasks derived from this core editor. A few examples from the iTask**3** framework are:

```
enterInformation        :: d → [EnterOpt m]
                          → Task m      | descr d & iTask m
updateInformation       :: d → [UpdateOpt m m] → m
                          → Task m      | descr d & iTask m
viewInformation         :: d → [ViewOpt m] → m
                          → Task m      | descr d & iTask m
updateSharedInformation :: d → [UpdateOpt r w] → RWShared r w
                          → Task w      | descr d
                                        & iTask r & iTask w
viewSharedInformation   :: d → [ViewOpt r] → RWShared r w
                          → Task r      | descr d & iTask r
```

With enterInformation an editor for type m is created, no initial value needs to be given. The update-editor variants allow editing of a given local, respectively shared, value. The view-editor variants only display the value of a given local, or shared, value. There are many more similar editor functions predefined in the library, with names like enterChoice, enterSharedChoice, updateChoice, updateSharedChoice, enterSharedMultipleChoice, and so on.

The overloaded argument d of class descr in these tasks is description of the task. This can be a simple string, or a more elaborate description. Although the generated view is certainly good enough for rapid prototyping, more fine-grained control is sometimes desirable. Therefore, the EnterOpt, UpdateOpt and ViewOpt arguments provide hooks for fine-tuning interactions.

```
:: ViewOpt a     =∃v: ViewWith   (a → v)            & iTask v
:: EnterOpt a    =∃v: EnterWith          (v → a)    & iTask v
:: UpdateOpt a b =∃v: UpdateWith (a → v) (a v → b)  & iTask v
```

By defining a mapping, a different type v can be used to view, enter or update information. In Section 4 we discuss in more detail how this and other pragmatic issues are dealt with.

3.3.1 Semantics of a Task Editor

The iTask**3** library provides many different editor task functions because this clarifies in the task descriptions what kind of interaction is required, and aids in creating the desired user interface. However, both in the implementation and the semantics all editor task variants can be created and handled by one single function. To understand how it works we restrict ourselves to a simplified version in which we omit the view list details because these are just trivial mapping functions. Before we discuss this function edit we first have a look at the use of Events and Responses. Due to the model-view nature of editor tasks, every user manipulation of an editor task of a value of type a can be expressed as sending a new value new of type a from the client to the server. If we wrap this value-type pair into a Dynamic and include the task identification number, no say, then this amounts to the (EditEvent no (**dynamic** new :: a)) event. The unique task number is used to map a task described in the code to the corresponding interactive view generated in the client, and is used to label the events and corresponding responses.

The responses of the server tell the client what interface should be rendered to the user.

```
:: Response        = EditorResponse EditorResponse
                   | ActionResponse ActionResponse // Section 3.4.1
:: EditorResponse  = { description :: String
                     , editValue   :: EditValue
                     , editing     :: EditMode
                     }
:: EditValue      :== (LocalVal, SharedVal)
:: LocalVal       :== Dynamic
:: SharedVal      :== Dynamic
:: EditMode        = Editing | Displaying
```

The response to an editor task executed on a client informs the client about the latest state of the editor (EditorResponse) and contains, in serialized form, the current local value to edit and a shared value to show. With these Events and Responses, we can define the semantics of the editor task combinator which updates a local value of type l while displaying the latest value r stored in an SDS of type RWShared r w.

```
edit :: String → l → RWShared r w → (l → r → Maybe a)
                               → Task a | iTask l & iTask r
edit descr lv sh_rw cv = newTask (edit1 lv)
where
  edit1 lv tn t ev st
  # (nt,nlv) = case ev of
               EditEvent tid dyn
                 → if (tid==tn)
                      (st.timeStamp,de_serialize dyn)
                      (t,lv)
                 _ →  (t,lv)
  # (sr,st)  = sh_rw.get st
  = ( Reduct (ValRes nt (toValue (cv nlv sr))) (edit1 nlv tn nt)
    , [(tn,EditorResponse
          { description = descr
          , editing     = Editing
          , editValue   = (serialize nlv, serialize sr)
          }
       )]
    , st
    )
  where
    toValue :: Maybe a → Value a
    toValue (Just a) = Val a Unstable                           24
    toValue Nothing  = NoVal                                    25
```

The `edit` function, and its continuation in the reduct (line 13), is defined in terms of `edit1` that keeps track of the latest local value edited, the unique task number given to this interactive task, and the time the latest modification has been made.

Editor tasks always have an unstable value (if any). They return a response containing the latest information on the state of the editor (lines 14-19 and 23-25). It includes the latest value of the data stored in shared memory (line 12) which might have been changed by some other task (e.g. using the `@>` operator). Only when the received event is an edit event intended for this editor (the task numbers match), the local value is updated with the new value received from the client (line 9). The new task value is computed using the most recent local and shared value (line 13).

3.4 Sequential Tasks

Once a task is started, it stays alive until it is no longer needed. Its value, which might change over time, can be inspected while the work is going on in order to decide whether or not to step to a next task. The task *step* operator `>>*` does exactly this.

```
(>>*) infixl 1 :: Task a → [TaskStep a b] → Task b | iTask a
                                                    & iTask b
:: TaskStep a b
   = OnAction Action (Predicate a) (NextTask a b)
   | OnValue         (Predicate a) (NextTask a b)
   | ∃e: OnException (e → Task b) & iTask e

:: Predicate a :== Value a → Bool
:: NextTask a b:== Value a → Task b

:: Action = Action String | ActionOk | ActionCancel | ...
```

The step operator is similar to an ordinary monadic “bind-operator” in the sense that it defines a sequence between two tasks. The first operand, a task of type `Task a`, is evaluated. Its current task value can be inspected to decide whether the next task can be stepped into. If so, the evaluation of the first task is abandoned and the application proceeds with the chosen task step. The step operator can offer several tasks to continue with in the list, but only one task step can be stepped into.

There are three categories of task steps: those that require the user to actively select an action (`OnAction`), those that inspect the current task value (if any) (`OnValue`), and those that handle exceptions (`OnException`). `OnAction` task steps are labeled with an `Action` that is presented to the user as a button or menu item. For frequently used action names such as `Ok` and `Cancel`, the `Action` data type enumerates a number of special combinators to enable the client to use special icons. The predicate determines which action steps are available at all. Selection of an action by the user causes the corresponding alternative to be continued with. `OnValue` task steps inspect the current task value to determine whether or not a task step can be performed. Finally, `OnException` task steps handle an exception only if their argument function matches the type of the exception. Uncaught exceptions are propagated by `>>*`.

It sometimes can be the case that none of the candidate task steps can be chosen. However, task values change over time, hence also the candidates that can be chosen change over time.

We illustrate the use of `>>*` with two examples.

```
palindrome :: Task (Maybe String)
palindrome =   enterInformation "Enter a palindrome" []
          >>* [ OnAction  ActionOk     ifPalindrome
                                 (return o Just o getValue)
              , OnAction  ActionCancel (const True)
                                 (const (return Nothing))
              ]
```

The `palindrome` task prompts the user to enter a palindrome. As usual, the user can enter a string and change it over time. With `>>*` two possible action task steps are added. The user can choose action `Ok`, but only when the entered string is indeed a palindrome. If `Ok` is chosen, `Just p` is returned, where `p` is the entered and checked palindrome. At any time, the user can choose `Cancel`, and the task returns `Nothing`.

In the second example we implement a traditional monadic bind operator `>>=` to demonstrate the general nature of `>>*`:

```
(>>=) infixl 1 :: Task a → (a → Task b) → Task b | iTask a
                                                  & iTask b
(>>=) ta atb = ta >>* [OnValue isStable (atb o getValue)]
```

Task evaluation starts with the first argument `ta`. *Only* when this task produces a *stable* value `a`, evaluation continues with `atb a`. For this reason, `>>=` is less suited in the domain of tasks that may not produce a stable result.

3.4.1 Semantics of the Step Combinator

First we finalize the details of `ActionResponses`. The client is informed by `>>*` about the current set of actions and whether they are enabled or disabled. This information is collected in the `ActionResponse` list and added to the response accumulator.

```
:: ActionResponse :== [(Action, Enabled)]
:: Enabled        :== Bool
```

The client may react by sending an action event `ActionEvent taskno action` telling which action is triggered by the user.

The complete semantic definition of `>>*` is given in Figure 3. It is rather long because it needs to handle all `TaskStep` cases and prioritize them properly. However, each of these cases is rather straightforward. The step combinator is handled by `step1` which memorizes the current task description in its first argument (initially task `ta`, line 3, and in the reduct `nta`, line 20). The semantic function

```
(>>*) infixl 1 :: Task a → [TaskStep a b] → Task b | iTask a       1
                                                    & iTask b       2
(>>*) ta steps = newTask (step1 ta)                                3
where                                                              4
  step1 ta tn t ev st                                              5
  # (Reduct tval nta, rsp, st) = ta ev st                          6
  = hd (  findTriggers tval                                        7
       ++ findActions  tval ev                                     8
       ++ [step1‘ tval nta rsp]                                    9
       ) ev st                                                     10
  where                                                            11
    findTriggers (ExcRes   e) = catchers e ++ [throw e]            12
    findTriggers (ValRes _ v) = values v                           13
                                                                   14
    findActions (ValRes _ v) (ActionEvent tid act)                 15
    | tid==tn               = actions act v                        16
    findActions _ _         = []                                   17
                                                                   18
    step1‘ (ValRes _ v) nta rsp _ st                               19
      = (Reduct no_tval (step1 nta tn t), nrsp ++ rsp, st)         20
    where                                                          21
      no_tval   = ValRes t NoVal                                   22
      as        = [(a,p v) \\ OnAction a p _←steps]                23
      nrsp      = if (isEmpty as) [] [(tn, ActionResponse as)]     24
                                                                   25
  catchers    e = [etb e \\ OnException  etb←steps      ]          26
  values      v = [atb v \\ OnValue    p atb←steps | p v]          27
  actions act v = [atb v \\ OnAction a p atb←steps | act==a        28
                                                   && p v]         29
```

Figure 3. The complete semantic definition of `>>*`.

newTask (Section 3.1.2) provides it with a unique task number for communication with the client and current time stamp t (line 3). The current task description is evaluated first (line 6), resulting in a new task value that is inspected to decide which task step can be stepped into. Triggers (line 7) take priority over actions (line 8). If no task step is applicable, then we proceed with step1 again, but now parameterized with the calculated reduced task (line 9).

A trigger is a task step that can continue without interference of the user. These are the OnException and OnValue task steps. In case of an exception, an exception handler is searched for (line 12 and 26). If none is defined, then the exception propagates (line 12). In case of a task value, all available OnValue task steps are searched for (line 13 and 27).

The actions are selected only if the event is an action event for this task (line 15 and 16). In that case all available OnAction task steps are searched for that match the received action and that are available, as determined by their predicate (lines 28-29).

Finally, when no task step can be selected a reduct is made by step1‘ that waits for a new event (line 20). All actions are collected in the response accumulator (line 23 and 24).

3.5 Parallel Tasks

Tasks can often be divided into parallel sub tasks if there is no specific predetermined order in which the sub tasks have to be done. It might not even be required that all sub tasks contribute sensibly to a stable result. All variants of parallel composition can be handled by a single parallel combinator:

```
parallel :: d → [(ParallelTaskType, ParallelTask a)]
            → Task [(TimeStamp, Value a)] | descr d & iTask a

:: ParallelTaskType = Embedded | Detached ManagementMeta
:: ManagementMeta   = { worker :: Maybe User
                      , role   :: Maybe Role
                      , ...
                      }
:: ParallelTask   a:== SharedTaskList a → Task a
:: SharedTaskList a:== ROShared (TaskList a)
:: TaskList       a = { state  :: [Value a]
                      , ...
                      }
```

We distinguish two sorts of parallel sub-tasks: Detached tasks get distributed to different users and and Embedded tasks are executed by the current user. The client may present these tasks in different ways. Detached tasks need a window of their own while embedded tasks may by visualized in an existing window. With the ManagementMeta structure properties can be set such as which worker must perform the sub-task, or which role he should have.

Whatever its sort, every parallel sub-task can inspect each others progress. Of each parallel sub-tasks its current task value and some other system information is collected in a shared task list. The parallel sub-tasks have *read-only* access to this task list. The parallel combinator also delivers all task values in a list of type [(TimeStamp,Value a)]. Hence, the progress of every parallel sub-task can also be monitored constantly from the "outside". For instance, a parallel task can be monitored with the step combinator >>* to decide if the parallel task as a whole can be terminated because its sub tasks have made sufficient progress for doing the next step. It is also possible to observe the task and convert its value to some other type using the conversion operator @? (Section 3.1.4).

For completeness, we remark that the shared task list is also used to allow dynamic creation and deletion of parallel sub-tasks. We do not discuss this further in this paper.

In the iTask**3** library parallel is used to predefine several frequently used task patterns. In Section 2.2 the -||- combinator was used to start to tasks in parallel.

```
(-||-) infixr 3 :: Task a → Task a → Task a | iTask a
(-||-) a b
  = parallel () [(Embedded,const a),(Embedded,const b)] @? first
where
  first NoVal = NoVal
  first (Value vs _)
    = hd (   [v \\ (_,v=:(Val _ Stable)) ← vs]
          ++ [v \\ (_,v=:(Val _ _)) ← sortBy newer vs]
          ++ [NoVal]
          )
  newer (t1,_) (t2,_) = t1 > t2
```

The first function inspects the progress of both parallel sub-tasks to determine the task value of the composition. The first sub-task to produce a stable task value turns the composition into a stable task with that value. If no sub-task has produced a stable value, then the most recent unstable task value, if any, is the observable result, or no task value is observable at all.

3.5.1 Semantics of the Parallel Combinator

In the semantic description we ignore the meta information assigned to detached tasks and therefore do not distinguish embedded tasks from detached tasks. As another non-essential simplification, we define the shared task list as a *read-write* SDS instead of a *read-only* SDS. The shared task list is a finite map from process ids to task reducts:

```
:: SharedTaskList a:== RWShared (TaskList a)
:: TaskList       a:== [(Pid a, Reduct a)]
:: Pid            a:== Int
```

The complete semantic definition of parallel is given in Figure 4. The semantic function parallel‘ (lines 15-26) defines the purpose of the parallel combinator: to evaluate each and every sub-task (line 18) until either an exception has been thrown (line 19), or all sub-tasks have become stable (lines 23-24). While this is not the case, parallel‘ proceeds to rewrite to itself (line 25-26).

Both parallel‘ and its sub-tasks require access to their progress, which is stored in the shared task list which is created as the first step of the parallel combinator (line 4 and lines 7-13). Initially, the task list consists of all initial parallel sub-tasks that have access to the shared task list.

The semantic functions evalParTasks and evalParTask define the evaluation of the parallel sub tasks: evalParTasks collects the current list of sub-tasks (line 31) and applies evalParTask to each and every sub-task (line 32). Evaluation of a sub-task (line 39) might result in an exception (line 41), in which case the exception is propagated throughout the evaluation of all sub-tasks (lines 44-45). If a sub-task does not result in an exception, then its new reduct is stored in the shared task list (line 42), thus allowing the other sub-tasks to inspect its progress (updateFM (pid,newr) updates any existing element (pid,_) in the shared task list with (pid,newr)). The responses of the evaluated sub-task are collected and returned (line 43).

4. Practical TOP

Although the TOP paradigm adopts functional programming's emphasis of *what* over *how*, some pragmatic issues remain unavoidable in practical TOP programming. In this section we discuss pragmatics issues that we encountered in the implementation of the TOP concept in the iTask**3** toolkit, and show examples of iTask**3** programs to illustrate its use in real-world applications.

4.1 Pragmatic Issues

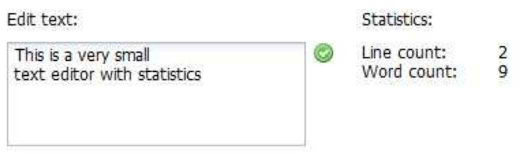
Custom Interaction: TOP programs focus on defining decompositions of tasks without worrying how interactions of basic tasks are implemented by the TOP framework. The underlying implementation has to take care of that. The iTask**3** system follows the

```
parallel :: [ParallelTask a]
        → Task [(TimeStamp,Value a)] | iTask a
parallel ptas ev st
# (stt,st) = createTaskList ptas st
= parallel‘ stt ev st

createTaskList :: [ParallelTask a]
               → *State → *(SharedTaskList a,*State) | iTask a
createTaskList ptas st=:{timeStamp = t}
# (stt,st) = createShared [] st
= (stt,stt.set [  (pid,Reduct (ValRes t NoVal) (pta stt))
               \\ pta←ptas & pid←[0..]
               ] st)

parallel‘ :: SharedTaskList a
         → Task [(TimeStamp,Value a)] | iTask a
parallel‘ stt ev st
= case evalParTasks stt ev st of
    (Left (ExcRes e),st) = throw e ev st
    (Right rsp,st)
    # (values,st) = get_task_values stt st
    # maxt        = foldr max 0 (map fst values)
    | all (isStable o snd) values
                  = stable maxt values ev st
    | otherwise   = (Reduct (ValRes maxt (Val values Unstable))
                            (parallel‘ stt),rsp,st)

evalParTasks :: SharedTaskList a → Event → *State
  → *(Either (TaskResult a) Responses,*State) | iTask a
evalParTasks stt ev st
# (tt,st) = stt.get st
= foldl (evalParTask stt ev) (Right [],st) tt

evalParTask :: SharedTaskList a → Event
           → *(Either (TaskResult a) Responses,*State)
           → (Pid a,Reduct a)
           → *(Either (TaskResult a) Responses,*State)
evalParTask stt ev (Right rsp,st) (pid,Reduct _ ta)
# (newr,nrsp,st)     = ta ev st
# (Reduct ntval nta) = newr
| isExcRes ntval     = (Left ntval,st)
# (_,st)             = updateShared (updateFM (pid,newr)) stt st
= (Right (nrsp ++ rsp),st)
evalParTask _ _ (Left e,st) _
= (Left e,st)

get_task_values :: SharedTaskList a → *State
              → *([(TimeStamp, Value a)],*State)
get_task_values stt st
# (tt,st) = stt.get st
= ([(t,val) \\ (_,Reduct (ValRes t val) _)←tt],st)
```

Figure 4. The complete semantic definition of `parallel`.

semantic definitions, with additional support for customization for obtaining practical applicable applications. The interactive applications that are generated by default by the system suffice for rapid prototyping. However, aesthetic and ergonomic properties of these interactions affect the ease of use and attractiveness of a system. For example, the task of choosing a file from a file system is performed more easily by navigating a tree structure than by selecting an item from a long list of all files.

To allow for such task specific optimization, all interaction tasks in the iTask**3** framework have a *views* parameter, in which optional mappings between the task's domain and another arbitrary domain can be defined. The library provides types that represent abstract user interface controls with which customized interactions can be composed. Here is an example (see Figure 5):


Edit text:

This is a very small
text editor with statistics

Statistics:

Line count: 2
Word count: 9

Figure 5. A very simple text editor

```
:: Statistics = { lineCount :: Int, wordCount :: Int }           1
derive class iTask Statistics                                    2
                                                                 3
simpleEdit :: Task Note                                          4
simpleEdit = withShared (Note "") edit                           5
where                                                            6
 edit note                                                       7
 =  updateSharedInformation "Enter text:" [] note                8
    -||-                                                         9
    viewSharedInformation "Statistics:" [ViewWith stat] note    10
                                             <<@ horizontal     11
                                                                12
  stat (Note txt) = { lineCount = length lines                  13
                    , wordCount = length words                  14
                    }                                           15
  where lines     = split Newline txt                           16
        words     = split " " (replaceSubString Newline " " txt) 17
```

By default, if a value of the predefined type `Note` is used in an iTask**3** editor, a text box is presented to the user on the client to enter text. In `simpleEdit` we create a shared memory for a value of this type `Note` with initial value `Note ""` and we define two interactive tasks on this shared value. The first task allows the user to update the initial text (line 8), while the second gives a view on the shared text that is fine-tuned with `ViewWith` which, in this case, converts the text into a value of type `Statistics`. As a result, while entering text, the user sees the corresponding statistics.

Customized Layout: For task compositions a similar need for customization exist. Depending on the composition, it may be more appealing or easier to use when tasks are divided over tabs or windows than when tasks are shown side-by-side. To customize layout, the iTask**3** framework provides an annotation operator (`<<@`) that can be used to annotate tasks with custom layout functions or post-layout processing functions. Such functions combine a set of abstract GUI definitions into a single definition. By default a heuristic layout function is used to provide a sensible default. Post-processing functions modify a GUI definition after a task is layed out. Such modifications are for example changing its size, adding margins or changing to a horizontal layout as is done with the `<<@ horizontal` annotation in the simple editor. It is defined as:

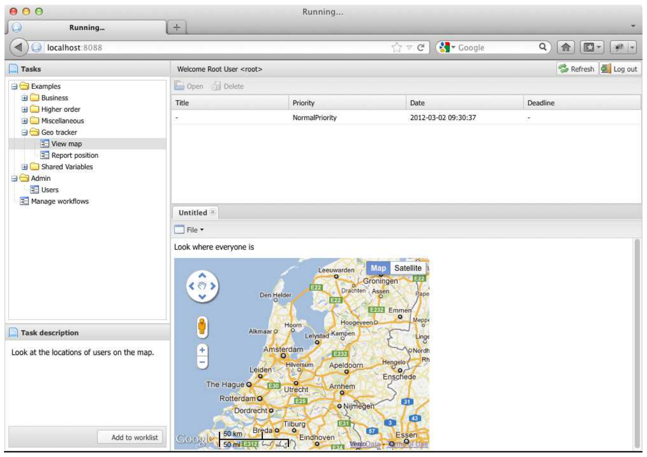
```
horizontal = AfterLayout (tweakUI (setDirection Horizontal))
```

Localization: Another pragmatic aspect one may need to deal with is localization. Because task definitions contain many prompts, hints and other texts, one needs to deal with localization of such texts without compromizing the readability of task definitions. Furthermore, localization may also be required on the task level. To comply with local law and regulations, different task definitions may have to be used in different countries. The iTask**3** framework does not offer any special support for localization, but one can make use of the standard modular structure of Clean to create different local versions.

Third Party Formats and Protocols: To integrate TOP applications with other applications, the gap between the domain of tasks and the formats or protocols required to interact with these systems must be bridged. With TOP one does not escape writing the parsing, formatting and communication code that is necessary for such integrations, but it can be separated from the application code by moving it to task libraries.

4.2 Examples

A Generic Work List: A major leap in the development of TOP as a general paradigm was the insight that, from a user's point of view, interaction with a "Work List", in which users can work on tasks assigned to them, is actually part of the work that has to be done. Work list handling e.g. as offered by an email application or a workflow system is commonly hard coded in the systems used. In iTask**3** this functionality is defined in the system itself as "just" any other task. Figure 6 shows the generic work list task we offer as a standard example. In the left panel a tree of tasks that can be started is displayed. The tasks to do are displayed in the upper-right pane, similar to an inbox in a email application. The user can work on several tasks at the same time in the lower right pane, by opening them in separate tabs. This complete work list application is defined in less than 200 lines of TOP code.



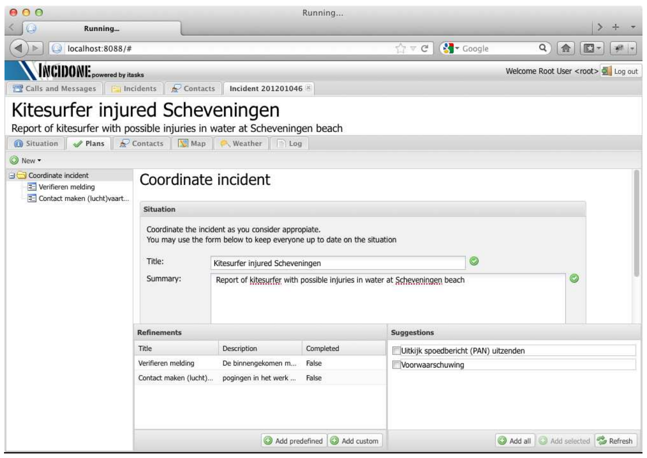
Look at the locations of users on the map.

Title	Priority	Date	Deadline
-	NormalPriority	2012-03-02 09:30:37	-

Look where everyone is

Figure 6. A generic WFMS Work List

The Incidone Incident Coordination Tool: The Coast Guard case study (Jansen et al., 2010; Lijnse et al., 2011) not only fueled the refinement of the task concept and the TOP paradigm, it also lead to the development of the Incidone tool (Lijnse et al., 2012). A preview of this tool for supporting Coast Guard operations is shown in Figure 7. It is being developed using the iTask**3** framework to illustrate the use of TOP for crisis management applications. In this tool immediate information sharing between team members working together is crucial to handle incidents properly.



Kitesurfer injured Scheveningen

Report of kitesurfer with possible injuries in water at Scheveningen beach

Coordinate incident

Situation

Coordinate the incident as you consider appropiate.
You may use the form below to keep everyone up to date on the situation

Title: Kitesurfer injured Scheveningen

Summary: Report of kitesurfer with possible injuries in water at Scheveningen beach

Refinements

Title	Description	Completed
Verifieren melding	De binnengekomen m...	False
Contact maken (lucht)...	pogingen in het werk ...	False

Suggestions

- [ ] Uitkijk spoedbericht (PAN) uitzenden
- [ ] Voorwaarschuwing

Figure 7. The Incidone Tool

5. Related Work

The TOP paradigm emerged during continued work on the iTask system. In its first incarnation (Plasmeijer et al., 2007), iTask**1**, the notion of tasks was introduced for the specification of dedicated workflow management systems. In iTask**1** and its successor iTask**2** (Lijnse and Plasmeijer, 2010), 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 (van der Heijden et al., 2011) and modeling the dynamic task of coordinating Coast Guard Search and Rescue operations (Jansen et al., 2010; Lijnse et al., 2011) we experienced 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 this paper.

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 academic, and many more in the open-source and proprietary commercial software markets. Examples from the academic functional programming community include: the Haskell cgi library (Meijer, 2000); the Curry approach (Hanus, 2001); writing xml applications (Elsman and Friis Larsen, 2004) in *SMLserver* (Elsman and Hallenberg, 2003); WashCGI (Thiemann, 2002); the Hop (Loitsch and Serrano, 2007; Serrano et al., 2006) web programming language; Links (Cooper et al., 2006) and formlets (Cooper et al., 2007). 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 iTask**1** system was influenced and partially motivated by the use of tasks in WFMSs (van der Aalst et al., 2002), iTask**3** has evolved to the more general TOP approach of structuring software systems. As such, it is more similar in spirit to the WebWorkFlow project (Hemel et al., 2008), which is an object oriented approach that breaks down the logic into separate clauses instead of functions. Cognitive Task Analysis methods (Crandall et al., 2006) 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.

6. Conclusions and Future Work

In this paper we introduced Task-Oriented Programming, a paradigm for programming interactive multi-user applications 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*. The step combinator >>* subsumes the classic monad bind operator >>=. The presented operational semantics specifies the constructs unambiguously. The development of this semantics was an important anchor point during the design of TOP.

TOP is embedded in Clean by offering a newly developed iTask**3** 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 rescue operations requires up-to-date information of subtasks, this is precisely the goal of TOP. In collaboration with Dutch industry we started to investigate and validate the suitability of the TOP paradigm to handle specific complex real world distributed application areas.

References

Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Ana Barros. Workflow patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology, 2002.

Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005. ISBN 3-540-67658-9.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, November 2006. Springer-Verlag.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 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`.

Beth Crandall, Gary Klein, and Robert R Hoffman. *Working Minds: A practitioner's guide to cognitive task analysis*. MIT Press, 2006. ISBN 978-0-262-03351-0.

Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *Proceedings of the 6th International Symposium on the Practical Aspects of Declarative Programming, PADL '04*, volume 3057 of *Lecture Notes in Computer Science*, pages 224–238. Dallas, TX, USA, Springer-Verlag, June 2004.

Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *Proceedings of the 5th International Symposium on the Practical Aspects of Declarative Programming, PADL '03*. New Orleans, LA, USA, Springer-Verlag, January 2003.

John van Groningen, Thomas van Noort, Peter Achten, Pieter Koopman, and Rinus Plasmeijer. Exchanging sources between Clean and Haskell - A double-edged front end for the Clean compiler. In Jeremy Gibbons, editor, *Proceedings of the Haskell Symposium, Haskell '10, Baltimore, MD, USA*, pages 49–60. ACM Press, 2010.

Michael Hanus. High-level server side web scripting in Curry. In *Proceedings of the 3rd International Symposium on the Practical Aspects of Declarative Programming, PADL '01*, pages 76–92. Springer-Verlag, 2001.

Zef Hemel, Ruben Verhaaf, and Eelco Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS '08*, volume 5301 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2008.

Ralf Hinze. A new approach to generic functional programming. In Tom Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000.

Jan Martin Jansen, Bas Lijnse, and Rinus Plasmeijer. Towards dynamic workflows for crisis management. In Simon French, Brian Tomaszewski, and Cristopher Zobel, editors, *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management, ISCRAM '10*, Seattle, WA, USA, May 2010.

Pieter Koopman, Rinus Plasmeijer, and Peter Achten. *An Effective Methodology for Defining Consistent Semantics of Complex Systems*, volume 6299 of *LNCS*, pages 224–267. Springer-Verlag, Komarno, Slovakia, 25-130, May 2009.

Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for End-users. In Marco Morazán and Sven-Bodo Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA*, volume 6041 of *LNCS*, pages 36–54. Springer-Verlag, 2010.

Bas Lijnse, Jan Martin Jansen, Ruud Nanne, and Rinus Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In David Mendonca and Julie Dugdale, editors, *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM '11*, Lisbon, Portugal, May 2011. ISCRAM Association.

Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. Incidone: A task-oriented incident coordination tool. In Leon Rothkrantz, Jozef Ristvej, and Zeno Franco, editors, *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*, Vancouver, Canada, April 2012.

Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '07*, pages 141–158, New York, NY, USA, 2-4, April 2007. Interact.

Erik Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.

Steffen Michels and Rinus Plasmeijer. Uniform data sources in a functional language. Submitted for presentation at Symposium on Trends in Functional Programming, TFP '12, 2012. URL `https://wiki.clean.cs.ru.nl/File:Sharing_Data_Sources.pdf`.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 2007. ACM Press.

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 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 '06*, pages 975–985, Portland, Oregon, USA, 22-26, October 2006.

Peter Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In Shriram Krishnamurthi and Raghu Ramakrishnan, editors, *Proceedings of the 4th International Symposium on the Practical Aspects of Declarative Programming, PADL '02*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208, Portland, OR, USA, 19-20, January 2002. Springer-Verlag.

Maarten van der Heijden, Bas Lijnse, Peter Lucas, Yvonne Heijdra, and Tjard Schermer. Managing COPD exacerbations with telemedicine. In *13th Conference on Artificial Intelligence in Medicine, AIME '11*, volume 6747 of *LNCS*, pages 169–178, Bled, Slovenia, July 2011. Springer-Verlag.

Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *Revised Selected Papers of the 14th International Workshop on the Implementation of Functional Languages, IFL '02, Madrid, Spain*, volume 2670 of *LNCS*, pages 101–117. Springer-Verlag, 2003.

Arjen van Weelden. *Putting types to good use*. PhD thesis, Radboud University Nijmegen, 17, October 2007. ISBN 978-90-9022041-3.