Task-Oriented Programming in a Pure Functional Language

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Pieter Achten, Pieter Koopman

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

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 pandemic 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 iTasks3 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 iTasks3 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 iTasks3 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 iTasks3 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
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 \texttt{planMeeting}. Its type (line 1) expresses that given a list of users, it is a task used concepts are defined in Section 3.

As discussed, the main structure of \texttt{planMeeting} consists of three subsequent sub-tasks (lines 2-4), which are glued together by means of the *step* combinator \texttt{>>}. The second argument of \texttt{>>*} enumerates the potential subsequent *task steps* that can be stepped into while the first argument task is in progress. Hence, the first sub-task, \texttt{enterDateTimeOptions}, is followed by \texttt{askPreferences}, which in turn is followed by either \texttt{tryAgain} or \texttt{decide}. Entering user information (performed by \texttt{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 \texttt{hasValue}. In that case, the current task value is retrieved (\texttt{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: \texttt{hasValue} tests for the \texttt{Val} data constructor, and \texttt{getValue} returns that value if present:

\begin{verbatim}
:: Stability = Unstable | Stable

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

getValue :: Value a -> a
getValue (Val a) = a
\end{verbatim}

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: \texttt{?} alters the task value of the preceding task, and \texttt{@} is similar, but only if a \texttt{Val} is present:

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

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 addi-
tion, 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 $\exists$ 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 $\forall$ to affect the user to identify who made that specific selection (lines 30-31).

Finally, the last sub-task can be stopped 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 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 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 iTask3 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 iTask3 API and code snippets, we display semantic definitions as framed verbatim text, and iTask3 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:

```haskell
:: TaskResult a = ValRes TimeStamp (Value a)
| \exists: 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. 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 $t_n$ (Val $p_n$, 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 $\exists$ and the type class context restriction $\forall$ 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:

```haskell
:: Task a ::= Event -> *State -> (*Reduced a, Responses, *State)
:: Event = RefreshEvent
| EditEvent TaskId Dynamic // Section 3.3.1
| ActionEvent TaskId Action // Section 3.4.1
:: *State = { taskId :: TaskId,
| timeStamp :: TimeStamp // Section 3.3.1,
| mem :: [Dynamic] // Section 3.2.1
| world :: *World }
:: Reduced a = Reduced (TaskResult a) (Task a)
:: TaskId ::= Int
:: Responses ::= [(TaskId, 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:

```plaintext
case = (newTask ta ev st = ta no t ev {st & newTask = 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 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.

```plaintext
evaluateTask :: Task a --> *world --> *((Maybe a, *State)) | iTask a  
evaluateTask ta world              = ta ev st = (ta no t ev {st & world = w})
# (st, world) = rewrite ta st
= (st & world = world)

rewrite :: Task a --> *state --> *((Maybe a, *State)) | iTask a
rewrite ta st = ([st] & world)
  
# (ev, world) = getNextEvent world
# (t, world) = getCurrentTime world
    # (t & world = world)
    # (Reduct res nta, resp, st) = ta ev st
  
  = case res of
    ValRes _ (Val a Stable) --> (J ust a, st)
    ExcRes _ (Nothing, st) --> rewrite nta
    [st & world = informClients resp st.world]  
```

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 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 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:

```plaintext
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 reduce (value and continuation). With this semantic function, we can define the semantic function of the core task return:

```plaintext
return :: a --> Task a
return a = st = (Reduct (ExcRes e) (throw e))[[], st]
```

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

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

```plaintext
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:

```plaintext
(@) infixl 1 :: Task a --> (Value a --> Value b) 
(@) ta f ev st = case ta ev st of
  (Reduct (ValRes t aval nta, resp, nst)) --> case f aval of
    Val b Stable --> stable t b ev nst
    bval --> (Reduct (ValRes t bval) (nta @ f), resp, nst)
  (Reduct (ExcRes e) __, nst)) --> throw e ev nst

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

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.
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.

```plaintext
:: ROShared r w
:: GSShared w := ROShared () w
:: Shared a := GSShared a a
```

A SDS has abstract type ROShared 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 (GSShared 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 iTask3 system to create SDSs:

```plaintext
sharedFile :: Path → a → Shared a | iTask a
currentTime :: ROShared Time
currentUsers :: GSShared [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 file, and reads and updates the stored values using the combinator withShared va (λsa → ta):

```plaintext
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 a using a function f with the combinator ta ⊗ (f,a):

```plaintext
(⊗) infix l :: Task a → (Value a → r → Maybe w, ROShared 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 a, the result of which is written to a. 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 the 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:

```plaintext
:: ROShared r w = { get :: *State → *(r, *State) set :: w → *State → *State }
```

To achieve this abstraction we use the concept of multi-purpose SDSs that integrate smoothly with interactive tasks. For every basic interactive task (such as enterChoice and enterMultipleChoice) a sub-task is using a function f (for offering shared memory between (parallel) tasks. These 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 createShared va (λsa → ta):

```plaintext
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 a using a function f with the combinator ta ⊗ (f,a):

```plaintext
(⊗) infix l :: Task a → (Value a → r → Maybe w, ROShared 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 a, the result of which is written to a. 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 the 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:

```plaintext
:: ROShared r w = { get :: *State → *(r, *State) set :: w → *State → *State }
```

To achieve this abstraction we use the concept of multi-purpose SDSs that integrate smoothly with interactive tasks. For every basic interactive task (such as enterChoice and enterMultipleChoice) a sub-task is using a function f (for offering shared memory between (parallel) tasks. These 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 createShared va (λsa → ta):

```plaintext
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 a using a function f with the combinator ta ⊗ (f,a):

```plaintext
(⊗) infix l :: Task a → (Value a → r → Maybe w, ROShared 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 a, the result of which is written to a. 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.
With `viewShared` 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 `Nothing`) 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 iTask3 framework are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enterInformation</code></td>
<td>: d → [EnterOpt m]</td>
</tr>
<tr>
<td><code>updateInformation</code></td>
<td>→ Task m</td>
</tr>
<tr>
<td><code>viewInformation</code></td>
<td>: d → [ViewOpt m] m → m</td>
</tr>
<tr>
<td><code>viewSharedInformation</code></td>
<td>→ Task m</td>
</tr>
<tr>
<td><code>updateSharedInformation</code></td>
<td>: d → [UpdateOpt r w] → RWShared r w → Task w</td>
</tr>
<tr>
<td><code>viewSharedInformation</code></td>
<td>: d → [ViewOpt r] → RWShared r w</td>
</tr>
</tbody>
</table>

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.

```haskell
:: ViewOpt a =⇒ ViewWith (a ⇒ v) & iTask v
:: UpdateOpt a b =⇒ UpdateWith (a ⇒ v) (b ⇒ v) & 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 iTask3 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` 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 response of the server tell the client what interface should be rendered to the user.

```haskell
:: Response = EditorResponse EditorResponse |
               ActionResponse ActionResponse // Section 3.4.1
:: EditorResponse = { descr = 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 `a` while displaying the latest value `r` stored in an SDS of type `RWShared r a`.

```haskell
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
t��nt y = case ev of
  EditEvent tid dyn |
  → if (tid == tn)
    → (st, timeStamp, de_serialize dyn)
    |
    (t, lv)
    |
    → (t, lv)

# (sr, st) = sh_rw.get st |
= (Reduc (ValRes nt (toValue (cv nlv sr)))) (edit1 nlv tn nt)
  |
  ([tn, EditorResponse |
    { descr = descr, 
      editing = Editing, 
      editValue = (serialize nlv, serialize sr)
    }])

where
toValue :: Maybe a → Value a
```
The palindrome task prompts the user to enter a palindrome. As usual, the user can enter a string and change it over time. With \( \Rightarrow \Rightarrow \) 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 the palindrome 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 \( \Rightarrow \Rightarrow \) to demonstrate the general nature of \( \Rightarrow \Rightarrow \):

\[
(\Rightarrow \Rightarrow) \text{ infixl 1 :: Task a } \rightarrow \text{[TaskStep a b]} \rightarrow \text{ Task b | iTask a & iTask b}
\]

\[
:: \text{ Task a b = OnAction Action (Predicate a) (NextTask a b)} \mid \text{ OnValue (Predicate a) (NextTask a b) \mid \exists e: QuException (e \rightarrow Task b) & iTask e}
\]

\[
:: \text{ Action = Action String | ActionOk | ActionCancel | ...}
\]

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 (QuException). 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, QuException task steps handle an exception only if their argument function matches the type of the exception. Uncaught exceptions are propagated by \( \Rightarrow \Rightarrow \).

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 \( \Rightarrow \Rightarrow \) with two examples.

The palindrome task description in its first argument (initially task ta, line 3, and in the reduct nta, line 20). The semantic function

\[
\text{palindrome = Task (Maybe String)}
\]

\[
>>> \text{ [OnException ActionOk ifPalindrome \ AsValue (return Just o getValue)\[1\] QAction ActionCancel (const True)\}
\]

\[
:: \text{ActionResponse} :=[([Action, Enabled])]
\]

\[
:: \text{Enabled} :=[\text{Bool}]
\]

\[
(\Rightarrow \Rightarrow) \text{ infixl 1 :: Task a } \rightarrow \text{[TaskStep a b]} \rightarrow \text{ Task b | iTask a & iTask b}
\]

\[
:: \text{Action = Action String | ActionOk | ActionCancel | ...}
\]

\[
\text{palindrome = } \text{enterInformation } \text{"Enter a palindrome" } [1]
\]

\[
\Rightarrow \Rightarrow \text{ OnException ActionOk ifPalindrome (return Just o getValue)\[1\] OnAction ActionCancel (const True)\}
\]

\[
:: \text{ActionResponse} :=[([Action, Enabled])]
\]

\[
:: \text{Enabled} :=[\text{Bool}]
\]
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
```

We distinguish two sorts of parallel sub-tasks: detached tasks get distributed to different users and 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 be 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 iTask3 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.

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 (update `?) (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 iTask3 toolkit, and show examples of iTask3 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 iTask3 system follows the
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 iTask3 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):

```
parallel :: [ParallelTask a] a
  → Task [(TimeStamp, Value a)] | iTask a
parallel pta st ev st
# (ev.st) = createTaskList pta st
= parallel' stt ev st
createTaskList :: [ParallelTask a] a
  → sState → *(SharedTaskList a, State) | iTask a
createTaskList pta stt stt
# (stt, stt) = createShared [] st
= (stt, stt).set (pid, Reduce (ValRes theNoVal) (pta stt))
  \ 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 resp, st) →
  # (values, st) = get_task_values stt st
  # max (max) = foldr max 0 (map fst values)
  | left = (isStable o ev st, values)
  | otherwise = (reduce (ValRes maxt values (Val unstable)),
  ) (parallel' stt).resp st

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

get_task_values :: SharedTaskList a → *State
  → *(Either TaskResult a, Responses, State)
get_task_values stt st
# (tt, st) = stt.get st
= ((tt, ValRes t val) \ Reduce (ValRes t val) \ tt).stt st
```

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

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 iTask3 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.

5. Related Work

The TOP paradigm emerged during continued work on the iTask system. In its first incarnation (Plasmeijer et al., 2007), iTask1, the notion of tasks was introduced for the specification of dedicated workflow management systems. In iTask1 and its successor iTask2 (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 iTask1 system was influenced and partially motivated by the use of tasks in WFMSs (van der Aalst et al., 2002), 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 (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 iTasks3 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 sub-tasks, 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


