Getting a Grip on Tasks that Coordinate Tasks

Rinus Plasmeijer
Radboud University Nijmegen
http://wiki.clean.cs.ru.nl/Clean
http://wiki.clean.cs.ru.nl/ITasks

Peter Achten (ru) - Pieter Koopman (ru) -
Thomas van Noort (stw) - John van
Groningen (ru) - Bas Lijnse (navy) - Jan
Martin Jansen (navy) - Steffen Michels
(SITPro) - Jeroen Henrix (SITPro) -
Laszlo Domoszlai (ELTE)
It is a combinator library written in

- It is a toolbox for the rapid development of WFMSs
- It is a Domain Specific Language embedded in Clean
iTask Server Coordinates Tasks executed by Clients

iTask User

RPC

Web Service

Web Service

Mobile iTask App

ExtJs - Sapl - Clean JavaScript

Browser

iTask Workflow

iTask Combinator Library

iTask Web Service

Server

WEB
It is declarative...

“Declarative specification of data and tasks is sufficient for generating an executable workflow”

Abstract from implementation details as much as we can using type driven generic functions

I/O handling, communication, JSON / XML exchange, web form generation, web form updating, persistent storage, …
i - Tasks - Embedded Workflow Description Language

basic tasks: Task a - unit of work delivering a value of type a

- Filling in a web form, web-service, OS-call (time, date), application call, database access

+ combinators for combining tasks

- Common usage
  - define order of tasks (sequential, parallel)
  - assign properties to tasks (worker, priority, deadline),

- Exceptional usage
  - workflow / task process handling (create, waitFor, suspend, kill)
  - exception handling

+ Clean host language features

- recursive -, higher order -, polymorphic -, overloaded -, generic - functions
- strongly typed + dynamic typing
### Examples of basic tasks for filling in forms

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enterInformation</td>
<td>d → Task a</td>
</tr>
<tr>
<td>updateInformation</td>
<td>d a → Task a</td>
</tr>
</tbody>
</table>

#### class iTask a

- `gVisualize {[*]}` // information for form creation
- `gUpdate {[*]}` // form update
- `gEq {[*]}` // equality test
- `gDefaultMask {[*]}` // form status
- `gVerify {[*]}` // predicate value has to obey
- `JSONEncode {[*]}` // JSON encoding - decoding
- `JSONDecode {[*]}` // JSON encoding - decoding
- `XMLEncode {[*]}` // XML encoding - decoding
- `XMLDecode {[*]}` // XML encoding - decoding
- `TC a` // serialization - de-serialization
module example

import iTasks

Start :: *World → *World
Start world = startEngine [workflow "demo task" myTask] world

myTask :: Task Int
myTask = enterInformation "Please fill in the form:"
iTak Client

![iTak Client Screenshot](image-url)
A very small *complete* example II

myTask = enterInformation "Please fill in the form:"
A very small *complete* example II

myTask :: Task [Person]
myTask = enterInformation "Please fill in the form:"

:: Person = { firstName :: String,
              surName :: String,
              dateOfBirth :: Date,
              gender :: Gender }

:: Gender = Male | Female

derive class iTask Person, Gender
Core Combinators

Basic combinator: interactive editor for filling in forms of a certain type:

\[
\text{updateInformation} :: d \ a \rightarrow \text{Task a} \ | \ i\text{Task a} \ & \ descr \ d
\]

Main task: define task properties (who has to work on it, priority, deadline):

\[
(@:) \ \text{infix} \ 3 :: p (\text{Task a}) \rightarrow \text{Task a} \ | \ i\text{Task a} \ & \ property \ p
\]

Sequencing of tasks using monadic bind >>= and return:

\[
(\gg=) \ \text{infix} \ 1 :: (\text{Task a}) (a \rightarrow \text{Task b}) \rightarrow \text{Task b} \ | \ i\text{Task a} \ & \ i\text{Task b}
\]

\[
\text{return} :: a \rightarrow \text{Task a} \ | \ i\text{Task a}
\]

Parallel evaluation of tasks:

\[
(-||-) \ \text{infix} \ 3 :: (\text{Task a}) (\text{Task a}) \rightarrow \text{Task a} \ | \ i\text{Task a}
\]

\[
(-&&-) \ \text{infix} \ 4 :: (\text{Task a}) (\text{Task b}) \rightarrow \text{Task (a, b)} \ | \ i\text{Task a} \ & \ i\text{Task b}
\]

With just a few combinators many frequently occurring flows can be defined

semantics: term rewriting system (IFL 2008, PEPM 2011)

Open question: What kind of combinators do we really need?
## Core Combinators

### Basic combinator: interactive editor for filling in forms of a certain type:

<table>
<thead>
<tr>
<th>combinator</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>updateInformation</td>
<td>p a → Task a</td>
</tr>
</tbody>
</table>

### Main task: define task properties (who has to work on it, priority, deadline):

<table>
<thead>
<tr>
<th>combinator</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(@:) infix 3</td>
<td>p (Task a) → Task a</td>
</tr>
</tbody>
</table>

### Sequencing of tasks using monadic bind >>= and return:

<table>
<thead>
<tr>
<th>combinator</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&gt;&gt;=) infix 1</td>
<td>(Task a) (a → Task b) → Task b</td>
</tr>
<tr>
<td>return</td>
<td>a → Task a</td>
</tr>
</tbody>
</table>

### Parallel evaluation of tasks:

<table>
<thead>
<tr>
<th>combinator</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>([a] → Bool) ([a] → b) ([a] → b) [Task a] → Task b</td>
</tr>
</tbody>
</table>

**Open question:** What kind of combinators do we really need?
Defined many toy applications: (see iTask distribution)
“Real” Prototype Applications using iTasks

Simple workflow:

- Aerial project: Home Healthcare project (Peter Lucas, Bas Lijnse, e.a.)
  - Testing chronically long diseases caused by smoking
  - Testing pregnancy disease

Real real-life workflow:

- Crisis Management:
  Capturing the Netherlands Coast Guard’s Search And Rescue Workflow
  (ISCRAM 2011, Bas Lijnse, Jan Martin Jansen, Ruud Nanne, Rinus Plasmeijer)
Home Healthcare project

Measurements are sent wirelessly to the phone

A questionnaire is answered by touchscreen
Coast Guard Search And Rescue
Coast Guard Search And Rescue
What did we learn?

- **Coordination panels** should not be built-in but become user-definable tasks as well.
  - E.g. the iTask main system panel.

- **Sharing of information between tasks** needed to monitor developments.
  - Also needed for many-to-many communication.

- Forms are not enough: need to be able to specify GUI's (windows, menus, ...).

- One cannot foresee everything: we have to be able to change running workflows.

Currently designing + implementing in version 3.0.

- All this functionality should be offered by the new API.

- Yet: we expect to base it only on very few Swiss Army-Knife combinators.

*Work in Progress*

*Check back soon!*
What have we done so far?

Small extensions to Clean:

- Added *(Generic)* context restriction in types
- Allow overloaded and generic functions in dynamics
- Allow generic functions to be overloaded in generic functions

Basic tasks:

- `updateInformation :: d (View i v o) [Action i] (Shared i o) → Task (Event, Maybe i)`
  - `iTask i & iTask v & iTask o & descr d`

Combinators for combining tasks

**Common usage:**

- `parallel :: d (Merge a ps b) [CTask a ps] [Task a] → Task b`
  - `iTask a & iTask ps & iTask b & descr d`

**Exceptional usage:**

- workflow/task process handling
- exception handling
- change handling
More Future work

- Improve Practical Applicability
  - Embedding with existing databases
    - ORM specification used to map RDB <-> Clean data types
  - Distributed Servers
  - Add iTasks running on the client, now in JavaScript
  - How to offer dynamic change to the end user ?
  - Reasoning ? Proving ? Testing ?