Tasklets: Client-side evaluation for iTask3

László Domoszlai[1] and Rinus Plasmeijer[2]

[1] Department of Programming Languages and Compilers, Eötvös Loránd University
Budapest, Hungary
[2] Software Technology Department, Radboud University
Nijmegen, The Netherlands
dlacko@pnyf.inf.elte.hu, rinus@cs.ru.nl

Abstract. iTask3 [14] is the most recent incarnation of the iTask framework for the construction of distributed systems where users work together on the internet. It offers a domain specific language for defining applications, embedded into the lazy functional language Clean. From the mere declarative specification a complete multi-user web application is generated. Although the generated nature of the user interface (UI) entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom UI building blocks. In this paper, we present an extension to the iTask3 framework which introduces the concept of *tasklets* for the development of custom, interactive web components in a single language manner. We further show that the presented tasklet architecture can be generalized in such a way that arbitrary parts of an iTask application can be executed on the client.

1 Introduction

The iTask framework was originally developed as a dedicated web-based Workflow Management System (WFMS). Its most recent incarnation, iTask3, however, extends its boundaries beyond classical WFMS and offers a novel programming paradigm for the construction of distributed systems where users work together on the internet.

According to the iTask paradigm, the unit of application logic is a *task*. Tasks are abstract descriptions of interactive persistent units of work that have a typed value. When a task is executed, it has an opaque persistent value, which can be observed by other tasks in a controlled way. In iTask, complex multi-user interactions can be programmed in a declarative style just by defining the tasks that have to be accomplished. The specification of the tasks is given by a domain specific language embedded in the pure, lazy functional language Clean. Furthermore, the specification is given on a very high level of abstraction and does not require the programmer to provide any user interface definition. Merely by defining the workflow of user interaction, a complete multi-user web application is generated, all the details e.g. the generation of web user interface, client-server communication, state management etc. are automatically taken care of by the framework itself.

Developing web applications such a way is straightforward in the sense that the programmers are liberated from these cumbersome and error-prone jobs, such that they can concentrate on the essence of the application. The iTask system makes it very easy to develop interactive multi-user applications. The down side is that one has only limited control over the customization of the generated user interface, but for this type of applications, this is often acceptable. However, the experiment with real world applications, e.g. the implementation of the Netherlands Coast Guard's Search and Rescue (SAR) protocol [10, 11], indicated that even if the functional web design is satisfactory, custom building blocks may be required for the purpose of user-friendliness. A good example is the aforementioned SAR workflow, where Google MAPS widgets complemented the otherwise functional web application to visualize the locations of incidents.

To overcome this shortcoming, in this paper we present an extension for the iTask3 system which enables the development of such widgets, the so called *tasklets*. Tasklets are seamlessly integrated into iTask to preserve the elegance of functional specification by hiding the behavior behind the interface of a task. Tasklets are developed in a *single-language*, declarative manner and in accordance with the *model-view-controller* user interface design (MVC) [9]. MVC decouples the application logic (the controller), the application data (the model) and the presentation data (the view) to increase flexibility and reuse. Technically speaking, tasklets are embedded applications which behavior is encoded in Clean written event handler functions. The event handlers are executed in the browser, where, they have unrestricted access to client-side resources. Using browser resources the tasklet can create custom appearance and exploit functionality available only in the browser (e.g. HTML5 GeoLocation API), utilizing the event-driven architecture the tasklet can achieve interactive behavior. With this extension, iTask gains similar characteristics to *multi-tier* programming languages like Links [4] or Hop [15, 16], in the sense that the same language is used to specify code residing on multiple locations or tiers, such as the client and the server.

We further show that the presented tasklet facility can be used to improve the responsiveness of an iTask application by enabling the execution of ordinary tasks (virtually any part of an iTask application) in the browser instead of the server. This, amongst other things, helps with avoiding the latency of communication, thus providing smoother user experience. Executing an iTask task in the browser demands much more than executing an ordinary function. Tasks have complex, interactive behavior and e.g. observable intermediate values which requires communication with other tasks; therefore the execution must obey a certain *evaluation strategy*. We will obtain general client-side execution support by encoding this evaluation strategy in a tasklet.

In this paper we make the following contributions:

- The iTask framework is extended to enable the development of client-side, interactive UI components in a single-language, declarative manner. These components can be used to increase the expressiveness of the functional iTask applications, and to provide functionality which is available only in

the browser. This facility, called tasklet, is designed in such a way to fit as seamlessly as possible into the iTask formalism, that is to be opaque for the developer of the functional specification and to retain the advantageous generated nature of user interfaces of iTask applications as much as possible;
- Tasklets foster the model-view-controller user interface design to separate the application logic, the application data and the presentation data. The separation of these roles helps with increasing code flexibility, reuse and maintainability;
- We further show that the tasklet architecture is versatile enough to pave the way for the evaluation of almost all tasks at the client-side. Executing tasks in the browser helps with avoiding client-server communication to reduce server load and provide smoother user experience. This feature also creates the preconditions for running iTask applications offline in a browser which is a desired direction of future development;
- Finally, tasklets utilize a special compilation technique to enable the execution of arbitrary expression of an iTask application in the browser without shipping of unnecessary code. This technique is based on run-time deserialization of Clean expressions and involves on the fly compilation to JavaScript. By minimizing the amount of client code, this approach has the definite advantages of reducing communication cost and memory usage in the browser. Moreover it makes it possible to dynamically tune the set of tasks executed in the browser by the current server load or other run-time information.

The remainder of this paper is organized as follows: in Section 2 we start with a short overview of the iTask framework and develop a non-trivial, but necessarily simplified example of a flight check-in application to give a taste of iTask. In Section 3 we introduce the tasklet architecture and demonstrate its usage by developing a tasklet to enrich the example of the previous section. Some real-world use cases studies are discussed in Section 4. In Section 5 we briefly discuss the design of the tasklet architecture, then we generalize it in Section 6 to enable the execution of legacy tasks; some common restrictions on its applicability is also given in this section. After a discussion of related work in Section 7, we conclude in Section 8.

The iTask framework has been created in Clean. A concise overview of the syntactical differences with Haskell is in [2]. We assume the reader is familiar with the concept of generic programming and uniqueness typing.

2 Introduction to iTask

The most recent incarnation of the iTask system, iTask3, is a prototype framework for programming workflow support applications in Clean using a new programming paradigm built around the concept of a *task*. iTask uses a combinator-based embedded domain specific language (EDSL) to specify compositions of interdependent tasks. From these specifications, complete multi-user web applications are generated.

```
:: Task a       // Task is an opaque, parameterized type constructor

// Exception handling:
throw    :: e                              → Task a | iTask a & iTask, toString e
catchAll :: (Task a) (String → Task a) → Task a | iTask a

// Sequential composition:
(>>=) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b
(>>|) infixl 1 :: (Task a) (Task b)      → Task b | iTask a & iTask b
return         :: a                      → Task a | iTask a

// Parallel composition:
(||-) infixr 3 :: (Task a) (Task b)      → Task b | iTask a & iTask b

// User interaction:
viewInformation  :: String m     → Task m | iTask m
enterInformation :: String       → Task m | iTask m
enterChoice      :: String (c o) → Task o | OptionContainer c & iTask o
```

Fig. 1. Combinators and primitive tasks used in the paper

Tasks are abstract descriptions of interactive persistent units of work that are represented by the opaque type `Task a`, where `a` denotes the type of the value that will be, eventually, delivered by the task when it is executed. Tasks can be combined *sequentially*. The infix functions `return` and `>>=` are standard monadic combinators. Task `f >>= s`, first performs task `f`, then the value produced by `f` can be used by task `s` to compute any new task expression. The combinator `>>|` works similarly, but it drops the value of the first task during composition. Task `return v` produces value `v` without any effect. Tasks also can be performed in *parallel*. In this paper only the rather special `||-` combinator is used; it groups two tasks in parallel and return the result of the right task.

The primitive task `enterInformation` is a *generic editor*, a type-driven task which generates a web form for the arbitrary (first-order) type `m` and allows the user to enter and edit a value of that type. Similarly, `enterChoice` allows the user to choose from a set of values of type `o`. The selectable values must be disposed in a container, the type of which is an instance of the type class `OptionContainer`. Predefined instances of the `OptionContainer` class are the list type and a simple tree type to enable hierarchical selection. Finally, `viewInformation` is used to display a given value of the type `m`. The first argument of these functions is a brief description of what the end-user is expected to do. Most type definitions of the iTask combinators contain a closure at the end of their type signature, e.g. `| iTask m`. This closure imposes a type restriction on the type variable `m`. It means, that `m` can be arbitrary type, provided that some generic functions, necessary for the iTask run-time system, must have instances for the given type.

A task can raise an exception in case it can no longer produce a meaningful value. Any value can be thrown as exception by the `throw` function, provided that it can be serialized as a string. Exceptions can be caught by `catchAll` the first argument of which is a task that will possibly raise an exception, and its second argument is a task to handle it.

In Figure 1, the small set of combinators and primitive tasks of the iTask DSL is presented which are used throughout this paper (for reasons of presentation, the types have been slightly simplified). The full language definition and its semantics can be found in [14].

In the rest of this section, we demonstrate the expressive power of iTask presenting an overly simplified, but still realistic example of a flight check-in application. The application will operate on the following types:

```
:: Seat = Seat Int Int    // Seat information: row, seat number in the row
:: Seats :== [Seat]

:: Booking = { bookingRef   :: String          // Unique booking reference number
             , firstName    :: String          // Passenger's first name
             , lastName     :: String,         // Passenger's last name
             , flightNumber :: String,         // Flight number
             , pid          :: Hidden String,  // Unique number of passenger's ID
             , seat         :: Maybe Seat      // Seat information
             }

:: Flight  = { flightNumber :: String          // Unique flight number
             , free         :: Seats           // List of free seats
             }
```

The `Booking` type describes a booking for a flight. It contains a unique reference number, the flight number, and data of the passenger, including the unique number of the ID document (`pid`). This latter is wrapped in the Hidden type to indicate for the framework that it is not supposed to be displayed on any of the screens. For the sake of brevity, the last field, `seat`, encodes seat information and also indicates whether the passenger is checked-in. If a seat number is present, the passenger is already checked-in, otherwise has not been yet. The `Flight` record type describes a simplified view of flight data; in our case it contains only the unique flight number and the list of vacant seats.

To concentrate on the essence of the application, the implementation of the following functions, comprising the data tier, are omitted:

```
// Find flight and booking records by flight number and reference number accordingly
findFlight  :: String → Task (Maybe Flight)
findBooking :: String → Task (Maybe Booking)
// Returns a list of booking records fulfilling a condition given by the first argument
listBookings :: (Booking → Bool) → Task [Booking]
// Update datasets and returns the up-to-date booking record
commitCheckIn :: Booking Seat → Task Booking
```

To keep the example as concise as possible, a very simple exception controlled mechanism is used to handle errors; when an exception occurs the application

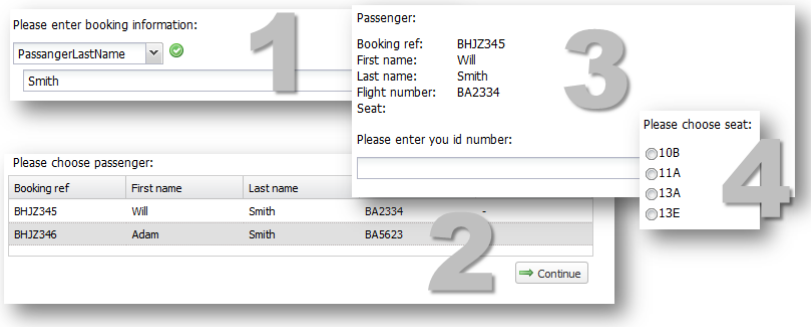


Fig. 2. The flight check-in screens

prints the error message and restarts the workflow. Therefore, the main task, `checkIn`, is responsible for handling exceptions only. The task does not return any meaningful value (`Void`), its semantics is based on side-effect:

```
checkIn :: Task Void
checkIn = catchAll workflow (λmsg → viewInformation "Error:" msg >>| checkIn)
```

Thanks to exceptions, the top level workflow can be straightforwardly decomposed to a sequence of tasks:

```
workflow = enterInformation "Please enter booking information:"          1
  >>= λbi   → lookUpBooking bi                                           2
  >>= λmbB  → verifyBooking mbB                                          3
  >>= λb    → findFlight b.Booking.flightNumber                          4
  >>= λf    → chooseSeat f                                               5
  >>= λseat → commitCheckIn b seat                                       6
  >>= viewInformation "Check-in succeeded:"                              7
  >>| checkIn                                                            8
```

First, the user is asked to provide booking information (line 1). The entered information is used to look up the booking record (line 2), then the identity of the user and other prerequisites are verified (line 3). After looking up the related flight record in line 4, the user is asked to choose seat (line 5). Finally, the check-in is committed to the database and the updated booking record is displayed (line 6-7). In the last line, the workflow is restarted to continue with a new check-in procedure.

The generic `enterInformation` function in line 1, generates a user interface for the `BookingInfo` type; this type is inferred by looking at the type of `lookUpBooking`. According to this type, the passenger is asked to provide the booking reference number or her last name:

```
:: BookingInfo = BookingReference String | PassangerLastName String
```

In `lookUpBooking`, if a reference number was provided, the booking record is looked up. Otherwise the user is asked to choose (using `enterChoice`) one of

the booking records in which the passenger's last name matches and contains no seat information. The function returns `Nothing` if a booking record could not be found:

```
lookUpBooking :: BookingInfo → Maybe Booking
lookUpBooking (BookingReference ref) = findBooking ref
lookUpBooking (PassangerLastName ln)
        =   listBookings (λb → b.lastName==ln && isNothing b.seat)
        >>= λbs → case bs of
            [] = return Nothing
            fs = enterChoice "Please choose passenger:" fs >>= return o Just
```

In the next step, the found booking record is validated. If some simple conditions hold, the passenger is kindly asked to prove her identity:

```
verifyBooking :: (Maybe Booking) → Booking
verifyBooking Nothing                  = throw "Passenger cannot be found"
verifyBooking (Just b) | isJust b.seat = throw "Passenger is already checked-in"
verifyBooking (Just b) =  viewInformation "Passenger:" b
                          ||-
                          enterInformation "Please enter you id number:"
    >>= λid → if (fromHidden b.pid==id) (return b) (throw "Identification...")
```

The final missing piece, the `chooseSeat` function, lets the passenger choose a seat using `enterChoice` by the list of free seats stored in the `Flight` record:

```
chooseSeat :: (Maybe Flight) → Seat
chooseSeat (Just f)
      = enterChoice "Please choose seat:" (map toString (sort f.free))
    >>= return o fromString
chooseSeat Nothing = throw "Flight information cannot be found"
```

Figure 2 shows the screenshots of the application. As it can be seen, the user interfaces are automatically generated from the type of the tasks only. Nevertheless they commonly look fine and intuitive to use. The only exception in this example is the fourth screen shown; choosing a seat from a list of seat numbers is anything but user friendly. In the next section we develop a more intuitive UI component, a tasklet, for choosing a seat by looking at the layout of the airplane.

3 Introduction to tasklets

Tasklets are designed for the development of interactive web components in a single-language manner. With this extension iTask3 becomes a multi-tier programming language since all the different tiers of the web application can be programmed in the single language Clean.

However, despite the common basis, there are many important differences to most multi-tier programming languages. First of all, tasklets are *not* for the development of complete, customized applications. It is designed to develop independent *components* to be attached to the generated trunk of an iTask application. As such, we decided not taking the usual lightweight, view-centric web

development approach but enforce the *model-view-controller* user interface design in tasklet development. We believe that the separation of roles suits better the development of components and it is more consistent with the objectives of iTask. This heavyweight approach also fits better for a lazy, purely functional language like Clean, where the expression of side-effects needs special attention.

Tasklets are designed to be *independent* in the sense that no facility is provided to initiate communication with other server or client components. One can argue that this imposes limitations, however in our experience, it suits well typical tasklets and enjoy an important advantage: this way the communication between the client and server components can be completely *implicit*. Any argument can be passed to a tasklet by enclosing it into a closure of the tasklet and the result is automatically shipped to the server when it is needed. The developer does not even have to be aware of programming different tiers. The accessible resources are statically controlled by the unique type that appears in the signature of the function.

Tasklets are defined by the means of the `Tasklet st val` record type. It has two type parameters denoting the type of the internal state (the *model*) of the tasklet (`st`) and the type of its result value (`val`):

```
:: Tasklet st val = { generatorFunc :: (*World → *(TaskletHTML st, st, *World))
                    , resultFunc    :: (st → TaskValue val)
                    }
:: TaskValue a    = NoValue | Value a Stability
:: Stability   :== Bool
```

During initialization, `generatorFunc` is executed on the server to provide the initial state and user interface of the tasklet. Its only argument, a value of the unique type `*World`, allows access to the external environment. The current value of the tasklet is calculated when necessary by `resultFunc` from its internal state. The result type, `TaskValue a`, an iTasks system type, expresses that the result of a task execution can be an actual value (`Value`) which is *stable* or *unstable*, or can indicate no meaningful value (`NoValue`). For the explanation of value stability, please refer to [14], in this paper we always use stable return values, which basically tells the task engine that the computation of the actual task is finished. The user interface (the *view*) and its behavior (the *controller*) are defined by the `TaskletHTML` structure:

```
:: TaskletHTML st = { html          :: HtmlDef
                    , eventHandlers :: [HtmlEvent st]
                    }
:: HtmlDef = ∃a: HtmlDef a & toHtml a

:: HtmlEvent st = HtmlEvent HtmlElementId EventType (EventHandlerFunc st)
:: EventType    = OnClick | OnMouseOver | OnMouseOut | ...
:: EventHandlerFunc st :== (st JSValue *JSWorld → *(st, *JSWorld))
```

The actual user interface (`html` field) can be given by any data structure provided that it has an instance of the function class `toHtml`. In the following, we will use an overly simplified ADT to create HTML definitions which suits well

our straightforward example, however may not satisfying for more complicated ones. Core iTask already supports the generation of high-level web forms based on the iData [12] toolkit. In this case full, low-level control over the definition of HTML elements is needed. This can be done in an abstract, monadic way like in Wash [19] or by an XML like domain specific language similar to that of Hop. Furthermore, the MVC concept enables that the three components can be developed separately, and specifically allowing the View to be developed by non-programmers. For this reason, some template mechanism also could be considered to be added similar to e.g. Yesod [17] or Snap [3]. However, providing any particular tool here would beyond the scope of this paper.

The run-time behavior, the *controller* part, of a tasklet is encoded in a list of event handler functions (`eventHandlers` field). Event handlers are defined using the `HtmlEvent` type. Its only data constructor has three arguments: the identifier of an HTML element, the type of the event and the event handler function. During the instantiation of the tasklet on the client, the event handler function is attached to the given HTML element to catch events of the given type.

The event handler functions work on the JavaScript event object (a value of type `JSValue` in Clean) and the current internal state of the tasklet. They also have access to the HTML Document Object Model (DOM) to maintain their appearance. The DOM is a shared object from the point of event handlers, therefore it can be manipulated only the way as IO done in Clean, through unique types. That is, accessing the DOM is possible only using library functions controlled by the unique `*JSWorld` type. This type is used in a similar way as the type `*World` on the server. Introducing a new type to have IO on the client has the advantage that reflects for the different purposes of client and server side code. The server code can access all resources of the server computer, like the file system, not available on the client; at the same time, the client code has external access to a resource accessible only on the client: the DOM.

Following the tasklet definition, a wrapper task must be created to hide the behavior of the tasklet behind the interface of a task:

```
mkTask :: (Tasklet st a) → Task a
```

The life cycle of a tasklet starts when the value of the wrapper task is requested. First, `generatorFunc` is executed on the server to provide the initial state and user interface of the tasklet. Then, the initial task state and the event handlers defined in Clean are on the fly compiled to JavaScript and, along with the UI definition, shipped to the browser. In the browser, the HTML markup is injected into the page and the event handlers are attached. As events are fired, the related event handlers catch them, and may modify the state of the tasklet and the DOM. If the state is changed, `resultFunc` is called to create a new result value that is sent to the server immediately. The life cycle of the tasklet is terminated by the framework when the result value is finally taken by another task.

3.1 Seat choosing by map

To clarify the usage of tasklets, we enrich our example with the aforementioned seat chooser component. So far the passenger was to choose a seat from the list of available seats by their designation. The new idea is to allow the user choosing by looking at a simplified seat map of the airplane as it is shown in Figure 3. For this, the Flight record is extended with layout information:

```
:: Flight = { ...
            , rows   :: Int    // Number of rows
            , layout :: [Int]  // Layout of a row
            }
```

The `rows` and `layout` fields contain the number of rows on the plane and the layout of the rows, respectively. If the layout value is [2,3], rows consist of 5 seats in 2 groups: 2 seats, corridor, 3 seats.

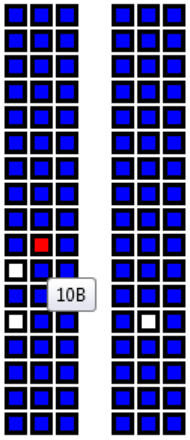


Fig. 3. Choosing a seat

The signature of `chooseSeat` does not have to be changed, we simply redefine its body:

```
chooseSeat (Just f) = mkTask seatChooserTasklet where
```

The internal state of the tasklet in this simple case is `Maybe Seat`. This expresses that a seat is already chosen or has not been yet. At the beginning it is `Nothing` (second value of the result of `generatorFunc`). According to `resultFunc`, the tasklet results in the chosen seat if its state is not empty, otherwise no meaningful value is propagated.

```
seatChooserTasklet :: Tasklet (Maybe Seat) Seat
seatChooserTasklet =
    { generatorFunc = (λworld → (TaskletHTML gui, Nothing, world))
    , resultFunc    = maybe NoValue (λv → Value v True)
    }
```

The `rowLayout` function transforms the row layout description to a list of seat numbers where corridors are denoted by `-1`:

```
rowLayout = intercalate [-1] (numbering 1 f.layout)
numbering i [] = []
numbering i [x:xs] =[take x [i..] : numbering (i+x) xs]
```

The result of this function can be straightforwardly mapped to HTML elements in `genRowUI`. In this example, we use only one data constructor of an overly simplified ADT to create HTML markup. The different kind of seats and the corridors are all mapped to HTML `div` elements using the `DivTag` data constructor. It has two list arguments, the first contains the description of the attributes, like `TitleAttr`, `IdAttr` and `StyleAttr`, and the second one contains child elements. For the sake of readability and simplicity the style attributes `corridorStyle`, `freeStyle`, `occupiedStyle` and `newRowStyle` are neglected.

```
genRowUI (Seat _ -1) = DivTag [corridorStyle] []
genRowUI seat | elem seat f.free
```

```
    = DivTag [TitleAttr (toString seat), IdAttr (genSeatId seat), freeStyle] []
    = DivTag [TitleAttr (toString seat), occupiedStyle] []

seatMap = DivTag [] (intercalate [DivTag [newRowStyle] []]
            [map (λs → genRowUI (Seat r s)) rowLayout \\ r ← [1 .. f.rows]])
```

The `genRowUI` function also takes into account whether the seat is still vacant or not. If a given seat has not been occupied yet, it gets different color and a HTML `id` attribute for the later attachment of event handlers. Finally, function `seatMap` generates and merges the markups of different lines. The special style attribute `newRowStyle` forces the browser to wrap subsequent `div` elements to the next line. The function `genSeatId` generates unique identifiers for HTML `id` attributes from a value of type `Seat`.

Now that we have defined the actual user interface, it is time to assign behavior to it. A seat should be chosen by simply clicking on it, furthermore, we would like the free, selectable seats to be highlighted when the mouse pointer is over them.

```
attachHandlers seat =
    [ HtmlEvent (genSeatId seat) OnClick (setState (Just seat))
    , HtmlEvent (genSeatId seat) OnMouseOver (setColor "red")
    , HtmlEvent (genSeatId seat) OnMouseOut (setColor "white")]

setState nst  _ _ w  = (nst, w)
setColor clr st e w  = (st, setObjectAttr e "target.style.backgroundColor" clr w)
```

Three event handlers are attached to each `div` element representing free seat. Clicking on one of them, the internal state of the tasklet is changed to indicate the corresponding seat. This triggers the execution of `resultFunc` which creates a value result to send to the server. As for highlighting, the color of the event target is changed on moving mouse over and out.

Setting the state is done by creating a closure of the `setState` function. It is an event handler function which does nothing more than return its first argument as the new state. The `OnMouseOver` and `OnMouseOut` event handlers also create a closure of the function `setColor` which simply set the background color of the target of the event. This is done by the `setObjectAttr` library function which sets an attribute of a JavaScript object. This function has a side effect thus the `*JSWorld` type appears in its signature. It takes a reference to an external object (`JSValue`), the name of an attribute and an arbitrary value. The value is converted to its JavaScript equivalent then the attribute of the object is set.

```
setObjectAttr :: JSValue String a *JSWorld → *JSWorld
```

The tasklet run-time system is shipped with a library which contains a large set of interface functions, similar to that of `setObjectAttr`. These functions enable tasklets to directly interface with the enclosing JavaScript environment, e.g. to access the HTML Document Object Model (DOM), create arbitrary JavaScript objects (including HTML elements), read/write/create object attributes, or execute methods of JavaScript objects. This low level, general library provides

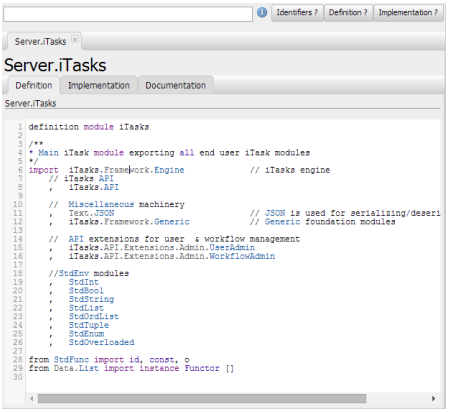
unrestricted access to the JavaScript environment, and enables the development of arbitrary higher level, special purpose libraries on top of it.

Finally, the last piece is the `TaskletHTML` record to assign the view (the HTML markup) and controller (the event handlers) components:

```
gui = { html          = HtmlDef seatMap
      , eventHandlers = concatMap attachHandlers f.free
      }
```

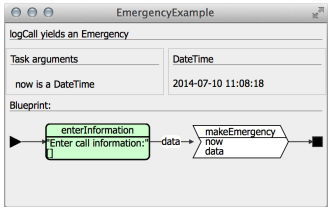
4 Use case studies

In this chapter, two real-world use cases of the presented tasklet architecture are discussed to prove its usefulness. Both examples are taken from ongoing projects of the iTask development team, and part of the current version of the iTask system.



The first of these projects aims the port of the Clean integrated development environment, the Clean IDE, to the iTask system. With this development, we believe to achieve a web based multi user development environment, and to be able to refine the semantics of the iTask combinators in the same time. The iTask system excel at generating traditional graphical user interfaces, however, there is one component, namely the *source code editor*, which cannot be generated in any way. Thus, we decided to develop a tasklet based on the CodeMirror JavaScript text editor component. The tasklet we gained is well customizable using a standard functional API, and seamlessly fits into the generated user interface

The goal of the second project, called Tonic, is to develop an infrastructure to graphically represent the definition and behavior of tasks. It translates a textual iTask specification into a graphical one, called a *blueprint*. The Clean compiler has been adjusted to generate blueprints, and a standalone application, a Tonic viewer, written in iTask, is developed to visualize them. Such a blueprint is basically a general *graph*, which consists of special kind of annotated nodes and edges. To be able to draw graphs, a general tasklet is developed. This tasklet is able to create a graphical representation of a graph `Graph n e`, provided that a



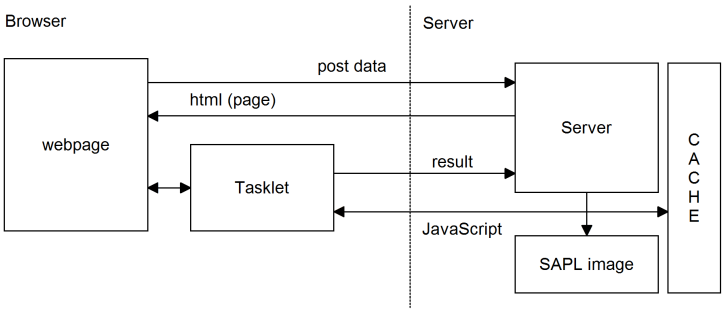


Fig. 4. The architecture of client-side execution

`GraphletRenderer n e` instance for the given node and edge types exist. The graph is given in a standard, functional way, while the renderer must provide a description understandable by the D3 JavaScript library, on which the tasklet is based.

5 The architecture of client-side execution

The client-side execution architecture is designed in such a way that the two groups of functions, executed on the client versus executed on the server, are not designated during compilation. Instead of this, two images of the same application are produced by the Clean compiler: the server executable running in native code and an intermediate representation that can be compiled to JavaScript (see Figure 4). For the intermediate representation, the so called Simple Application Programming Language (SAPL) [7], a core, lazy functional language is utilized. It is used to execute arbitrary Clean expressions in the browser as follows:

0. There are two images produced by the Clean compiler: a server image (native code, executable) and a SAPL image (intermediate representation);
1. The executable on the server is started;
2. Instead of evaluating an expression on the server, one can decide, at run-time, to evaluate it on the client instead. This can in principle be done for any expression;
3. The expression to evaluate on the client is at run-time converted to an equivalent SAPL expression;
4. This SAPL expression is passed to the run-time linker specially developed for this purpose. The linker collects the dependencies of the expression recursively using the SAPL image of the application;
5. The result is run through a caching mechanism to filter out SAPL code already processed in a previous session;
6. The remaining SAPL code is on the fly compiled to efficient JavaScript code by a newly developed SAPL-to-Javascript compiler [6];

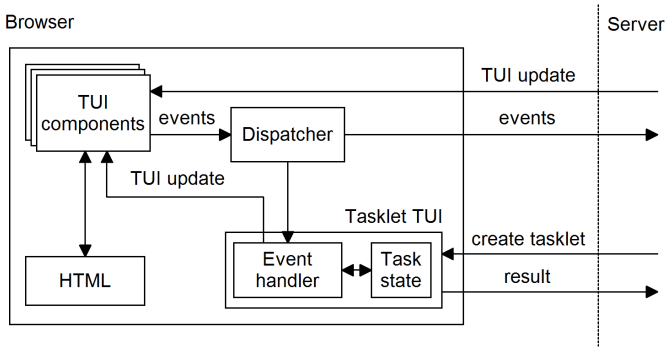


Fig. 5. The generalized tasklet architecture

7. The generated JavaScript code can be used e.g. by tasklets to perform computation in the browser.

Therefore arbitrary Clean expressions of an iTask application can be executed in the browser. Furthermore this is done by minimizing the necessary JavaScript code shipped to the client which has the advantages of reducing communication cost and memory usage of the browser.

6 Task evaluation on the client

Executing tasks is an intricate job compared to executing ordinary functions because tasks have interactive behavior which needs life cycle management. The difficulties can be understood by seeing the big picture of task *evaluation logic*.

A task basically consists of a *state* and a state *transition function*. When the state transition function is executed, it produces (1) a new state (2) an abstract description of the user interface of the task (hereafter Task User Interface, TUI) and (3) an observable task value. Based on this, task execution involves the following steps:

1. The state transition function is executed on the server to create the user interface and the result value;
2. The result value can be observed by other tasks; they can decide to continue with this current value. In that case the observed task is terminated;
3. The user interface information is sent to the browser to display;
4. If any event occurs on the client, it is passed to the state transition function on the server and the procedure continues with step 2.

The standard way tasks are evaluated closely fits the architecture of tasklets: (1) there is a distinct state to work on (2) the state transition function generates a new state and user interface just as we need in `generatorFunc` (3) the user interface generates events (4) event handlers modify the state and the user

interface (see step 4). The consequence of this perfect fit is that it is possible to define *one* general tasklet creator to run *any* task exclusively in the browser:

```
runOnClient :: (Task a) → Task a
```

The result of the `runOnClient` task is a tasklet in which the state transition function of the enclosed task is utilized in `generatorFunc` and in the event handlers. Neglecting any details, at this point the tasklet API was slightly *generalized* to enable these functions to create and interact with TUI elements in addition to HTML. When the value of `runOnClient anyTask` is requested, the state transition function of `anyTask` is called on the server to create the initial user interface and state of `anyTask`. These, and the JavaScript counterpart of the event handlers (implicitly containing the state transition function) are sent to the browser. Figure 5 summarizes the client part of the generalized architecture:

1. In the browser the TUI elements are displayed;
2. The events emitted by the TUI are passed to an event dispatcher function which can decide if the target of the event runs on the server or on the client;
3. In the latter case the event is forwarded directly to the wrapper tasklet running on the client instead of being sent to the server;
4. The event handler of this tasklet executes the state transition function of `anyTask` on the client to create a new state, result value and TUI definition;
5. If the result value is changed, it is shipped back to the server;
6. The user interface is updated by the TUI definition resulted by the state transition function and the procedure continues with step 2.

6.1 Limitations

As for the current implementation there are some restrictions to the applicability of the tasklet architecture. Some of them derives from the limitation of the Clean to SAPL compiler and give constraints on the application of Clean language elements: (1) tasks evaluated on the client can only produce higher order functions as intermediate value. Higher order values cannot be returned as final result, because the de-serialization of SAPL expressions into a Clean executable is possible only in the case of first order values; (2) certain tasks are intended to be executed on the server e.g. when a database is accessed, or global information is shared between distributed tasks. Such tasks cannot easily be shipped to the client, still a general solution is possible using a server side mediator service which is being under development.

7 Related work

The iTask3 system with the tasklet extension is a unique multi-tier programming language. In contrast to most web programming languages where the functionality is view-centric, built around the user interface, iTask proposes an *inverted* development model: the trunk of an iTask application is generated by a functional specification then augmented with custom web components.

Several other languages address multi-tier programming. In the imperative world the most modern approach is the Google Web Toolkit (GWT) [1], Google Dart [5] and Node.js [18]. GWT utilizes a Java to JavaScript compilation technique for building complex browser-based applications. GWT fosters classical GUI programming where widgets can be developed using a programming model comparable to that of tasklets.

The Dart language and the Node.js framework take a different approach. They enable multi-tier programming by providing a run-time environment of their languages for both client and server side. The language of Node.js is JavaScript, which is native in the web browsers; the framework also provides a run-time environment, including IO libraries, for the server side. Dart is a programming language developed by Google specially designed for web application engineering. On the client, it compiles to JavaScript, on the server it is executed by a Dart virtual machine. However, these systems have a more general approach than iTask and tasklets, they still share the idea of using the same language on both client and server side and implicitly bridging the communication between them.

Hop [15, 16] uses a declarative approach. It is a dedicated web programming language with a HTML-like syntax built on the top of Scheme. Hop uses two compilers, one for compiling the server side program and one for compiling the client-side part. The client side part is only used for executing the user interface. Hop uses syntactic constructions for indicating client and server part code. The application essentially runs on the client and may call services on the server. In contrast, an iTask application essentially runs on the server and may execute services, tasklets, on the client.

Links [4] and its extension Formlets is also a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively.

The iTask framework differs from the latter two by fostering a non view-centric approach even in the component development. Links and Hop have extended syntax for embedding XML descriptions in the language; this is used to mix the user interface definition and the behavior of the application. During tasklet development the model-view-controller user interface design is enforced to separate these roles.

Another important difference is that tasklets blur the boundaries of different tiers. Links uses location annotations, Hop utilizes special syntactic construction to denote the target tier of a given function or expression. In tasklets this is implicit (basically the controller role runs in the browser) but unconcerned. If a function is pure, it does not matter where it is executed. If it is not pure, the available resources are controlled statically by the signature of the function. Furthermore, the communication between the tiers is also implicit for tasklets.

As for iTask, there are earlier implementations of similar features utilizing a Java written SAPL interpreter [7] as a browser plug-in. The iEditors [8] en-

ables the development of interactive web UI elements as tasklets do, however, it does not allow direct access to browser resources, therefore its applicability is restricted to functionality provided by the plug-in. As a consequence, it does not have the single-language property either, because for some functionality the plug-in has to be extended using Java. There also had been client-side task evaluation attempts for an early version of iTask using the same plug-in based interpretation technology [13]. However, our approach, to give one general solution for both of the problems is a novel strategy.

8 Conclusion

In this paper we have presented a number of contributions to the iTask3 system, a web-enabled combinator library written in the lazy functional language Clean. In iTask, complex, multi-user web applications are generated from a mere functional specification. However, up to now, the system lacks the possibility to create custom, interactive web components.

We introduced *tasklets*, an extension to iTask3, for the development of interactive web components in a single-language manner. With this extension iTask3 becomes a unique multi-tier programming language which offers an unusual web development model based on the enrichment of a generated trunk program. Furthermore, in contrast to most multi-tier programming languages, the extended iTask framework enforces the model-view-controller user interface design in component development and blurs the boundaries of different tiers.

For the execution of Clean code in the browser, a special client-side execution facility was developed. It is designed in such a way that instead of evaluating an expression on the server, one can decide, at run-time, to evaluate it on the client. The expression is compiled to JavaScript on the fly.

Finally, we showed that the presented tasklet facility can be generalized to enable the execution of ordinary tasks in the browser instead of the server by turning an arbitrary task into a tasklet. This, amongst other things, can be used to improve the responsiveness of an iTask application by avoiding the latency of communication.

Acknowledgements

The research of the first author was supported by the European Union and the European Social Fund under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

1. The Google Web Toolkit site. `http://code.google.com/webtoolkit/`.
2. P. Achten. Clean for Haskell98 programmers - A quick reference guide. Avalaible at: `http://www.st.cs.ru.nl/papers/2007/achp2007-CleanHaskellQuickGuide.pdf`, 13 July 2007.

3. Gregory Collins and Doug Beardsley. The snap framework: A web toolkit for haskell. *IEEE Internet Computing*, 15(1):84–87, 2011.
4. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. of the 5th International Symposium on Formal Methods for Components and Objects*, FMCO'06, 2006.
5. DART. Dart : structured web programming, 2011.
6. L. Domoszlai, E. Bruël, and J.M. Jansen. Implementing a non-strict purely functional language in Javascript. *Acta Univ. Sapientiae. Informatica*, 3(1):76–98, 2011.
7. J.M. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In *Proc. 6th Symposium on Trends in Functional Programming*, TFP'06, 2006.
8. J.M. Jansen, R. Plasmeijer, and P. Koopman. iEditors: extending iTask with interactive plug-ins. In *Proc. of 20th Int'l Conf. on Implementation and Application of Functional Languages*, IFL'08, 2011.
9. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug 1988.
10. B. Lijnse, J.M. Jansen, R. Nanne, and R. Plasmeijer. Capturing the Netherlands Coast Guard's SAR Workflow with iTasks. In *Proc. of the 8th Int'l Conf. on Information Systems for Crisis Response and Management*, ISCRAM'11, 2011.
11. B. Lijnse, J.M. Jansen, and R. Plasmeijer. Incidone: A task-oriented incident coordination tool. In *Proc. of the 9th Int'l Conf. on Information Systems for Crisis Response and Management*, ISCRAM'12, 2012.
12. R. Plasmeijer and P. Achten. iData for the world wide web - programming interconnected web forms. In *Proc. of 8th Int'l Conf. on Functional and Logic Programming*, FLOPS'06, 2006.
13. R. Plasmeijer, J.M. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proc. of 14th Int'l Symposium on Principles and Practice of Declarative Programming*, PPDP'08, July 2008.
14. Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 195–206, New York, NY, USA, 2012. ACM.
15. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'06, 2006.
16. M. Serrano and C. Queinnec. A multi-tier semantics for hop. *Higher-Order and Symbolic Computation*, 23:409–431, 2010.
17. Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., 2012.
18. Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. *Node.Js*. Betascript Publishing, Mauritius, 2010.
19. Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *Proc. 4th Int'l Symposium on Practical Aspects of Declarative Languages*, PADL'02, Jan 2002.