

Clean up the web!

Rapid client-side web development with Clean

László Domszalai and Tamás Kozsik

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{dlacko, kto}@pnyf.inf.elte.hu

Abstract. Programming in Clean is much more appealing than programming in JavaScript. Therefore, solutions that can replace JavaScript with Clean in client-side web development are widely welcomed. This paper describes a technology for the cross-compilation of Clean to JavaScript and for the tight integration of the generated code into a web application. Our solution is based on the iTask framework and its extension, the so-called Tasklets. The application server approach provides simple and easy deployment, thus supporting rapid development. Examples are shown to illustrate how communication between the Clean and JavaScript code can be established.

1 Introduction

Using JavaScript for the development of client-side web applications displeases the Clean programmer and former web developer writing these words. JavaScript, even despite it has some functional features, creates a hostile environment compared to Clean. Consider, for example, the lack of type safety, the ugly and sometimes unnecessarily verbose syntax, and the productivity loss caused by these. One can also miss very much the elegance of a well-designed and mature functional language, and the programmers' self-confidence enhanced by the strong type system and referential transparency.

Still, JavaScript, as the only language of the platform for browser development, is inevitable. As a consequence, several attempts have been made for cross-compiling all kinds of languages to JavaScript. It is a well-established technique considering imperative languages, but the picture is not that clear when the subject of compilation is a functional language. Even worse, compiling a *lazy* functional language, such as Clean, to JavaScript is definitely a delicate job.

The main problem is the limitation of the available resources in the browser: the run-time system imposes severe constraints on heap and stack usage. As iteration in functional languages is accomplished via recursion, stack limitation seems to be the most serious issue. A standard technique to overcome this is trampolining [16], but, as it increases the memory footprint and the running time of the application, usually it does not perform effectively enough in the case of lazy functional languages. The reason for the higher memory footprint in these languages is the need to maintain thunks, i.e. delayed computations.

As for Clean, a mature JavaScript compilation technique is available which solves these problems to an extent which is applicable for most practical tasks [5]. However, this is only half the job. Compiling a Clean program to JavaScript still involves numerous steps which hampers the development of client side web applications in Clean: (1) the Clean program must be transformed to an intermediate language using the Clean compiler, (2) this intermediate must be compiled to JavaScript using a standalone application, and (3) the generated JavaScript code must be integrated into the web application. This complex and mundane process can nullify the advantages of non-JavaScript development.

In this paper an extension to iTask and Tasklets is presented, to make the above mentioned deployment process transparent. With this extension, iTask becomes a rapid development environment, or even an application server for client side web applications written in Clean. Furthermore, this extension does not only solve the aforementioned deployment problem, but it also enables complex information interchange between the Clean and JavaScript code in a type safe manner.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to the iTask system and Tasklets. Section 3 presents what contributions the present paper makes. To that end, it illustrates the approach and some of the technical issues through three carefully selected examples. Section 4 discusses type correspondence between the JavaScript and Clean side of the code. Related work is described in section 5. Finally, section 6 concludes. The system as well the examples presented here can be downloaded from the web.¹

2 Preliminaries

The iTask system [10] is a framework for programming workflow supporting applications in Clean using a new programming paradigm built around the concept of *tasks* [15]. A task is an abstract description of an interactive persistent unit of work which delivers a value when it is executed. From a practical point of view, a task can be anything from a system call to some interaction to be performed in a web browser by a user.

iTask provides a combinator-based embedded domain specific language to specify compositions of such interdependent tasks. A complete multi-user web application can be generated from the specification of the workflow and of the different data types involved – all the details (including the 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

¹ <http://people.inf.elte.hu/dlacko/papers/rapmix/>

interface. Sometimes, even if the functional web design is satisfactory, custom building blocks may be required for the purpose of user-friendliness.

Tasklets, a recent extension to `iTask`, are introduced to overcome this short-coming [6]. Tasklets enable the development of interactive web components directly in Clean. A tasklet consists of an inner state, user interface, and behavior provided by non-pure event handler functions. The user interface can be defined in any abstract or concrete way that enables HTML code generation. The event handlers are written in Clean, but compiled to JavaScript and 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; utilizing the event-driven architecture the tasklet can achieve interactive behavior.

From a technical point of view, tasklets are defined by the means of the `Tasklet st val` record type. It has two type parameters: one of the parameters denotes the type of the internal state of the tasklet (`st`) while the other gives the type of its observable state (`val`):

```
:: Tasklet st val = { generatorFunc :: (*World → *(TaskletGUI st, st, *World))
                    , resultFunc   :: (st → Maybe val)
                    }
```

During initialization, `generatorFunc` is executed on the server to provide the user interface and the initial state of the tasklet. Its only argument, a value of the unique type `*World`, allows access to the external environment. Whenever needed, the current observable value of the tasklet can be computed from the internal state by calling `resultFunc`. This value is optional (`Maybe`). The user interface and its behavior are defined by the `TaskletHTML` structure:

```
:: TaskletGUI st = TaskletHTML (TaskletHTML st) | ...
:: 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 HtmlObject *HtmlDocument → *(HtmlDocument, st))
```

The actual user interface (`html` field) can be given by any data structure provided that it has an instance of the function class `toHtml`.

The run-time behavior 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 `HtmlObject` in Clean) and on 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 is done in Clean, through unique types. That is, accessing the DOM is possible only using library functions controlled by the unique `*HtmlDocument` type.

Following the tasklet definition, a wrapper task must be created to hide the behavior of the tasklet behind the interface of a task (tasks are represented by the opaque type `Task a`, where `a` denotes the type of the value of the 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 Rapid development with iTask

In iTask, the deployment process during development is fairly straightforward. Given an iTask task, `aTask`, by adding the following main function and running the application, an embedded web server is started, which publishes the task on the local host.

```
Start :: *World -> *World
Start world = startEngine aTask world
```

When the page is requested in the browser, first a client-side run-time environment is loaded, which manages the user interface (UI) of the tasks. The actual task is published on a special URL where it provides the abstract description of its UI as a JSON encoded descriptor object. The run-time environment can load and display such abstract UI descriptions.

A tasklet is self-contained in the sense that its UI description contains all the JavaScript code necessary to run the tasklet in the browser. Thus, to turn an iTask application into an application server for non-iTask applications, all we have to do is to provide, as a standalone JavaScript library, a small part of the aforementioned run-time environment: a part which is able to load and create a tasklet. On the server side, a list of tasklets can be published all at once:

```
Start world = startEngine [{ PublishedTask
  | url = "/test"
  , task = TaskWrapper (const testTasklet)
  , defaultFormat = JSONGui}]
world
```

This overloaded version of function `startEngine` enables the specification of a list of tasks together with the URLs where they will be published (in the example above the list had only one element).

On the client side, loading the published tasklet is this simple:

```
<html>
  <head>
    <script type="text/javascript" src="tasklet-runtime.js"/>
    <script type="text/javascript">

      loadTasklet("http://localhost/test", function(tasklet){
        tasklet.display(document.getElementById("out"));
      });

    </script>
  </head>
  <body>
    <div id="out"/>
  </body>
</html>
```

The JavaScript library `tasklet-runtime.js` is less than 10 kB compressed. It contains the logic for loading and instantiating tasklets, as well as the run-time environment of the Clean to JavaScript compiler. Function `loadTasklet` tries to load a tasklet from the URL given in its first argument. Since the loading mechanism is implemented with an asynchronous AJAX request, a call-back function must also be provided as a second argument; this will be called when the tasklet is loaded and created in the browser.

An instantiated tasklet is represented as a JavaScript object with the predefined prototype `Tasklet`. It encapsulates and hides all the properties of a tasklet (the user interface, the state and the behavior), and exposes only the `display` method which injects the UI of the tasklet into a given point in the HTML DOM.

Using the above method, an arbitrary tasklet can be included into a non-`iTask` application. However, it is still a foreign element in the application, as it runs independently and has no way for information exchange. In the following sections our solution is presented to this problem. We propose a mixed-language programming model where different parts of a web-application are written in either Clean or JavaScript, making the best use of the two languages. Each functionality can be coded in the language which suits better to the given task, and interaction is made easy between fragments written in the two languages. Rapid application deployment is supported by the concept of an application server for client-side web applications. Tasklets run embedded in a lightweight application server, which generates and supplies the JavaScript code through a standard web socket; the client side support library automatically injects this JavaScript code into the web page. At the end of program development, the application server can be eliminated: the JavaScript code generated from the tasklets can be saved into a `.js` or `.html` file, and deployed on a web server.

Our approach will be demonstrated step-by-step in the following sections through a series of example applications. What these examples have in common is the lack of a user interface and observable state of tasklets. According to the principle that in a mixed language environment both languages should be used at their best, we chose to implement control and user interaction in JavaScript, and stress pure style in the Clean code. This approach results in a very special, unconventional use of `iTask` and `Tasklets`. To indicate that a given tasklet does not encapsulate a GUI, a new data constructor `NoGUI` for the type `TaskletGUI` is introduced. Moreover, as it is used for task-to-task communication in proper `iTask` applications only, no return value (observable state) for tasklets are needed here. Therefore, in the forthcoming examples, for the creation of tasklets, we use the `initially` function which specifies the initial internal state only.

```
initially :: st → Tasklet st Void
initially st = { generatorFunc = λworld = (NoGUI, st, world)
                , resultFunc   = const Nothing
                }

```

3.1 Writing the logic of a web application in Clean

One day the need to display Clean source code in a web application, as part of a source code repository, has emerged. Many Clean developers use the integrated Clean development environment, `CleanIDE`, for programming. This environment provides excellent syntax highlighting, and Clean developers have really got used to it. Therefore the same style to present Clean code seemed highly desirable for our web application. Reprogramming the functionality in JavaScript would have been a fairly complex task. However, with our tasklet-based framework it has proven to be relatively easy. We decided to use the modules responsible for syntax highlighting in the `CleanIDE`, which meant more than 1000 lines altogether. We had to add a main module containing a tasklet definition (which mimics the `CleanIDE` for calling in the syntax highlight module) and a `Start` rule: 30 effective lines of code. Furthermore, 18 effective lines of JavaScript and 13 effective lines of HTML code had to be written only. The Clean to JavaScript compiler generated 136 kB of JavaScript from the 33 kB of Clean code, and the source code viewer was up and running. Now we take a closer look at the code.

```
:: Color ::= String

highlight :: [String] → [(String, Color)] // definition omitted

annotateI (Just dynArg) st eventqueue = (res, st, eventqueue)
  where res = case dynArg of (lines :: [String]) = highlight lines

highlighter = mkInterfaceTask (initially Void) [InterfaceFun "annotate" annotateI]
Start world = startEngine [{PublishedTask | url = "/highlighter"
    , task = TaskWrapper (const highlighter)
    , defaultFormat = JSONGui}]
    world

```

The unnecessary technical details have been omitted, as well as the body of the `highlight` function, which can be written as the composition of some functions defined already in the `CleanIDE`.

In the case of this simple tasklet, not only the GUI and the result value, but also the internal state is absent, i.e. `Void`. The only way to interact with the highlighter tasklet is to call its single *interface function*, `annotateI`, from the JavaScript code. When a tasklet is created with `mkInterfaceTask` (defined in the tasklet library), a list of interface functions can be passed. In this case, this list has a single entry: whenever the JavaScript code calls the `annotate` method of the tasklet, the code generated from the `annotateI` function is executed. This `annotateI` takes the current (`Void`) state of the tasklet and an event queue (explained in section 3.3), and returns them unmodified. Information from JavaScript to Clean is received through the second parameter, which is of type `Maybe Dynamic`. Dynamics provide dynamic typing facilities in a statically typed language [1, 14].) We expect here that a list of strings is stored in the `Dynamic`, namely the lines of some Clean source code. If the dynamic pattern matching fails, the run-time engine triggers an exception to inform the caller. The `highlight` function will be called with the lines found in the dynamic: it splits each line into tokens, and annotates each token with its colour. The token and its colour is represented as a pair of strings, a list of pairs corresponds to a line, and the list of lists is the whole program text syntax highlighted. This list of lists of pairs of strings is sent back to the JavaScript side as a component of the triple returned by `annotateI`.

To understand how types are handled in our Clean to JavaScript compiler, consider below the interesting part of the JavaScript side in our mixed-language application.

```
function onLoadTasklet(tasklet){
  var lines = prepareLines();
  var tokens = tasklet.intf.annotate(lines);

  for(var i=0; i<tokens.length; i++){
    for(var j=0; j<tokens[i].length; j++){
      var token = tokens[i][j][0];
      var color = tokens[i][j][1];
      appendToken(token, color);
    }
    appendNewLine();
  }
}

loadTasklet("http://localhost/highlighter", onLoadTasklet);
```

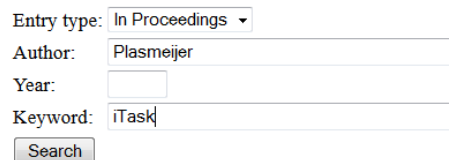
When the page is loaded, the function `loadTasklet` is executed by the browser. The tasklet is loaded from the specified URL, instantiated, and `onLoadTasklet` is called with it. This latter function first creates an array of strings (i.e. `lines`), which is passed to the `annotate` interface function of the tasklet (the interface functions are created under the `intf` namespace to avoid possible name collisions

with the original properties of the Tasklet prototype). Note that this array of strings corresponds to a `Dynamic` containing a list of strings in the Clean side (the details of the type correspondence algorithm are explained in section 4). Function `annotate` returns an array of arrays of arrays of strings, `tokens`, which is processed in a straightforward way in the `for`-loop.

Communication between JavaScript and Clean sources is, therefore, accomplished in the following way. Primitive types of Clean are represented with similar primitive types in JavaScript, while lists and tuples are represented by arrays (an n -tuple is represented as an array of length n , e.g. 2 in our example). Algebraic types are also represented by arrays – the name of a data constructor is stored in the first element of such an array. Values from JavaScript correspond to `Dynamic` in Clean, so that pattern matching on types in the Clean side facilitates type-safe programming. This has been an example of an interface function with a single argument. Interface functions with no arguments receive a `Nothing`, and those with multiple arguments will find a tuple in the `Dynamic`.

3.2 Adding state and interaction

Suppose you must write some interactive presentation logic to be executed in a browser. For example, you want to display the bibliographic data of your publications in a searchable, filterable way on the web (Fig. 1). The application should receive a `BIBTEX` file as input, and parse, filter and pretty-print the entries found in this file. To write a client-server application for this, and implement parsing and filtering on the server would be too much hassle. It is more reasonable to send over the data to the browser all at once, parse it, and then let an interactive client side application filter the data and display the selected items. Coding all these activities in JavaScript is not what you would like to do on a rainy



[1] Rinus Plasmeijer and Bas Lijnse and Steffen Michels and Peter Achten and Pieter W. M. Koopman. Task-oriented programming in a pure functional language. In *PPDP*, 2012. [\[DOI\]](#)

[2] Rinus Plasmeijer and Bas Lijnse and Peter Achten and Steffen Michels. Getting a grip on tasks that coordinate tasks. In *LDTA*, 2011. [\[DOI\]](#)

[3] Rinus Plasmeijer and Peter Achten and Pieter W. M. Koopman and Bas Lijnse and Thomas van Noort and John H. G. van Groningen. iTasks for a change: type-safe run-time change in dynamically evolving workflows. In *PEPM*, 2011. [\[DOI\]](#)

Fig. 1. Web application for filtering bibliographic data

Friday afternoon. Contrarily, much of the functionality is fairly straightforward to develop in Clean, using higher order functions. To implement parsing, for instance, the Parser Combinator library of Clean may prove useful. It turns out that tasklets are a valuable tool for building this application.

The main difference between this and the syntax highlighter application is that interaction with the user is required, and that there is some state that should be preserved between user interactions. We suggest that the state should be stored in the JavaScript side of the code, and state-to-state functions should be written in Clean. The following fragments present the interesting parts from the JavaScript side of the code.

```
var entries;
var tasklet;

function onLoadTasklet(aTasklet){
    tasklet = aTasklet;
    entries = tasklet.intf.init();

    var refs = prepareReferences();

    for( var i=0; i<refs.length; i++ )
        entries = tasklet.intf.parse(entries, refs[i]);

    display_bibitems(entries);
}
```

The state of the application, stored in the global variable `entries`, represents all the entries of the BIB_T_E_X file. Right after the page is loaded and the tasklet is created, function `onLoadtasklet` will be called, which parses the bibliography items. First, it creates the initial state by calling the `init` interface function, then the bibliography items are parsed and added to the state one by one using the `parse` interface function of the tasklet. Parsing is performed in such a “per item” basis as a precaution only – otherwise, in the case of a long bibliography list, like that of Rinus Plasmeijer, parsing might run out of stack.

Whenever the user interacts with our application, namely when the search button on the web page is pressed, function `search` will be called. It filters the bibliography items, again using interface functions of the tasklet.

```
function search(){
    var selected = entries;

    var year = document.getElementById("year").value;
    if( year != "" ) selected = tasklet.intf.filter(selected,"year",year);
    // similarly for entry type and author

    var keyword = document.getElementById("keyword").value;
    if( keyword != "" ) selected = tasklet.intf.search(selected, keyword);

    display_bibitems(selected);
}
```

Similarly to the syntax highlighter, this tasklet is also stateless and provides no GUI. It does not make use of `eventqueue` either.

```
bibtex = mkInterfaceTask (initially Void)
  [ InterfaceFun  "init"  initI
  , InterfaceFun  "parse" parseI
  , InterfaceFun  "toString" toStringI
  , InterfaceFun  "filter" filterI
  , InterfaceFun  "search" keywordI]
```

The interface functions of the tasklet have a similar structure to that of `annotateI` in the previous example. The only argument they use is the one of type `Maybe Dynamic`, on which they pattern match. The `filter` method calls in the JavaScript code, for instance, has three actual arguments, therefore the dynamic in the corresponding Clean function, `filterI`, should be a triple.

```
filterI (Just dynArg) st eventqueue = (dynamic res, st, eventqueue)
  where res = case dynArg of
    ((entries,tag,value) :: ([Entry],String,String))
      = filterEntries entries tag value
```

Section 4 will explain why `res`, the result from filtering is wrapped in a `dynamic`.

3.3 Even more state and even more interaction

In the `BIBTEX` example, the state of the application was stored in the code written in JavaScript, and the internal state of the tasklet was `Void`. Our next challenge is to write a game for solving Rubik's cube – but now in this application a stateful tasklet will be used. Similarly to the previous examples, the tasklet will have neither a GUI nor an observable state, and it will provide interface functions available for the controlling JavaScript side of the code.

The level of interactivity is much higher in this example than in the previous one. The Rubik cube is controlled by moving the mouse and by pressing some keys; the cube is rendered (Fig. 2) on-the-fly by the Clean side of the code when

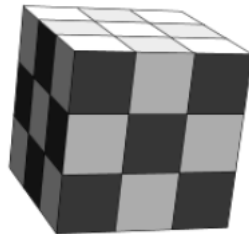


Fig. 2. Rubik's cube rendered in Clean, drawn by JavaScript

its state is changed. Another interesting issue in this example is how information flows between the pure Clean side of the code and the impure JavaScript side. Although the tasklet depends on information from the outer environment (the browser), and has an impact on its environment as well, referential transparency is not violated.

This is achieved by using a technique we call *the method of the blind chess player*. A blind chess player cannot observe the chessboard in any way, only possesses a mental picture, an inner representation of the board. The blind chess player depends on an independent observer to announce the movements. The blind chess player cannot even move the pieces directly, but can ask someone to carry out the movement. The first sign of using this technique is that from the nine interface functions of the tasklet, eight are merely used to delegate events. When such an event is delivered, the inner state of the tasklet is updated, the cube is re-rendered, and finally displayed.

```
rubik = mkInterfaceTask (initially (State standard (pi/10.0,pi/10.0,0.0) Nothing))
    [ InterfaceFun "display" displayI
    , InterfaceFun "mouseDown" mouseDownI
    , InterfaceFun "mouseUp" mouseUpI
    , InterfaceFun "mouseMove" mouseMoveI
    , InterfaceFun "turnLeft" (turnI fst left)
    , InterfaceFun "turnRight" (turnI fst right)
    , InterfaceFun "turnUp" (turnI snd up)
    , InterfaceFun "turnDown" (turnI snd down)
    ]

:: R3    ::= (Real,Real,Real)
:: Color ::= String
:: Cube ::= (R3 → Color)
:: State = State Cube R3 (Maybe (Int,Int))
```

The internal state of the tasklet will keep track of the actual configuration of the cube (initially it is the “standard” configuration, explained a bit later), an angle describing the viewpoint of the user (R3), and the mouse coordinates if the mouse is pressed (initially it is not). Note that the second and the third components in the internal state of the tasklet describe the state of the user interface.

To model Rubik’s cube, we follow Péter Diviánszky.² The cube is placed in such a way that its size is $3 \times 3 \times 3$, its middle point is the origin of the Cartesian coordinate system and its edges are parallel to the axes. The representation is given as a partial function $R3 \rightarrow Color$, which assigns a color to the middle point of each of the 6×9 small faces of the cube. The operations, namely rotating the cube and twisting one of the 6 layers, can be implemented by composing functions that describe coordinate transformations, for instance $left(x,y,z) = (z,y,\sim x)$. The initial, standard configuration can be given in the following way.

² http://pnyf.inf.elte.hu/fp/Rubik_en.xml

```

standard (x,y,z)
  | abs x > abs y && abs x > abs z = if (x < 0.0) "green" "blue"
  | abs y > abs x && abs y > abs z = if (y < 0.0) "yellow" "white"
  | otherwise                       = if (z < 0.0) "orange" "red"

```

Now we come to an essential question, namely how to display this cube from a pure environment. A trivial solution would be to return the list of polygons from an interface function and let the JavaScript side to display it. However, that would clutter the interface of the tasklet and would move a substantial part of the algorithm from Clean to JavaScript. Therefore, another solution was chosen: the tasklets are allowed to fire events just as arbitrary JavaScript objects can do. In the JavaScript side, functions can be subscribed to these events.

```

tasklet.addListener("draw", function(event){

    var v = event.value;
    var color = v[1];
    var p1 = v[0][0];
    ...

    drawPolygon(p1,p2,p3,p4,color);
}

```

The `displayI` interface function computes a 2D projection of the cube from the viewpoint of the user (`polygons`), asks the JavaScript side to clear the display, and asks it again and again to draw each polygon with the appropriate color. To achieve this, the function fires events `clear` and `draw`.

```

displayI :: (Maybe Dynamic) State *EventQueue → *(Void, State, *EventQueue)
displayI Nothing st=(State cube angle _) eventqueue
  # eventqueue = fireEvent eventqueue "clear" Void
  # eventqueue = foldl (λq p → fireEvent q "draw" p) eventqueue polygons
  = (Void, st, eventqueue)
  where polygons = project cube angle

```

In Clean unique types (`*EventQueue` here) are used to thread effectful computations in a pure functional way. Since `fireEvent` interacts with the outside world, namely with the user interface of our application, a “new event queue” is formed after each invocation of `fireEvent`, and the previous event queue is “consumed”. However, this is not enough to preserve referential transparency. What is missing is that events are not allowed to interfere with the interface function that triggers them. No return value is coming back to the Clean side from the JavaScript function(s) triggered by an event, and there is no means to access the outside world from the Clean side other than through the parameters of the interface functions. Type `*EventQueue` is abstract; it can only be used to ensure that events are delivered, and to define the order of event delivery. Due to this mechanism, the meaning of an interface function does not depend on *when* the event handlers are executed in the JavaScript side. They can be executed either interleaved with the Clean side of the code (i.e. by synchronous method calls) or asynchronously, after the completion of the interface function.

The division of labour described above is advantageous: pure definitions are written in Clean, while in JavaScript only the control and the effectful user interactions are implemented. In this application, for example, the JavaScript side is responsible for drawing polygons (it is straightforward in JavaScript using its browser-independent primitives), for capturing pressed keys and mouse events, and for doing some hacks to make the application work with different browsers. Altogether the JavaScript side is made up of a few dozens of effective lines of code here, such as the one catching key events.

```
function key(event){
  switch(event.charCode){
    case 119: tasklet.intf.turnUp();   break;
    case  97: tasklet.intf.turnLeft();  break;
    case 115: tasklet.intf.turnDown(); break;
    case 100: tasklet.intf.turnRight(); break;
  }
}
```

Most of the application, that is, roughly 200 effective lines of code, is written in Clean. All the decisions, all the difficult parts are in the Clean side. For instance, those interface functions of the tasklet which are partial applications of `turnI` make decisions on what to do with the key events based on the tasklet state, viz. whether rotate the cube (if the mouse button is not pressed) or twist a layer (if the mouse button is pressed over a polygon which belongs to the 2D projection of a layer of the cube).

```
turnI _ rotation Nothing st::(State cube angle Nothing) eventqueue
  = displayI Nothing (State (cube o rotation) angle Nothing) eventqueue
turnI selector rotation Nothing st::(State cube angle (Just coord)) eventqueue
  = displayI Nothing (State new_cube angle (Just coord)) eventqueue
  where polygons = project cube angle
        new_cube = case (select_layer polygons coord) of
                      Nothing      = cube
                      Just layer    = twist cube layer selector rotation
```

Some details of the definition are left uncovered here, and some other details were left out completely in order to increase readability – for the precise definitions the Reader can look up the code of the example on the web.³

4 Type correspondence in parameter passing

The communication between the JavaScript side and the Clean side of the code is bidirectional. The JavaScript side calls the interface functions of tasklets, passing arguments and expecting results. Moreover, the Clean side fires events, with parameters attached, and the JavaScript side may observe these events and receives their attached parameters. In both cases information exchange between the two sides is achieved through pass-by-value parameters and, in the first case,

³ <http://people.inf.elte.hu/dlacko/papers/rapmix/rubiksource.html>

through pass-by-value return values. The proper transmission of data requires a consequent correspondence between Clean types and JavaScript types. Certain types carry over between the two languages quite straightforwardly, others need special encoding.

It must be emphasized, however, that when we talk about the Clean side of the code, we actually mean some JavaScript code that was generated from Clean code by our cross-compiler. For clarity, we will refer here to the JavaScript side of the application as JS code, and to the code generated from the Clean side as JS* code. JS* uses a special run-time encoding of Clean types. For details on this encoding, the Reader is referred to [5].

To facilitate information exchange between Clean and JavaScript, a conversion from the JS* encoded values to JS is provided. The programmer could use the JS* encoded values in the JS code directly, but the structure of the encoded values is quite unnatural. Therefore, our runtime environment converts JS* values to JS values that are easier to use. As the examples in section 3 revealed, (1) during conversion primitive types are preserved; (2) the encoding of lists and tuples of Clean are converted to arrays; (3) algebraic types are also represented by arrays, where the name of a data constructor is stored in the first element of such an array. The conversion of functions in JS* to JS is not supported in the current version of the system. Handling partially applied functions and lazy arguments would demand special care of these values on the JS side, which, in our opinion, is not worth the effort.

The opposite direction, however, is not that simple. Clean has a much richer type system than JavaScript, thus JS values cannot be converted to JS* unequivocally. A further problem is that JavaScript is dynamically typed, and thus special care must be taken to avoid passing values of wrong type from JS to JS* and prevent run-time errors. Due to laziness, these run-time errors would emerge in the most unexpected moments.

A solution to overcome these problems is based on the *dynamics* feature of Clean. A value of an arbitrary Clean type can be converted to the special type `Dynamic`, then later the value of such a dynamic can be extracted by run-time pattern matching on the enclosed type using an algorithm called *type unification*.

When a value is passed from JS to JS*, the run-time environment tries to convert it to a `Dynamic` first. Obviously, this cannot be done in every case, but using the following (conservative) unification rules the most frequently occurring cases are covered.

1. JS booleans can be unified with Clean `Bools`.
2. Although JS has no special character type, strings of one length can be unified with `Char` in Clean.
3. There are no separate integer and floating point types in JS, a JS integer value can be unified with both `Int` and `Real` in Clean.
4. Non-integer numbers can be unified with Clean `Reals` only.
5. JS strings can be unified with the `String` type of Clean.
6. An array of JS values can be unified with a Clean list type, if
 - (a) all of its elements can be determined by the preceding rules,

- (b) they have the same type, and
 - (c) this type is equivalent with the type parameter of the Clean list.
7. In all other cases type unification fails.

Finally, there is one more important case to consider. As the `BiTeX` example revealed, it can be very useful to allow passing `JS*` values of some intricate type to `JS` as a state. Such a value is not supposed to be used directly by the `JS` code, it is only to be passed around between interface calls. Unfortunately, the `JS*` to `JS` to `JS*` conversion of such an intricate value would destroy the original type. In this case we allow the `JS*` code to pass a Clean `Dynamic` to `JS`. The run-time environment detects whether a value has type `Dynamic` and does not convert it into a `JS` value. When such a `Dynamic` is passed from `JS` to `JS*`, the run-time detects again its special nature, and does not try to recognize the type of the `JS` value, but uses its original Clean type (generated by the `dynamic` keyword) for type unification.

5 Related work

Compilation of traditional programming languages to JavaScript has drawn much attention in the last few years as client-side processing for Internet applications has been gaining importance. Virtually every modern language has some kind of technology which allows its client-side execution – see [2] for an overview.

An interesting approach to avoid the usage of JavaScript, is the so called single-language compilation technique. Single-language systems allow the development of all tiers of a whole client-server application in the same language. Those parts of the application which are needed on the client are automatically transformed to JavaScript, while the other parts are compiled to some server-side binary. Communication between the client and the server can be transparent. The most mainstream example is GWT [9] for Java. As for functional languages, the prominent representatives of this approach are Links [3] and Hop [11]. A notable advantage of single-language systems is that the whole application can be type checked. However, mixed-languages solutions, like ours, are also advantageous: one can use the best of all languages. GWT, for instance, also makes it possible to export libraries as well [12].

In this section we are particularly interested in compiler technologies for lazy functional languages, paying special attention to the deployment process and the possibilities of interacting with JavaScript.

UHC-JS is the JavaScript backend of the Utrecht Haskell Compiler [4]. Although it is still in beta stage, it can already successfully compile a fair amount of Haskell programs. Its main advantage is that the generated JavaScript code is acceptably small, albeit relatively slow. Compilation can either proceed in a per-module basis or the modules can be linked together using source code level linking. Unfortunately, in the second case the whole application has to be compiled, and the start expression cannot be specified. Its abilities to interact with JavaScript are very limited. In fact, they are restricted to a standard foreign

function interface (FFI) and some DOM manipulation libraries implemented above it.

The Fay language [7] has a unique approach – namely, it does not utilize a Haskell compiler for preprocessing, but directly parses Haskell source code using third party libraries, and generates JavaScript code from the abstract syntax tree. As a consequence, Fay supports only a limited subset of the Haskell language, which makes it less appealing for us. JavaScript interoperability is enabled through a trivial foreign function interface.

GHCJS [13] is the most promising compiler technology among those discussed here. However, it has a rather heavyweight approach compared to our solution. It compiles most Haskell libraries without a problem, but suffers from a relatively slow engine (an advanced engine is under development) and huge code footprint. It uses GHC as a front end, and JavaScript code is generated from the resulting STG. Complete interactive applications can be developed using GHCJS through non-standard support libraries, such as WebKit, bindings for WebKitGTK+, which provide a low level DOM interface, and different low and high level interfaces for JavaScriptCore. Unfortunately, due to the use of these libraries, even the most trivial application will consist of several hundred kB (or even MB) of JavaScript. On the other hand, these libraries enable the most advanced JavaScript interoperability among the compilers of study. Besides the ubiquitous FFI support, GHCJS enables callbacks to the Haskell code as well. Type safety of these calls are ensured, but limited to primitive types, like Numbers, Booleans and Strings. Furthermore, GHCJS utilizes an algebraic data type to deal with JavaScript values – this is highly limited compared to our Dynamic-based approach. The deployment process is overcomplicated, several JavaScript files are generated, and have to be included in the final application along with numerous pre-compiled libraries.

Finally, the Haste compiler [8] is a relatively new approach aiming at small code footprint and a fast engine. Currently it compiles only full applications, which sets a limit on its applicability. Haste supports calling JavaScript functions from Haskell through a standard foreign function interface.

In summary, the cross-compilers studied in this section stress the quality of compilation and the compiler infrastructure, but place no particular emphasis on deployment, and on integration of the generated code into a larger application. None of them provide a simple way for the inclusion of the generated JavaScript code into a web application as a library, and only one of them, the GHCJS, enables callbacks to the Haskell code through a type safe, albeit limited, interface.

6 Conclusions

In this paper an extension to iTask and Tasklets has been presented, which enables rapid client-side web development with Clean. The solution is basically an unorthodox application of the iTask system, which in this way becomes an application server for client-side web applications. The presented method, in terms

of deployment and integration, makes web development in Clean a competitive alternative to development directly in JavaScript. In terms of productivity, the balance is clearly tilted towards programming in Clean.

A mixed-language programming model has been proposed, where different parts of a web-application are written partly in Clean, and partly in JavaScript, making the best use of the two languages. Bidirectional communication between the two languages was a major concern. A particular strength of the ideas presented here is that instead of compiling a whole application to JavaScript, we propose to compile libraries (call-in) or components (call-in/call-out) only – the latter is achieved through events triggered by the Clean side of the applications.

Our approach enables the use of special interface and event handler functions. Furthermore, the communication interface is well typed from the point of view of the Clean code, which is achieved by the Dynamic feature of the Clean language. The applicability of the proposal has been proven through a series of carefully selected non-trivial examples.

The technology described here can be generalized in at least two ways. First, languages other than Clean can be used for writing the main body of applications. Our Clean to JavaScript compiler uses Sapl [5] (one of the core languages of Clean) as an intermediate language. A Haskell to Sapl compiler is currently under development. Besides writing a small server-side application for run-time source code level linking of Sapl and the compilation of the result to JavaScript, one technical problem must be solved: to obtain dynamically the Sapl source code of an arbitrary expression. This would make Haskell a proper replacement for Clean here.

The second option for generalization is due to the loosely-coupled communication interface between the Clean-side and the control-side of the applications. One could use platforms other than the web as a run-time environment, i.e. platforms supporting JavaScript. Such platforms are, for instance, Android and iOS, where the control logic could be implemented in Java or Objective-C, respectively; the JavaScript code generated from Clean could be used without any modifications.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.
2. Jeremy Ashkenas. List of languages that compile to JavaScript. <https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS>.
3. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proc. of the 5th International Symposium on Formal Methods for Components and Objects*, FMCO'06, 2006.
4. Atze Dijkstra. The Utrecht Haskell Compiler JavaScript Backend. <http://uu-computerscience.github.com/uhc-js/>.

5. László Domoszlai, Eddy Bruël, and Jan Martin Jansen. Implementing a non-strict purely functional language in JavaScript. *Acta Univ. Sapientiae. Informatica*, 3(1):76–98, 2011.
6. László Domoszlai and Rinus Plasmeijer. Tasklets: Client-side evaluation for iTask3. <http://people.inf.elte.hu/dlacko/papers/tasklets.pdf>, 2012.
7. Chris Done. The FAY language. <http://fay-lang.org/>.
8. Anton Ekblad. Towards a DeclarativeWeb. Master’s thesis, University of Gothenburg, Göteborg, Sweden, August 2012.
9. The Google Web Toolkit site. <http://code.google.com/webtoolkit/>.
10. Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for end-users. In *Proc. of the 21st Int’l Conf. on Implementation and application of functional languages*, IFL’09, 2010.
11. Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *Proc. 7th Symposium on Trends in Functional Programming*, TFP’07, 2007.
12. Manuel Carrasco Moñino. The Google Web Toolkit site. <http://code.google.com/webtoolkit/>.
13. Victor Nazarov. The GHCJS Haskell to Javascript translator. <https://github.com/ghcjs/ghcjs>.
14. Marco Pil. Dynamic types and type dependent functions. In *Selected Papers from the 10th International Workshop on 10th International Workshop*, IFL ’98, pages 169–185, London, UK, UK, 1999. Springer-Verlag.
15. 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.
16. David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2):161–177, June 1992.