Bachelor's Thesis Computing Science

Running iTasks tasks in the browser

Reducing server dependencies and communication overhead

Sofie Vos
s1068747

February 10, 2025

First supervisor/assessor:
Mart Lubbers

Second assessor:
Peter Achten

Radboud University

Abstract

Web developers benefit from minimizing the effort required to create user interfaces, enabling greater focus on design. The task-oriented language iTasks simplifies this process by abstracting complex implementation details and automatically generating user interfaces. However, its user interface behavior relies on server-side processing, which introduces communication overhead and creates strong server dependencies. We propose a function to mitigate these challenges. The function can run a part of iTasks, a task, in the browser, completing in a single evaluation step. It provides a foundation to further reduce the communication overhead and server dependencies.

Contents

Chapter 1

Introduction

Software plays a critical role in modern society, driving efficiency, convenience, and innovation across the globe. It enables the management and analysis of vast datasets and fosters global interaction and collaboration. However, the increasing demand for software brings significant implementation complexities, often frustrating developers as they struggle to ensure their software functions correctly.

Task-Oriented Programming (TOP) simplifies development by abstracting units of work into tasks and automatically managing their dependencies and interactions. These tasks represent computer operations, primarily focusing on user interactions, such as functions, methods, remote procedures, or web services. By abstracting units of work into tasks, developers can focus on integrating predefined tasks into interactive multi-user systems without concerning themselves with implementation details. The iTasks programming language implements the TOP paradigm, enabling tasks to automatically generate user interfaces (UIs), reducing the need for manual UI development [29]. In iTasks, UIs run in the browser while the task management is handled on the server.

Yet, managing multiple UIs from the server introduces communication overhead and creates strong server dependency. When the server is unavailable, tasks cannot function [25, p. 5]. These issues were addressed by running tasks in the browser. Around a decade ago, L. Domoszlai and R. Plasmeijer introduced the `runOnClient` function to execute tasks in the browser. This function relied on tasklets, a type of task designed to generate user interfaces [10]. Tasklets were eventually replaced by improved alternatives, leading to the deprecation of both tasklets and the `runOnClient` function that depended on them [9][34, p. 5–6].

We developed a function named `runSimpleTaskInBrowser`, which uses the current interactive tasks referred to as editors. Like `runOnClient`, our function addresses the communication overhead and strong server dependency of tasks by running them in the browser. The full implementation of our function is provided in Appendix A and achieves the following research contributions:

1. Evaluate and execute a task in the browser
2. Show any resulting UI changes from the executed task in the browser
3. Inspect exceptions of the executed task in the browser and send them to the server
4. Return a resulting task from the executed task

In the next two chapters, we explain editors and the implementation of key TOP concepts in iTasks. We then discuss two programming concepts frequently used in iTasks but not part of TOP: uniqueness and strictness. These concepts are crucial for understanding how iTasks works, which we cover in Chapter 4, and the decisions made in our research, presented in Chapter 5. Finally, we review the related work and conclude our study.

Chapter 2

TOP

We begin by explaining TOP, the programming paradigm briefly mentioned in the introduction, and how some of its key concepts are implemented in iTasks. These concepts are integral to our solution: a function that runs a task in the browser and returns its resulting task, called `runSimpleTaskInBrowser`. To understand our solution, it is essential to grasp the concept of a task and other TOP concepts, such as editors, which form its foundation.

2.1 What is TOP?

TOP is a programming paradigm that primarily structures programs around tasks, which are defined as units of work. A task can represent any activity, like sending an email or writing text. By defining tasks at a high level with clear and descriptive names, such as `sendMail` or `writeText`, developers can easily select and use tasks without needing to understand their underlying technical details. TOP, as stated earlier, including its concept of tasks, is implemented in the programming language iTasks [29].

2.2 TOP in iTasks

iTasks is a domain-specific language (DSL), embedded in the functional programming language Clean. It leverages Clean's syntax and semantics to develop task-oriented workflows [32]. As an implementation of TOP, iTasks incorporates the following core concepts [25]:

basic tasks fundamental units of work

task combinators functions that link multiple tasks together enabling different workflows

editors interactive tasks with a UI

shared data sources information sources for storing shared data

We explain how each of these concepts are implemented in the following subsections.

2.2.1 Basic tasks

Tasks are units of work with a specific goal, such as writing text. In iTasks, each basic task is of the type `Task a`. For instance, the task `writeText` would be defined as `writeText :: Task String`. Here, the `String` in `Task String`, or any type `a` in `Task a`, represents the task value that can be observed by other tasks while it updates, to make decisions upon. If `String` in `writeText` represents the text being written, other tasks can observe the text as it changes and possibly adjust their course of action accordingly. [34, p. 4–5].

You can define functions that produce basic tasks. For example, the function `return` creates a basic task with its argument as the task value. Even though the task value created with `return` is stable, meaning it does not change, other tasks can observe it for decision making [29, p. 199]. Similarly, in Haskell, `return` wraps its argument in a monad, while in iTasks, return wraps its argument as the task value of a task [16]. This concept of wrapping and working with values is foundational in iTasks, particularly when transforming or combining tasks, which is the focus of the next subsections.

Transform Combinators

Tasks can be transformed or combined. A simple example of a transformation combinator is `@ :: !(Task a) !(a -> b) -> Task b`, which is similar to Haskells `fmap`: `fmap :: Functor => (a -> b) -> f a -> f b`, as it applies a function to the value inside the task (or functor in Haskell) and returns a new task with the transformed value [15]. Another example of a transform combinator is our function `runSimpleTaskInBrowser`. It takes a task as an argument and modifies it to return its resulting task from execution in the browser along with potential exceptions.

2.2.2 Task Combinators

Assume you have written text and wish to share it with your friend. In addition to the `writeText` you created, you could define a function `shareWithFriend :: String -> Task String`, which shares its argument with your friend after you click a proceed button and holds the argument in its resulting task for other purposes, such as logging. Now, to share the written text, you can use the task combinator `>>?` and the lambda syntax, which looks like \`... -> ...` [32, Section 3.1.4], to define:

```
writeText >>? \text -> shareWithFriend text
```

`>>?` waits for the task value of `writeText` (i.e. the written text) to exist, then waits for the user to click the proceed button. Once the proceed button is pressed, `>>?` passes the written text via the lambda function as `text` to `shareWithFriend` to share the content.

The condition for `>>?` is to wait until there exists a task value and the user presses a proceed button or until the task value is stable. Each task combinator, like `>>?`, has a specific condition related to one or more task values. These conditions enable various decision making possibilities. For example, in the scenario above, you could use `>>-` instead of `>>?`. Unlike `>>?`, `>>-` waits for the task value to be a stable, finalized value before executing the next task. Consequently, it does not display a proceed button for sharing text.

Step and Parallel Combinators

Both `>>-` and `>>?` are examples of step combinators. Step combinators are one of two kinds of task combinators used to structure tasks in iTasks. The two types of task combinators are [25, p. 1–2]:

- **Step combinators**, which allows sequential execution where a new task begins after the previous one has progressed and the task value of the previous task satisfies a specified condition.
- **Parallel combinators**, which allows tasks to run concurrently by interleaving them: switching between tasks to create the idea of parallelism. How they are switched depends on the observed task values and the specified condition [29, p. 204].

An example of a parallel combinator browser is `-&&-`, which groups two tasks in parallel and completes both. Resuming with the example from the top of this subsection (Section 2.2.2): If you want not only to share your text with a friend when you are done writing it but also with your mom, you can define a function `shareWithMom` and use `-&&-` to define:

```
writeText >>- \text → (shareWithFriend text -&&- shareWithMom text)
```

which shares the text with your friend and mom.

2.2.3 Shared Data Sources

Step combinators enable communication by passing task values to subsequent tasks. In contrast, parallel combinators and interactive tasks communicate using shared data sources (SDS). While tasks can operate on isolated values, SDSs allow multiple tasks to work with the same values. These sources are assignable references that instantly propagate changes to all dependent tasks, including interactive ones. This mechanism enables the creation of networks of interactive tasks that collaboratively act on a SDS [29].

Interaction Primitives

Imagine you want to write some message on a website your friend created with iTasks. While tasks can access each other's values, such as `writeText` in Section 2.2.1, a user like you cannot enter or view text on another user's website without one or more user interaction functions. `enterInformation` is such a user interaction function, or better called: an interaction primitive. Your friend could write a task similar to the following to allow users to write messages:

```
enterText :: Task String
enterText = enterInformation []
```

`enterInformation` keeps the value entered by a user as a task value. Other interaction primitives serve other interaction goals, such as updating or viewing information. Despite their different interaction goals, they all use an `interact` function to interact with the browser and a SDS containing the task value being entered/updated/viewed to notify tasks dependent on it [9, p. 3]:

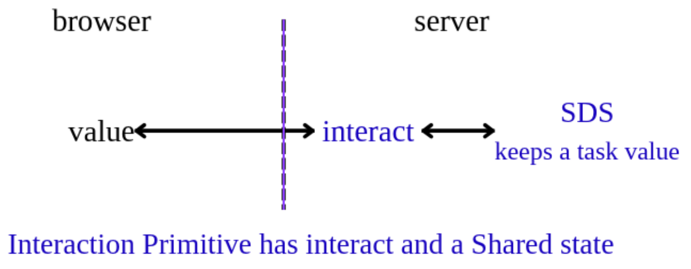


Figure 2.1: Interaction primitive.

The browser displays the value of Figure 2.1. The UI of the value is built using interactive tasks called editors.

2.2.4 Editors

In the last section, `enterInformation` of `enterText` takes an empty list as an argument. This list does not have to be empty as it may contain an editor. We explain editors by resuming the example of last section:

Your friend may have created a text field on their website for you to indicate where to type your message. A text field is a UI component. In iTasks, UI components on the browser, like a text field, are created and interacted with through special tasks called editors. The editor for a text field is called `textField`. The version of `enterText` which also shows the text field is:

```
enterText :: Task String
enterText = enterInformation [EnterUsing (\x→x) textField]
```

The `textField` or any other editor is a task that directly interacts with its end-users [25, p. 1]. iTasks has basic editors, like `textField`, and more advanced editors, such as `pikadayDateField`, which allows users to pick and display a date from a calender.

Editor's key components

Every editor is of the type `Editor r w` in which:

- `r` stands for the type of the data the editor reads (when displaying or updating the data).
- `w` stands for the type of the data the editor writes back after an event (when entering or updating the data).

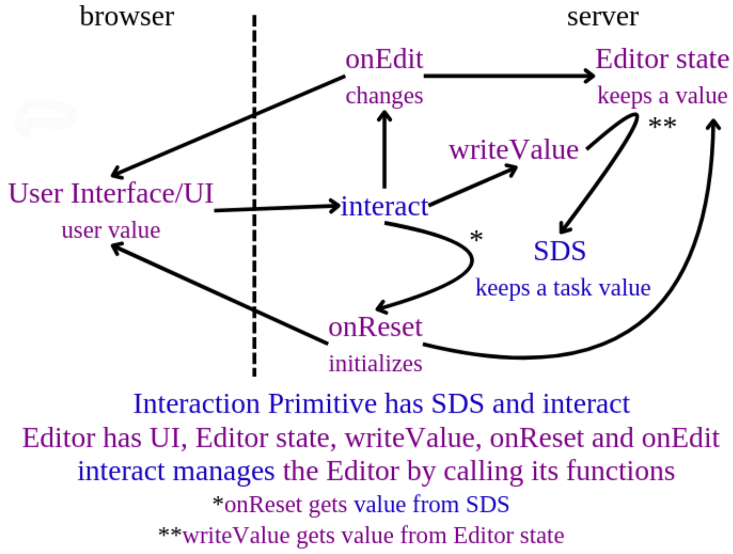
In many editors, the types `r` and `w` are identical. Editors in interaction primitives for updating and entering values require the write type `EditorReport a`. This write type differentiates between empty, invalid, and valid editor states, supporting the communication between the possibly inconsistent or unparsable value of the editor state and the task value of the SDS. For the editor `textField :: Editor String (EditorReport String)`, the editor state is always a valid string.

Every editor implements four functions, of which the following three are important for our research [21]:

- `onReset` initializes the editor's UI and the task value from the SDS of the interaction primitive. The task value is stored in the state of the editor on the server, the editor state. The JavaScript implementations for generating the UI components are sent to the browser before the iTasks program executes. `onReset` uses these implementations to initialize the UI [33, p. 7].
- `onEdit` handles edit events, determining if and how to update the editor state and UI accordingly. An edit event is, for example, a user typing into a text field or selecting a date.
- `writeValue` computes the write value from the value kept in the editor state. While the user is editing in the UI, the value in the editor state may be inconsistent or unparsable. Once the value becomes consistent and parsable, `writeValue` updates the SDS, changing the task value to the computed write value. [29, p. 201–202].

The editor functions above are called by the `interact` function of an interaction primitive to manage the editor.

To visualize the interactions between the interaction primitive and an editor, we show you a high level overview of these interactions in the figure below:



browser
server
onEdit
changes
Editor state
keeps a value
**
writeValue
User Interface/UI
user value
interact
*
SDS
keeps a task value
onReset
initializes
Interaction Primitive has SDS and interact
Editor has UI, Editor state, writeValue, onReset and onEdit
interact manages the Editor by calling its functions
*onReset gets value from SDS
**writeValue gets value from Editor state

Figure 2.2: Interaction primitive & editor

The interaction primitive in `enterInformation`, used in `enterText`, includes the identity function `\x -> x`. This function ensures that the `String` value entered in the editor is directly used as the task value. It is passed as the `vw -> a` argument in the `EnterUsing` constructor within `enterInformation`, which you can see in their abstracted definitions:

```
enterInformation :: [EnterOption m] → Task m
:: EnterOption a = EnterUsing (vw → a) (Editor vr (EditorReport vw))
```

Here, `vw` represents the editor's write type, while `a` represents the task value stored in the SDS. By using `\x -> x` in `enterText`, we explicitly state that the editor's value is the same as the task value. For other interaction primitives, such as `updateInformation`, other ways for value reading and writing must be specified. Below are the abstracted definitions related to `updateInformation` to compare with `enterInformation`:

```
updateInformation :: [UpdateOption m] m → Task m
:: UpdateOption a = UpdateUsing (a → vr) (a → vw → a)
    (Editor vr (EditorReport vw))
```

Unlike `enterInformation`, where specifying a function `vw -> a` is enough, `updateInformation` needs functions of type `a -> vr` to update the editor's `r` type from the task value in the server and `a -> vw -> a` to update the task value from the interaction in the browser.

doEditEvent

Task values and the editor's UI can both be updated. The latter is achieved using `doEditEvent`. The JavaScript function `doEditEvent` sends edit events from the browser to the server, triggering the `onEdit` function, to possibly change the UI. Without these edit events from the UI to the server, represented by the arrow from UI to `interact` in Figure 2.2, the UI would be static; typing or other editing events in the browser would not modify it. Calling `doEditEvent` is currently the only way to send any data from the browser and potentially update the editor's state on the server. When an iTasks program runs in the browser, the implementation of `doEditEvent` is included with the UI components [33, p. 7].

Additionaly, the UI attributes allow the reverse of `doEditEvent` by sending data from the server to the browser during execution. They further define the behavior and looks of the chosen UI components [21].

Chapter 3

Strictness & Uniqueness

Strictness and uniqueness are features in Clean and, by extension, iTasks. While this research emphasizes TOP and iTasks, these features surface throughout the study, particularly in Sections 5.3.3, 5.4.1 and 5.4.2 of Chapter 5, where they are crucial for decision making in our solution–a function, or more specifically, a transform combinator, that runs a task in the browser. Therefore, we dedicate this chapter to explain them.

3.1 Strictness

Strictness is an evaluation strategy that defines when expressions, such as arguments, are evaluated. In strict evaluation, arguments are evaluated *before* being passed to a function and are called strict arguments. In contrast, lazy expressions are only evaluated when needed *after* being passed to a function. Strictness enforces immediate evaluation before an expression is passed to a function, while laziness delays it [6, p. 9]. In the functional programming language Clean, lazy evaluation is the default strategy [28, p. 23]. However, lazy evaluation can increase space usage due to many unevaluated expressions of small data structures. Strictness reduces this space usage by evaluating such expressions before applying the function [35], [32, Section 10.1.1].

3.1.1 Strictness implementation

Even though Clean is a lazy language, strictness can be enforced for specific arguments or fields in a data type by placing an exclamation mark before the type definitions of these arguments and fields, such as `!String` [1, p. 244]. The strict expressions in Clean are not fully evaluated; instead, they are evaluated to a form known as weak root or head normal form.

3.1.2 Weak Head/Root normal form

A Clean program can be represented as an execution graph. The nodes of the graph represent the expressions in the program, such as functions and arguments. When you, for example, run two imaginary binary functions `f` and `g` of `f 1 (g (2+3) 4)` strictly in Clean, the one node execution graph `f 1 (g (2+3) 4)` becomes a three-node graph:

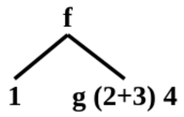


Figure 3.1: weak head normal form graph.

Strict evaluation in Clean rewrites the strict arguments to a form called *weak head normal form*, also known as *weak root normal form*. The rewriting to weak head normal form causes the root node of the strict arguments to be fully evaluated. Apart from the root, the other nodes do not have to be fully evaluated. To illustrate, Figure 3.1, the root node `f` and `1` are fully evaluated, but `g (2+3) 4` is not [32, Section 3.7.5].

3.1.3 Hyperstrict

Strictness does not fully evaluate expressions, as non-root nodes might not be evaluated. Full evaluation, nevertheless, can be useful, as it detects faults, such as stack overflows and illegal instructions, deeper in the evaluation graphs of expressions before execution. Additionally, it reduces the amount of data to be serialized and transmitted, as fully evaluated expressions avoid carrying unevaluated subexpressions. The `hyperstrict` function achieves this by fully evaluating expressions before passing them to other functions. It is particularly useful when passing arguments to the browser, as it ensures that faults in deeper parts of the evaluation graph are detected server-side, preventing them from occurring in the browser. Detecting faults early with hyperstrictness on the server-side eliminates the need to detect them in the browser and communicate them back to the server [33, p. 5].

3.2 Uniqueness

Functional languages have different ways of dealing with the side effects of I/O operations to keep themselves pure. In Haskell, the I/O monad takes care of the I/O operations. The actions are encapsulated in the I/O monad as explicitly sequenced immutable values, such that any side effects can be managed explicitly [12]. Clean's way of keeping I/O operations pure is uniqueness. It can be applied to arguments and data types by prepending them with an asterisk (`*`). An example of a unique argument is the argument of type `*File` in `fwrites`, which is a function that writes to the given unique file. While the I/O function `fwrites` writes a string to the unique file, no other operation can access the file. The file only has one reference to it. If that reference is taken by an operation, like `fwrites`, there is no reference left for other operations to use. Any side effect of `fwrites` is localized and does not leak to the rest of the program [2].

Chapter 4

iTasks program & Browser tools

In earlier chapters, we covered the essential concepts of TOP and iTasks. This chapter explains how an iTasks browser tools and its program work, laying the groundwork for understanding the execution our solution: a transform combinator that runs a task in the browser.

4.1 iTasks browser interpreter: the ABC interpreter

Running parts of an iTasks program in the browser requires several steps. When an iTasks program executes on the server, it is first compiled to an intermediate language called ABC. This language represents the program as a graph of expressions. The ABC code is then translated into machine code and ABC bytecode. To run the program in the browser, the ABC bytecode is sent to the browser. There, an ABC interpreter interprets the bytecode, allowing the program to run in a web environment [20][33].

4.1.1 JavaScript Foreign Function Interface

The UI components in Clean provide much of the desired functionality but lack certain features, such as dragging a file into a message or selecting a date from a calendar. To enhance interactivity, JavaScript functions can be directly invoked from the iTasks program using the JavaScript Foreign Function Interface (JavaScript FFI). The functions provided by the JavaScript FFI form a DSL, designed for performing JavaScript operations within Clean. The ABC interpreter facilitates this process by allowing direct calls to JavaScript functions and passing the results back to the interpreter. This integration ensures that the iTasks program can continue execution in the browser using the returned data [33, p. 7–9].

4.2 Overview

The following figure provides a high-level overview of the execution process of an iTasks program that partially runs in the browser and includes JavaScript FFI functions:

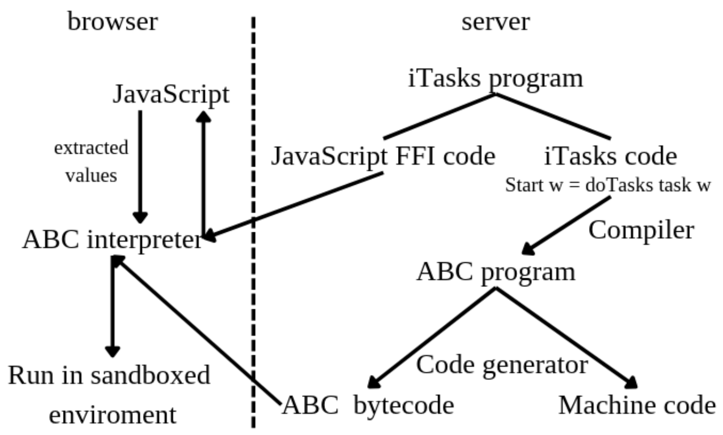


Figure 4.1: iTasks program.

4.3 iTasks program

As in Clean, every iTasks program contains a `Start` function with the type `*World -> *World` [1, p. 195]. The `*` enforces uniqueness. Consequently, that means there is only one `World` value available in an iTasks program (see Section 3.2). The `World` type, used as both argument and result, represents the state of the external environment. It is essential for managing side effects, such as reading input or writing output, in a purely functional way [28, p. 27]. The file system, as an external environment, requires an value of `World`, called `w` in the example program below, whenever a task reads from or writes to a file. By using and returning the `World` value, tasks make changes to the environment explicit, keeping the language pure and reliable [2, p. 87–91].

An iTasks program typically has the following structure:

```
implementation module <name>

// Here usually reside the imported modules containing
// functions you wish to use

// <task> is a placeholder for an (initial) task,
// such as enterInformation []

Start :: *World → *World
Start w = doTasks <task> w
```

`doTasks` executes the specified tasks in the iTasks engine [27, p. 145]. It uses a value of the `World` type to manage any side effects.

4.3.1 How to use the JavaScript FFI

We explain some of the functionality provided by the JavaScript FFI and how to use it to enhance browser interactivity.

JSWorld

In addition to the `World` type, the JavaScript FFI provides the type `JSWorld`. The `World` type maintains a pure interface and represents the server-side, non-web environment in iTasks. A value of `JSWorld` handles impure interactions with JavaScript, such as the `doEditEvent` or `console.log()`, representing the client-side JavaScript environment. They are separate: `World` manages non-web side effects, while `JSWorld` handles web-specific side effects [33, p. 9][9, p. 5].

Example Editor using the JavaScript FFI

The JavaScript FFI provides functions to use JavaScript within Clean and iTasks by extension. You can add these functions to an editor. By adding the functions to an editor, they are executed together with the editor's UI in the browser.

Before we give an example of the implementation of an editor executing a JavaScript function, we have to explain some functions and syntax first:

- `@>>` (or `<<@`, its mirrored counterpart) is the function to tune task UIs, by changing or adding functionality. You can, for example, add some text to show with a task resulting from entering information by defining:

  ```
  enterInformation [] <<@ Label "enter here"
  ```

  Here, `<<@` fine tunes the task returned by `enterInformation` by combining it with a `Label`, a string. Apart from a `Label`, tasks can be fine tuned with JavaScript functionality.

- `# ... = ...` is the let-before syntax of Clean. An expression with this syntax will be executed before any textually subsequent expressions in the same function [32, Section 3.5.4].

We also have to explain functions from the JavaScript FFI:

- `jsGlobal` is a function which represents the global address space in JavaScript.

- `.#` represents a dot in JavaScript to access properties of objects, such as the dot in `console.log()`.
- `.$!` calls a JavaScript function defined before it [9, p. 7].

With the previously provided lists of functions and syntax, we show you the implementation of the `textField` editor that directly calls the JavaScript function `console.log("first")` and `console.log("second")` afterwards in the browser when used with interaction primitives for entering or updating (see Section 2.2.4 on why):

```
textFieldAndLogEditor :: Editor String (EditorReport String)
textFieldAndLogEditor = JavaScriptInit initUI @>> textField
where
    initUI :: !FrontendEngineOptions !JSVal !*JSWorld -> *JSWorld
    initUI _ me jsworld
        # jsworld = (jsGlobal "console" .# "log" .$! "first") jsworld
        # jsworld = (jsGlobal "console" .# "log" .$! "second") jsworld
        = jsworld
```

The `JavaScriptInit` function initializes client-side functionality, setting up the additional JavaScript functions in `initUI`, which in this case logs `first` and `second` to the console. The let-before bindings and passing of `jsworld` define the order of execution: `first` is logged to the console before `second`.

Chapter 5

Running tasks in the browser

In this chapter, we propose and explain our solution for running iTasks in the browser. Currently, when running a part of the iTasks program in the browser, most is handled by the server (see Figures 2.1 and 4.1). The parts of an iTasks program running in the browser, such as the UI and JavaScript functions, are managed by the server and depend on the communication between browser and server. To reduce the communication and server dependency, we created the function `runSimpleTaskInBrowser`. This transform combinator runs a task in the browser and transforms it into a new task, the resulting task.

The task provided to `runSimpleTaskInBrowser` must be simple. A simple task is self-contained and cannot access values from the server environment (represented by `World`) or interact with UI elements it did not generate. The task is evaluated and executed once in the browser. For example, the task:

```
get currentDate >>- \date → return date
```

is invalid because `get currentDate` requires access to the server environment. This is unavailable in the browser due to single-step evaluation and the lack of SDS handling (see Section 2.2.3). These limitations could be addressed in the future by introducing SDS handling in the browser and enabling multiple rewrites with current techniques.

`runSimpleTaskInBrowser`'s implementation can be found in Appendix A.

5.1 runSimpleTaskInBrowser components

The function operates across three layers, shown in Figure 5.1: the server, the browser engine, and the communication between them. We start by clarifying the server functionality, then address the communication layer, and conclude with the browser engine.

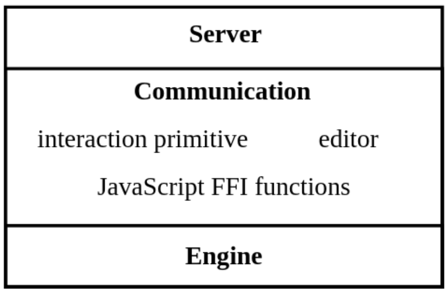


Figure 5.1: runSimpleTaskInBrowser overview.

5.2 Server

The server receives the resulting task value from the communication layer after the browser engine executes the task. Since the communication layer checks for errors and includes them in the resulting task value, the server receives the value wrapped in other types: `TaskValue (MaybeError TaskException (TaskValue a))`. To extract the actual result, `TaskValue a`, we use the transform combinator `transformError` (see Section 2.2.1).

5.2.1 transformError

The `transformError` combinator functions similarly to Haskell's `join`, which has the type `Monad m => m (m a) -> m a` and ”flattens” nested monadic structures. In addition to flattening, `transformError` checks for errors and, if any are present, injects exceptions to notify other tasks or the user. `transformError` retrieves the inner `TaskValue` from the result of type `TaskValue (MaybeError TaskException (TaskValue a))`, injects exceptions when `MaybeError` is of type `Error`, and transforms the result into a resulting task of type `Task` with the inner task value, the value of type `a`.

5.3 Communication

The communication layer is constructed from the interaction primitive `enterInformation` and a custom editor we developed, called `ribEditor`, which we elaborate on in this section. The `initUI` is added to the `ribEditor` (see Section 4.3.1). It combines JavaScript and Clean functionality for communicating the given task and the resulting task value.

5.3.1 enterInformation

We need an interaction primitive to communicate the given task with the browser. We did not wish to reinvent the wheel, implementing our own SDS with proper sharing of task values, and therefore chose to use `enterInformation`. Another valid option was `updateInformation`. However, `updateInformation` uses `UpdateUsing` instead of `EnterUsing`, which requires specifying more arguments than necessary (see Section 2.2.4).

5.3.2 ribEditor

Below is part of a simplified implementation of the `ribEditor` passed to `enterInformation`:

```
:: ReadWriteType a  :== MaybeError TaskException (TaskValue a)
:: EditType a       :== MaybeError TaskException (TaskValue a)
:: StateType a      :== MaybeError TaskException (TaskValue a)

ribEditor =
        JavaScriptInit initUI @>> baseEditor
    where
        baseEditor :: Editor (EditType a) (StateType a)
        (ReadWriteType a) (ReadWriteType a)
        baseEditor =
            { Editor
            | onReset = onReset
            , onEdit = onEdit
            , writeValue = writeValue
            }
```

The `ribEditor` is localized in `runSimpleTaskInBrowser` and receives the given task to run in the browser as a lifted argument. It runs the task and returns its possibly modified task value, the result. This process relies on the implementation of the `onReset` function, initalizing the UI of an editor, and of the `initUI` function added to the editor to integrate JavaScript for retrieving the given task, executing it and inspecting the result for exceptions or UI changes that should be done. The `onReset` function transfers the task to the browser and initializes a loading icon with `UILoader` to inform users that the task is being executed. It adds the task as an attribute to the initialized UI, allowing `initUI` to retrieve the task from the attribute (see Sections 2.2.4 and 4.3.1). We explore the `onReset` and `initUI` functions in greater detail in the next sections.

5.3.3 onReset

Transferring a task involves sending its ABC bytecode and execution graph. The ABC bytecode represents the machine code of parts of the ABC program compiled from the iTasks program (see Figure 4.1). The ABC bytecode is sent to the browser before parts of it are executed with execution graphs managed by the interpreter. The execution graphs specify which ABC bytecode to evaluate in the browser (see Section 3.1.2). The function `jsSerializeGraph` retrieves the task's execution graph and serializes it into a `String` [33]. The serialized execution graph of the given task is then converted into a `JSONString`, a JavaScript data type, and put into a UI attribute to be retrieved by `initUI`. The UI attribute ensures that the graph is sent from the server to the browser before it is retrieved to execute the given task.

Keeping laziness

We do not call `jsSerializeGraph` on the given task directly, but wrap the task in a type we call `Box`:

```
:: Box t = Box t
```

This wrapper type makes sure the task is not evaluated on the server (see Appendix C). The weak head normal form evaluation ensures that only the root, `Box`, is evaluated on the server, not the given task.

5.3.4 initUI: Communicating the result

After the task is executed, its result is inspected in `initUI` in the browser for exceptions and changes of the UI. The UI changes are done in the browser with the JavaScript FFI functions. The exceptions check results in `enterInformation` returning a task value of type `TaskValue (MaybeError TaskException (TaskValue a))`. This task value is sent to the server using `doEditEvent`, which triggers an edit event, notifying the server that the given task has been executed (see Section 2.2.4). The server further handles the nested task value in `transformError` (see Section 5.2).

5.3.5 Overview ribEditor

We summarize the `ribEditor` in the following figure:

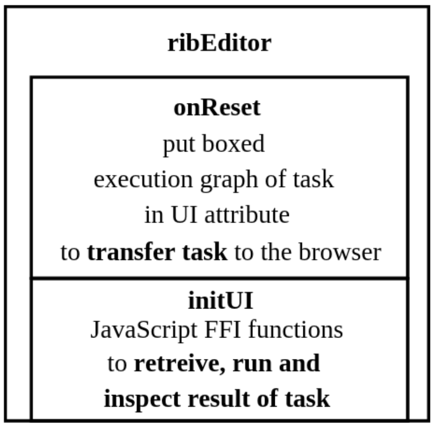


Figure 5.2: ribEditor key components.

5.4 Engine

The browser engine uses the function `apTask`. The task is executed, and its result inspected, using the `calculate` function within `initUI`, ensuring it runs in the browser.

To allow UI components from other parts of a program to initialize before executing the task, `calculate` is wrapped using the JavaScript FFI function `jsWrapFun`. This wrapped function is then passed to the JavaScript function `setTimeout`, which schedules `calculate` to run in the browser after UI components from the main program are initialized.

5.4.1 Core engine: apTask

The iTasks function used in `calculate` to execute a task is `apTask`. `apTask` runs the task but requires a value of `IWorld`, which depends on the `World` value.

In the browser, the `IWorld` fields, including a value of `World`, are not actively used but may become necessary for evaluating multiple browser tasks after each other in the future. Therefore, most fields in the `IWorld` record are filled with empty values. Still, in an iTasks program, the `World` value is unique and cannot be an empty value.

5.4.2 Another value of `World`

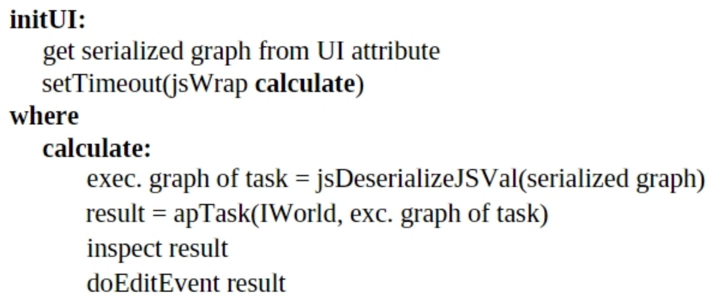
The `World` value is unique. Assigning it to a field in the `IWorld` record of `runSimpleTaskInBrowser` and running it would cause uniqueness errors, as this would create multiple references to the unique `World` value (see Section 3.2). To avoid this, we create and assign a new `World` value to the `World` field in the `IWorld` value. This new `World` value should not be outside of the browser to keep iTasks a deterministic and pure language. We avoid creating multiple `IWorld` and, by extension, `World` values by checking if they have already been initialized and stored in the browser. If an `IWorld` value exists, `runSimpleTaskInBrowser` reuses it.

5.4.3 Retrieving the task for execution

`apTask` needs the `IWorld` value, which is initialized in the browser, and the execution graph of the task to run it in the browser. The execution graph of the given task is taken from the UI attribute, unboxed, and deserialized in `calculate` with `jsDeserializeJSVal`. `jsDeserializeJSVal` deserializes the encoded string to the execution graph, such that `apTask` can run the corresponding task in the browser.

5.4.4 Pseudocode of initUI and calculate

Below is a figure showing the pseudocode of the key functionality in `initUI` and `calculate` explained in this chapter:



```
initUI:
    get serialized graph from UI attribute
    setTimeout(jsWrap calculate)
where
    calculate:
        exec. graph of task = jsDeserializeJSVal(serialized graph)
        result = apTask(IWorld, exc. graph of task)
        inspect result
        doEditEvent result
```

Figure 5.3: Pseudocode of initUI and calculate.

5.4.5 Avoiding recalculating

The functions `calculate` and `apTask` are used within `initUI`, which is called each time the UI is replaced or initialized. To avoid recalculating the task during UI replacements, we delete the execution graph from the UI attribute after initialization and check for its existence before using it.

5.4.6 IWorld and Future work

The `IWorld` value is essential for evaluating tasks independently in the browser while allowing task values to be shared for collaboration. Currently, `runSimpleTaskInBrowser` does not support sharing task values between tasks of the same program evaluated in the browser at different times. Although the server uses SDSs and a value of the `IWorld` to share task values and notify other tasks of changes, these SDSs can not share data in the browser. To enable independent task evaluation and value sharing, the `IWorld` value must persist in the browser, and a mechanism for sharing task values in the browser must be developed.

We store the `IWorld` value in the browser's `sessionStorage`. Alternatively, we could store the `IWorld` value as an attribute of the root UI component to retain it across multiple task evaluations. Both approaches are tab-specific and valid. However, because the iTasks framework is central to this research, and its functionality or structure may evolve, the root UI component could undergo significant changes over time. By contrast, `sessionStorage` provides a stable and framework-independent option, making it the more reliable choice for now.

Chapter 6

Related Work

Our solution runs a task—a concept from TOP implemented in iTasks—in the browser using an editor. This chapter introduces related work, including the implementations of TOP, the evolution of editors, and a programming paradigm similar to TOP.

Our solution also depends on iTasks' construction, which uses uniqueness to manage I/O and browser tools like the ABC interpreter. We describe how iTasks adopted the ABC interpreter and present alternative solutions used by functional programming languages for running code in the browser. Finally, since iTasks combines an interpreter and a compiler, we provide two examples of similar approaches.

6.1 Task-Oriented

TOP is known for being beneficial in developing multi-user workflow systems [34]. This it is not surprising, as there are task-oriented solutions, such as TOP, used for workflows [14][17][4] and collaboration [13][30].

6.2 FRP

Functional Reactive Programming (FRP) is a programming paradigm with some similarities to TOP. FRP models dynamic value changes declaratively by defining networks of interdependent values, known as behaviors, which evolve over time in response to events. When a behavior updates, all dependent behaviors automatically adjust. Implementations of FRP often connect input devices, such as a mouse or keyboard, to event streams and map behaviors to output UI elements like text fields. This approach allows for declaratively programming UIs, which is also the case when using editors of TOP. However, the two paradigms achieve this in different ways.

TOP automatically derives UI components by sending their implementations to the browser, whereas FRP requires manually creating the UI components. A key distinction lies in their goals and how they handle updates: FRP focuses on automatically updating data dependencies, while TOP expresses collaboration patterns. Additionally, an event in TOP only causes updates up to the next step of the task, whereas an event in FRP propagates through the entire network.

Despite these differences, there are notable similarities. Both paradigms enable declaratively programming UIs. Furthermore, the ”stepper” behavior in FRP, which processes events and retains the most recent event’s value, is conceptually similar to TOP editors. Editors in TOP hold a value in the editor state and process edit events. These shared traits highlight their overlapping capabilities, even as they pursue distinct goals [34, p. 12].

FRP stems from the paper *Functional Reactive Animation* [11] and has many variants and implementations, such as Elm [7], Flapjax [24] and Lively RaTT [5].

6.3 TOP

So far, we found two implementations of TOP defining tasks, task combinators and SDSs: iTasks and mTasks.

6.3.1 mTasks

mTasks is similar to iTasks, a TOP implementation embedded in Clean. Unlike iTasks, it focuses on microcontroller units (MCUs). MCUs have resource-constraints causing issues with rich OS features and threading. mTasks deals with these issues and integrates with iTasks for the use of web-interfaces for remote control and monitoring [23, p. 1587, 1591].

6.3.2 iTasks

Our investigation focused on running iTasks in the browser while preserving and transferring the TOP concepts for lazy execution, rather than optimizing the performance of iTasks in the browser. As part of this work, we developed an editor. The editor’s structure evolved over time, starting with tasklets, then editlets, and finally editors to build the UI.

Tasklets

Tasklets are client-side components designed to handle specific, discrete parts of a task from iTask. They allow for partial evaluation and execution of tasks in the browser, reducing the server’s workload and improving interactivity [10]. Editlets succeeded tasklets, as tasklets have the following limitations [9, p. 4]:

1. Tasklets exchange all the underlying data (UI state, task state and additional data) between the client and server during edits, resulting in significant communication costs.

2. Tasklets cannot work with shared data. When you want an interactive map in which multiple users making concurrent modifications, this is not possible with tasklets.

3. Tasklets do not integrate with the generic, type-driven UI generation of iTasks, meaning:

   - UIs are generated automatically based on the structure of types.
   - UIs can adapt to changes in types.
   - Developers need to write custom UIs for every variation or combination of types.

   This contrasts with tasklets, which are tightly coupled to specific tasks and do not support automatic, compositional UI generation.

Editlets

The editlets are components that encapsulate the client-side state, user interactions, and incremental communication with the server (only communicating changes). They overcome the three limitations of tasklets (listed above) by [9]:

1. exchanging the updates instead of the whole shared value, which reduces possible communication overhead

2. working with shared data by handling the conflicting updates

3. integrating with the generic, type-driven UI generation of iTasks, making it easier to customize, extend, reuse and compose UIs.

Editors

The editors are based on the editlets, but do not use the intermediate language SAPL (Simple Application Programming Language) nor a SAPL-to-JavaScript compiler to run parts of the program in the browser. SAPL and the SAPL-to-JavaScript compiler were replaced by the ABC interpreter as it tackled the following problems: First, due to the way Clean was compiled to SAPL, not all Clean values could be sent to the browser, as they had to be evaluated to a normal form first. Second, because SAPL branched off early in the compiler toolchain, it was time-consuming to maintain. Third, the compiled JavaScript had a bad worst-case performance [33].

The ABC interpreter is a solution to these three problems. With the ABC interpreter, there is no need to evaluate values to normal form to send to the browser, as the values can be directly copied from server to client. The ABC language operates on a lower level (ABC bytecode) than SAPL, simplifying the compiler toolchain. The ABC interpreter also improves the performance predictability and removes the need for developers to write separate optimized versions of functions. Due to the advantages of the ABC interpreter in comparison to SAPL, the editors improved by replacing the need for SAPL with the ABC interpreter.

6.4 Functional Programming in the browser

This section explores how functional programming languages compile to JavaScript, handle I/O, and use compilers and interpreters. These approaches provide context for understanding iTasks' current position.

6.4.1 Compilation to JavaScript

Before the ABC interpreter, the Clean compiler could generate code in SAPL, a pure and lazy language. SAPL contains the essential minimum of a Clean program, while preserving Clean's semantics. Parts of the SAPL code would be compiled to JavaScript with the SAPL-to-JavaScript compiler during runtime to run the JavaScript in the browser [18][8].

ClojureScript is a language from the functional language Clojure to bring Clojures expressiveness, performance and host interoperatibily to the front-end. ClosureScript does not use an intermediate language like SAPL, but translates the ClosureScript semantics into JavaScript constructs before runtime. Standard web communication, such as HTTP APIs and WebSocket protocols can then be used for the communication between server and client [31].

Elm is a functional language implementing FRP, a programming paradigm with similarities to TOP (see Section 6.2). The Elm-to-JavaScript compiler translates Elm programs to HTML for layout, CSS for styling, and JavaScript for reactive behavior, managing events, signals, and state updates [7].

Racket, a functional language typically used for eduction, has apart from the JavaScript compiler Whalesong, a JavaScript FFI for adding new reactive features and a framework called the World programming model. Unlike iTasks where the world value represent the external environment, Racket's framework keeps track of a world value that handles events. [36].

6.4.2 Handling I/O

Handling I/O in functional programming involves several approaches. In iTasks, the unique world manages side effects and ensures the correct execution order of I/O operations. In Haskell, the I/O Monad encapsulates I/O operations to manage side effects and execution sequencing [19]. Elm uses commands and subscriptions to separate side effects. Commands specify operations for the Elm runtime to execute, while subscriptions define external events the program listens to. These abstractions isolate side effects from the program's logic and prevent direct execution. The Elm runtime manages execution in a strict order, maintaining the language's functional purity [22].

6.5 Interpreters & Compilers

Compilers execute code before runtime, while interpreters evaluate code during runtime. Interpreters are in general easier to implement and more flexible, but slower because they may repeatedly process certain code, such as loops [3]. The choice between using a compiler, an interpreter, or both depends on the language's design and purpose.

In iTasks, both a compiler and an interpreter are used to execute programs. Similarly, the functional language Erlang supports both compiled and interpreted code, which share the same heap and stack. This setup enables just-in-time (JIT) compilation, reducing context-switching overhead and improving execution speed [26].

The functional language weScheme also uses a compiler and a (JavaScript) interpreter. However, unlike iTasks and Erlang, compiled and interpreted code in weScheme do not share the same heap or stack, maintaining separate execution contexts [37].

Chapter 7

Conclusions

We addressed the server dependency and communication overhead in iTasks by developing the function `runSimpleTaskInBrowser`. This function evaluates and executes a task in the browser, updates the UI with any resulting changes, inspects exceptions during execution and sends them to the server, and returns the resulting task.

During the implementation, we introduced the wrapper type `Box` to ensure laziness and created a new `World` value to execute tasks in the browser, as the existing `World` value is unique.

The task provided to `runSimpleTaskInBrowser` is evaluated in a single step within the browser and cannot depend on server-side values or interact with UI elements it did not generate.

In the future, introducing a SDS mechanism in the browser could allow multiple independently evaluated tasks to share values and coordinate execution. Combined with a value of `IWorld` for task coordination, this would enable more complex, interdependent tasks to run simultaneously in the browser.

Acknowledgement

I sincerely thank my supervisor, Mart Lubbers, for his guidance and support throughout this thesis. His willingness to explain concepts, collaborate on challenges, and provide thoughtful feedback greatly enhanced my study. I am also deeply grateful to my friends and my supervisor for their encouragement and belief in me, which have been a source of motivation throughout this journey.

Bibliography

[1] Achten, P., Koopman, P., and Plasmeijer, R. An introduction to task-oriented programming. In *Central European Functional Programming School (CEFP 2013), Revised Selected Papers*, vol. 8606 of *Lecture Notes in Computer Science.* Springer, 2015, pp. 187–245.

[2] Achten, P., and Plasmeijer, R. The ins and outs of clean i/o. *Journal of Functional Programming 5*, 1 (1995), 81–110.

[3] Aggarwal, A., Singh, D., and Jain, S. A hybrid approach of compiler and interpreter. *International Journal of Scientific & Engineering Research 5*, 6 (2014), 4.

[4] Andreas, J., Bufe, J., Burkett, D., Chen, C., Clausman, J., Crawford, J., Crim, K., DeLoach, J., Dorner, L., Eisner, J., Fang, H., Guo, A., Hall, D., Hayes, K., Hill, K., Ho, D., Iwaszuk, W., Jha, S., Klein, D., Krishnamurthy, J., Lanman, T., Liang, P., Lin, C. H., Lintsbakh, I., McGovern, A., Nisnevich, A., Pauls, A., Petters, D., Read, B., Roth, D., Roy, S., Rusak, J., Short, B., Slomin, D., Snyder, B., Striplin, S., Su, Y., Tellman, Z., Thomson, S., Vorobev, A., Witoszko, I., Wolfe, J., Wray, A., Zhang, Y., and Zotov, A. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics 8* (2020), 556–571.

[5] Bahr, P., Graulund, C. U., and Møgelberg, R. E. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang. 5*, POPL (2021).

[6] Barzilay, E., and Clements, J. Laziness without all the hard work: combining lazy and strict languages for teaching. In *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education* (New York, NY, USA, 2005), FDPE '05, Association for Computing Machinery, p. 9–13.

[7] Czaplicki, E., and Chong, S. Asynchronous functional reactive programming for guis. *SIGPLAN Not. 48*, 6 (2013), 411–422.

[8] Domoszlai, L., Bruël, E., and Jansen, J. M. Implementing a non-strict purely functional language in javascript. *Acta Universitatis Sapientiae 3*, 1 (2011), 76–98.

[9] Domoszlai, L., Lijnse, B., and Plasmeijer, R. Editlets: type-based, client-side editors for itasks. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2014), IFL '14, Association for Computing Machinery.

[10] Domoszlai, L., and Plasmeijer, R. *Tasklets: Client-Side Evaluation for iTask3.* Springer International Publishing, Cham, 2015, pp. 428–445.

[11] Elliott, C., and Hudak, P. Functional reactive animation. *SIGPLAN Not. 32*, 8 (1997), 263–273.

[12] Gordon, A. D., and Hammond, K. Monadic i/o in haskell 1.3. In *Proceedings of the haskell Workshop* (1995), Citeseer, pp. 50–69.

[13] He, G., Cui, S., Dai, Y., and Jiang, T. Learning task-oriented channel allocation for multi-agent communication. *IEEE Transactions on Vehicular Technology 71*, 11 (2022), 12016–12029.

[14] Huang, T.-W., Lin, D.-L., Lin, Y., and Lin, C.-X. Taskflow: A general-purpose parallel and heterogeneous task programming system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41*, 5 (2022), 1448–1452.

[15] Hudak, P., and Fasel, J. H. A gentle introduction to haskell. *SIGPLAN Not. 27*, 5 (May 1992), 1–52.

[16] HUTTON, G., and MEIJER, E. Monadic parsing in haskell. *Journal of Functional Programming 8*, 4 (1998), 437–444.

[17] Injun Choi, C. P., and Lee, C. A transactional workflow model for engineering/manufacturing processes. *International Journal of Computer Integrated Manufacturing 15*, 2 (2002), 178–192.

[18] Jansen, J. M., Koopman, P., and Plasmeijer, R. Efficient interpretation by transforming data types and patterns to functions. In *Trends in Functional Programming Volume 7.* Intellect, 2005, pp. 73–90.

[19] Klinger, S. *The Haskell Programmer's Guide to the IO Monad: Don't Panic.* No. 05-54 in CTIT Technical Report Series. Centre for Telematics and Information Technology (CTIT), Netherlands, 2005. Imported from CTIT and EWI/DB PMS [db-utwente:tech:0000003696].

[20] Koopman, P. W. *Functional programs as executable specifications.* [Sl: sn], 1990.

[21] Lijnse, B., and Plasmeijer, R. Typed directional composable editors in itasks. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2021), IFL '20, Association for Computing Machinery, p. 115–126.

[22] Loder, W. *Web Applications with Elm.* Apress Berkeley, CA, 2018.

[23] Lubbers, M., Koopman, P., and Plasmeijer, R. Multitasking on microcontrollers using task oriented programming. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2019), IEEE, pp. 1587–1592.

[24] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. Flapjax: a programming language for ajax applications. *SIGPLAN Not. 44*, 10 (2009), 1–20.

[25] Oortgiese, A., van Groningen, J., Achten, P., and Plasmeijer, R. A distributed dynamic architecture for task oriented programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '17, Association for Computing Machinery, pp. 1–12.

[26] Pettersson, M., Sagonas, K., and Johansson, E. The hipe/x86 erlang compiler: System description and performance evaluation. In *Functional and Logic Programming* (Berlin, Heidelberg, 2002), Z. Hu and M. Rodríguez-Artalejo, Eds., Springer Berlin Heidelberg, pp. 228–244.

[27] Plasmeijer, R., Achten, P., and Koopman, P. itasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2007), ICFP '07, Association for Computing Machinery, p. 141–152.

[28] Plasmeijer, R., and Eekelen, M. v. Keep it clean: a unique approach to functional programming. *SIGPLAN Not. 34*, 6 (1999), 23–31.

[29] Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., and Koopman, P. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2012), PPDP '12, Association for Computing Machinery, p. 195–206.

[30] Shao, J., Mao, Y., and Zhang, J. Task-oriented communication for multidevice cooperative edge inference. *IEEE Transactions on Wireless Communications 22*, 1 (2023), 73–87.

[31] Sierra, S., and VanderHart, L. *ClojureScript: Up and Running.* " O'Reilly Media, Inc.", 2013.

[32] Staps, C., Lubbers, M., Van der Veen, E., Dercksen, K., Michels, S., and Alberts, G. Clean language report version 3.0. `https://cloogle.org/doc/`. Accessed: 2024-11-12.

[33] Staps, C., van Groningen, J., and Plasmeijer, R. Lazy interworking of compiled and interpreted code for sandboxing and distributed systems. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (2021), IFL '19, Association for Computing Machinery, pp. 1–12.

[34] Steenvoorden, T., Naus, N., and Klinik, M. Tophat: A formal foundation for task-oriented programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2019), PPDP '19, Association for Computing Machinery.

[35] Wadler, P. Lazy versus strict. *ACM Computing Surveys 28*, 2 (1996), 318–320.

[36] Yoo, D., and Krishnamurthi, S. Whalesong: running racket in the browser. *SIGPLAN Not. 49*, 2 (2013), 97–108.

[37] Yoo, D., Schanzer, E., Krishnamurthi, S., and Fisler, K. Wescheme: the browser is your programming environment. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2011), ITiCSE '11, Association for Computing Machinery, p. 163–167.

Appendix A

runSimpleTaskInBrowser implementation

```
module RunInBrowser

import iTasks
import iTasks.UI.JavaScript
import StdEnv
import Data.Error.GenJSON
import qualified Data.Map as DM
import iTasks.Internal.TaskEval
import iTasks.Internal.IWorld
from AsyncIO import :: AIOState{..},
     :: OutputData, :: AsyncIOFD, :: Client, :: Listener
import ABC.Interpreter
import System._Finalized
import Math.Random
import ABC.Interpreter.JavaScript
import StdGeneric
import iTasks.UI.Editor.Common

// This wrapper type makes sure lazyness happens (HNF evaluation apparantly)
:: Box t = Box t

:: ReadWriteType a  :== MaybeError TaskException (TaskValue a)
:: EditType a  :== MaybeError TaskException (TaskValue a)
:: StateType a :== MaybeError TaskException (TaskValue a)
:: UnexpectedDestroyedResultException = UnexpectedDestroyedResultException

/**
 * Run a simple task (one that finishes with one evaluation step) in the
 * browser.
 *
 * @param Task to execute in the browser
 * @result Transformed task
 * @throws UnexpectedDestroyedResultException when the task evaluates to a
     DestroyedResult
 */
runSimpleTaskInBrowser :: (Task a) → Task a | iTask a
runSimpleTaskInBrowser task
     =   get applicationOptions
     >>− \options→transformError transformFun
```

```
        (enterInformation [EnterUsing id (ribEditor id options)])
where
    transformFun :: (TaskValue (ReadWriteType a)) → MaybeError TaskException
      (TaskValue a)
    // Task not done yet
    transformFun NoValue = Ok NoValue
    // Ignore stability
    transformFun (Value v _) = case v of
        Ok tv = Ok tv
        Error e = Error e

    // Added an identity function as an argument so that later
    // the deserialised value can be cast
    ribEditor :: ((Task a) → Task a) EngineOptions → Editor (ReadWriteType
     a) (EditorReport (ReadWriteType a)) | iTask a
    ribEditor thisid engineOptions = mapEditorWrite ValidEditor (
        JavaScriptInit (initUI (thisid)) @>> leafEditorToEditor leafRibEditor
     )
    where
        leafRibEditor :: LeafEditor (EditType a) (StateType a) (ReadWriteType
      a) (ReadWriteType a)
        leafRibEditor =
            { LeafEditor
            | onReset = onReset
            , onEdit = onEdit
            , onRefresh = onRefresh
            , writeValue = writeValue
            }

        initUI :: ((Task a) → Task a) !FrontendEngineOptions !JSVal !
            *JSWorld → *JSWorld | JSONEncode{|*|} a
        initUI thisid opts me world
            # (timeMs, world) = (jsGlobal "Date" .# "now" .$? ()) (0, world)
            // Generate a fresh list of random integers
            # randoms = genRandInt timeMs
            //get the iworld from sessionStorage
            # (iworld, world) = (jsGlobal "sessionStorage" .# "getItem"
                .$ "iworld") world
            // Using undefs/aborts in fields of iworld
            // results in an abc-interpreter error? Probably has to do
            // with hyperstrictness of the initUI function···
            // If the IWorld does not exists in the sessionStorage
            // (need to check bc of initUI calls when UI is replaced)
            // initialize iworld for browser and set it in sessionStorage
            | jsIsUndefined iworld = world
                #! (serIworld, world) = jsSerializeOnClient { IWorld
                    | options               = engineOptions
                    , clock                 =
                        { Timespec
                        | tv_sec = timeMs / 1000
                        , tv_nsec = (timeMs rem 1000) * 1000000
                        }
                    , clockDependencies     = [||]
                    , current               =
            { TaskEvalState
            | nextTaskNo = 0
            }
        , currentInstance      = 1
        , random               = randoms
        , symbols              = {#}
        , sdsNotifyRequests    = 'DM'.newMap
        , sdsNotifyReqsByTask  = 'DM'.newMap
        , memoryShares         = 'DM'.newMap
        , readCache            = 'DM'.newMap
        , writeCache           = 'DM'.newMap
        , readSdsSourceVals    = 'DM'.newMap
        , abcInterpreterEnv    =
            { PrelinkedInterpretationEnvironment
            | pie_code_start       =
                { Finalizer
                | finalizer_implementation = DummyFinalizer 0 0 0
                }
            , pie_symbols          = {#}
            , pie_symbols_64       = {#}
            , pie_sorted_symbols   = {#}
            , pie_host_symbols     = {#}
            , pie_symbol_offset    = 0
            }
        , ioStates             = 'DM'.newMap
        , ioHandleMap          = 'DM'.newMap
        , nextIOHandle         = 0
        , lastOnDataCId        = ?None
        , webServiceInstances  = []
        , signalHandlers       = []
        , world                = newWorld
        , aioState             =
            { AIOState
            | listeners    = 'DM'.newMap
            , clients      = 'DM'.newMap
            , aioFd        = { finalizer_implementation =
                  DummyFinalizer 0 0 0 }
            , writeQueues = 'DM'.newMap
            }
        , resources            = []
        , onClient             = True
        , shutdown             = ?None
        } world
    # world = (jsGlobal "sessionStorage" .# "setItem"
        .$! ("iworld", serIworld)) world
// Get the iworld from sessionStorage
// (needed in possible future work
// for multiple evaluation steps in the browser)
# (iworld, world) = jsGlobal "sessionStorage" .# "iworld"
    .? world
// Get the graph from attributes
# (graph, world) = me .# "attributes" .# "graph" .? world
// initUI runs each time a replace UI is done,
// so we need to make sure we don't deserialize graph twice.
```

```
    | jsIsUndefined graph = world
    // Remove the graph string (you can only use it once!).
    # world = jsDelete (me .# "attributes" .# "graph") world
    // Make a callback out of the calculation function
    # (cb, world) = jsWrapFun (calculate me graph iworld) me world
    // Run it asynchronously, this gives other UIs the chance
    // to do something but eventually it will block the client···
    # world = (jsGlobal "setTimeout" .$! (cb, 0)) world
    = world
where
    calculate :: !JSVal !JSVal !JSVal !{!JSVal} !*JSWorld → *JSWorld
    calculate me graph iworld args world
        // Deserialize and immediately unbox
        # (Box jsres, world) = jsDeserializeJSVal graph world
        // Deserialize iworld
        # (desIworld, world) = jsDeserializeJSVal iworld world
        // Run the task
        // (cast to correct type using identity function)
        # (taskResult, _) = apTask (thisid jsres) ResetEvent
            // Invent a TaskEvalOpts record
            { TaskEvalOpts
            | noUI              = False
            // Instance 0 is reserved for system things and sometimes
            // treated differently by the engine,
            // so we use instance 1
            , taskId            = TaskId 1 0
            , lastEval          = 0
            , sessionInstance = ?None
            , attachmentChain = []
            } desIworld
        // Inspect the task result
        # (editVal, world) = case taskResult of
            ValueResult tv _ uiChange _
                # world = (me .# "onUIChange" .$!
                    (toJS (encodeUIChange uiChange))) world
                = (Ok tv, world)
            ExceptionResult exc
                = (Error exc, world)
            //Tasks only return DestroyedResults when they are sent a
            // DestroyEvent, so this shouldn't happen.
            DestroyedResult
                = (Error
                ( dynamic UnexpectedDestroyedResultException
                , "Unexpected DestroyedResult when evaluating task"
                ), world)
        # world = (me .# "doEditEvent" .$! toJSON editVal) world
        = world

    // World made to use in browser
    // Should not be used elsewhere
    newWorld :: *World
    newWorld
        = code inline {
            fillI 65536 0
        }

onReset :: !UIAttributes !(?(ReadWriteType a)) !*VSt →
    *(!*MaybeErrorString (!UI, !(StateType a),
    !?(ReadWriteType a)), !*VSt)
// Already calculated
onReset attrs (?Just oldres) vst
     = (Ok (doneUI, oldres, ?None), vst)
onReset attrs ?None vst=:{VSt|abcInterpreterEnv}
    // Explicitly serialize to keep lazyness
    #! graph = jsSerializeGraph (Box task) abcInterpreterEnv
    // Add the serialized graph to the attributes
    # attrs = 'DM'.put "graph" (JSONString graph) attrs
    // Using UILoader to show evaluation in browser is happening
    = (Ok (uia UILoader attrs, Ok NoValue, ?None), vst)

onEdit :: !(EditType a) !(StateType a) !*VSt → *(!*MaybeErrorString
    (!UIChange, !(StateType a), !?(ReadWriteType a)), !*VSt)
// An edit event always means the calculation is done
onEdit res _ vst = (Ok (NoChange, res, ?Just res), vst)

doneUI :: UI
doneUI = UI UILabel (textAttr "done") []

onRefresh :: !(?(ReadWriteType a)) !(StateType a) !*VSt →
    *(!*MaybeErrorString (UIChange, !(StateType a),
    !?(ReadWriteType a)), !*VSt)
// Refresh we can ignore, this concerns SDSs
onRefresh _ st vst = (Ok (NoChange, st, ?None), vst)

writeValue :: (StateType a) → MaybeErrorString (ReadWriteType a)
writeValue st = Ok st
```

Appendix B

In this appendix we test how `runSimpleTaskInBrowser` handles when the result of a given task contains a UI change or an exception.

B.1 UI Change

We test if the UI changes of the result from running a task in `runSimpleTaskInBrowser` are done:

```
import Data.GenHash
import RunInBrowser //or write program in same file

Start w = doTasks task w

//can also use another task that generates UI
task = runSimpleTaskInBrowser (enterInformation
    [EnterUsing (\x →x) textField])
```

When the above is run, a textfield will be shown in the browser. When you comment out the line:

```
# world = (me .# "onUIChange" .$! (toJS (encodeUIChange uiChange))) world
```

of the `runSimpleTaskInBrowser` and run the program again, you should not see a textfield as the line you commented out shows the UI changes in the browser.

B.2 Exception

We test whether the exception of the result from a task given to `runSimpleTaskInBrowser` is injected on the server-side:

```
import iTasks.WF.Tasks.Core
import RunInBrowser //or write program in same file

Start w = doTasks task w

//can also use another task that throws an exception
exception :: Task Int
exception = throw "exception text example"

task = runSimpleTaskInBrowser exception

Start w = doTasks task w
```

When running the program above, it should result in a `RuntimeError` exception shown on the server-side.

Appendix C

Test if the task given as an argument to `runSimpleTaskInBrowser` is evaluated in the browser:

```
import RunInBrowser //or write program in same file

Start w = doTasks task w

//can also use another task that uses a a lot of computations
//when evaluated in the browser:
task = get currentTimestamp
    >>- \startTs → viewSharedInformation [] currentTimestamp
        -&&- runSimpleTaskInBrowser (return (last [1..100000000]))
    >>~ \(endTs, answer) →
            (viewInformation [] (toInt endTs - toInt startTs)
            <<@ Label "Answer took (s): ")
        -&&- (viewInformation [] answer <<@ Label "Answer")
```

Running the task above should take a few seconds, indicating the task is evaluated in the browser (takes longer due to the ABC interpreter). If not, try adding another 0 to `100000000` in the task. When you go to the lines:

```
#! graph = jsSerializeGraph (Box task) abcInterpreterEnv
```

and

```
# (Box jsres, world) = jsDeserializeJSVal graph world
```

delete `Box` in both lines. When you run your program again, it should take less time than before. This indicates the server is evaluating the task.