Nuno Castro Martins

Bachelor of Computer Science and Engineering

Loom: Unifying Client-Side Web Technologies in a Single Programming Language

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Ricardo Viegas da Costa Seco,
Assistant Professor, NOVA University of Lisbon

Examination Committee

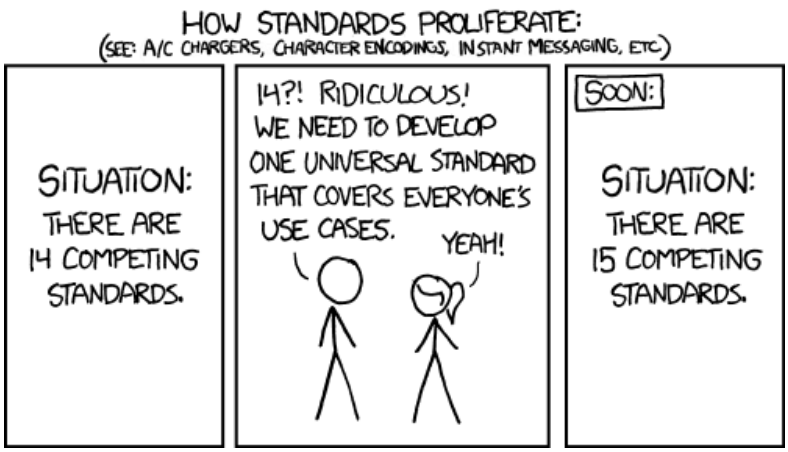
Chairperson: Jácome Cunha, NOVA University of Lisbon
Raporteur: Pedro Alves, Lusófona University
Member: João Costa Seco, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2017

Loom: Unifying Client-Side Web Technologies in a Single Programming Language

This document was created using the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt



Source: https://xkcd.com/927/

Acknowledgements

I would like to express my gratitude to everyone that supported me during the development of this work.

To my family: my parents and sister; they are all amazing people that I can always count on.

To João Costa Seco, my adviser, for supporting me when I proposed this project to him and for the thereafter availability when it came to discussing ideas.

Finally, I would like to thank all my friends who dealt with the continuous nagging that would typically sprout off my mouth as I worked through issues in the project; as well as for the hours of company they provided me with whilst my work was being developed.

This work was partially funded by NOVA LINCS (UID/CEC/04516/2013) and CLAY (PTDC/EEI-CTP/4293/2014).

Abstract

Modern client-centred web applications typically depend on a set of complementary languages to control different layers of abstraction in their interfaces: the behaviour, structure, and presentation layers (in order, traditionally: JavaScript, HTML, and CSS). Applications with dynamic interfaces whose structure and presentation depend on the data and state of the application require tight links between such layers; however, communicating between them is often non-trivial or simply cumbersome, mainly because they are effectively distinct languages—each with a specific way of being interacted with.

Numerous technologies have been introduced in an attempt to simplify the interaction between the multiple layers; their main focus so far, however, regards the communication between structure and behaviour—leaving room for improvement in the field of presentation.

This dissertation presents Loom: a novel reactive programming language that unifies the enunciated abstraction layers of a client-side web application. Loom allows the specification of an interface's structure and presentation in a declarative, data-dependent, and reactive manner by means of signals—values that change over time—inspired by the field of functional reactive programming: reactive meaning that when the structure and presentation of an interface depend on application-data, changes to said data cause an automatic update of the application's interface.

We provide an implementation of the language's compiler that allows the creation of interfaces with performance comparable to that of most existent frameworks.

Keywords: Web Programming Languages; Reactive Programming

Resumo

Aplicações *web* modernas no lado do cliente tipicamente dependem de um conjunto de linguagens e tecnologias para controlar diferentes camadas de abstração das suas interfaces: nomeadamente, para controlar o comportamento, a estrutura e a apresentação das mesmas (por ordem, tradicionalmente: JavaScript, HTML e CSS). Aplicações com vistas dinâmicas cuja estrutura e apresentação dependem dos dados e estado da aplicação requerem uma forte interligação entre as diferentes camadas; no entanto, a complexidade inerente à comunicação entre elas é muitas vezes elevada, essencialmente pelo facto de se tratarem de linguagens efetivamente distintas—sendo a interação com cada uma feita de modo diferente.

Múltiplas tecnologias têm sido introduzidas com o intuito de simplificar a interação entre as diferentes camadas; no entanto, tais tecnologias têm vindo a focar-se na comunicação entre estrutura e comportamento—sendo ainda possível melhorar a interação com a camada de apresentação.

Esta dissertação apresenta *Loom*: uma nova linguagem reativa que unifica as camadas de abstração de uma aplicação *web* no lado do cliente acima especificadas. *Loom* permite definir a estrutura e apresentação de uma interface de forma declarativa, reativa e dependente de dados da aplicação através de sinais—valores que mudam ao longo do tempo—com inspiração na área de programação funcional reativa: as interfaces são reativas no sentido em que, quando a estrutura ou apresentação da interface dependem de dados da aplicação, tal estrutura ou apresentação é automaticamente atualizada sempre que os dados se alteram.

Disponibilizamos uma implementação do compilador da linguagem que permite a criação de interfaces gráficas com desempenho comparável ao da maioria das *frameworks* atuais.

Palavras-chave: Linguagens de Programação para a Web; Programação Reativa

Contents

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

BEM Block, Element, Modifier.

CFG Context-Free Grammar.

CLI Command-Line Interface.

CSS Cascading Style Sheets.

DOM Document Object Model.

EJS Embedded JavaScript.

ES6 ECMAScript 6.

FRP Functional Reactive Programming.

GUI Graphical User Interface.

HTML Hypertext Markup Language.

PEG Parsing Expression Grammar.

REPL Read-Eval-Print-Loop.

SASS Syntactically Awesome Style Sheets.

SQL Structured Query Language.

XHTML Extensible Hypertext Markup Language.

XSS Cross-Site Scripting.

Chapter 1

Introduction

Today's industry is increasingly concerned with the notion of user experience [Gar10]. In the context of web applications, this causes developers to place a significant amount of effort in determining what is shown in the screen—the client side of the application—as well as how it is shown. As a result, and in contrast with the early days, web technologies are increasingly complex. Various technologies—from frameworks to programming languages—have been created in order to ease the process of creating such client-focused web applications; yet, throughout this chapter, we show that there is still room for improvement.

1.1 Client-Side Development of Web Applications

Modern web applications—those that benefit from placing a large part of their logic and complexity in the client (typically the web browser)—often require a combination of different languages and technologies to specify and control the structure, behaviour, and presentation of their user interface components. With user experience in mind, these applications are usually interactive and reactive, often depending on yet another set of technologies and techniques. Given the number of technologies and the complexity involved, how are these sort of applications effectively built?

Traditionally, the behaviour of user interfaces in such applications is specified by means of JavaScript, the *lingua franca* of the web. However, as web applications become increasingly more complex, and due to the shortcomings of JavaScript (*e.g.* lack of a static type system), the importance of using languages with higher orders of abstraction that compile to JavaScript has grown. There is a large number of languages that compile to JavaScript [Com], each packed with a set of features that attempt on making web development safer or simply easier (*e.g.* static typing; dealing with asynchronous code;

support for reactive computations).

Hypertext Markup Language (HTML) is the standard markup language for defining an interface's structure in a web context. As an example, we may define the structure for a simple application that manages a list of tasks using HTML with:

```
<div id="todo-app">
  <h1>Todos</h1>
  <input class="new-task" placeholder="Add a task"/>
  <ul class="tasks-list">
    <li class="task">
      <input class="task-done" type="checkbox" checked/>
      <span class="task-text">Learn HTML</span>
    </li>
    <li class="task">
      <input class="task-done" type="checkbox"/>
      <span class="task-text">Read Introduction</span>
    </li>
  </ul>
</div>
```

However, this structure is static. As HTML was originally designed to specify static documents, it is, by itself, unsuited to handle the full specification of the structure of modern web applications—mainly due to their dynamic nature. This type of application is typically data-centred, with a structure that depends on such data. Dynamically changing the structure of a given interface requires direct manipulation of the Document Object Model (DOM), usually accessed via JavaScript. The following shows a possible (unsafe[1]) way of adding new tasks to the view previously specified:[2]

```
const template = document.createElement("template")
function createTask(text) {
  template.innerHTML =
    `<li class="task">
       <input class="task-done" type="checkbox"/>
       <span class="task-text">${text}</span>
     </li>`
  return template.content.firstChild
}

const tasks = document.querySelector(".tasks-list")
const newTask = document.querySelector(".new-task")

newTask.addEventListener("keydown", evt => {
  if (evt.key === "Enter") {
    tasks.appendChild(createTask(newTask.value))
    newTask.value = ""
  }
})
```

[1] User input is not escaped—which, in real applications, may lead to Cross-Site Scripting (XSS) attacks.
[2] Working example at: https://jsfiddle.net/YarnSphere/tvt61bzp.

This type of *ad hoc* interaction with an application's structure leads to the creation of the so called "update logic", commonly implemented with the aid of strategies such as the observer pattern, where data changes are listened to and manipulations to the interface's structure via the DOM performed accordingly. The logic behind these patterns is often complex, leading to errors being easily introduced by construction and maintenance operations [Mai+10]. For this reason, a set of libraries and languages such as AngularJS [Anga]/Angular 2 [Angb], React [Rea], Vue.js [Vue], or Elm [Elm] have been introduced to ease the creation of dynamic views, usually by allowing structures to be defined declaratively whilst being reactive by design (more on this in section 4.2.2). The same todo application's structure using AngularJS[3] may be defined with:

```
<div id="todo-app" ng-app="TodoApp" ng-controller="TodoController as Todos">
  <h1>Todos</h1>
  <input class="new-task" placeholder="Add a task" ng-model="Todos.newTask"
         ng-keydown="$event.key === 'Enter' && Todos.addTask()"/>
  <ul class="tasks-list">
    <li ng-repeat="task in Todos.tasks">
      <input class="task-done" type="checkbox" ng-model="task.done"/>
      <span class="task-text">{{task.text}}</span>
    </li>
  </ul>
</div>
```

Note that there is an implicit "binding" between the structure and the application data—AngularJS is an example of a framework that allows the declarative definition of data-dependent structures that are reactive (that update themselves when the data they depend on changes).

When it comes to the presentation layer of a web interface, Cascading Style Sheets (CSS) are the standard specification used by web browsers. As applications become large, and because CSS was not initially developed with maintainability concerns in mind, CSS development can be challenging: the size of the CSS applied to the interfaces of complex web applications is often large, many times resulting in duplicated information [Maz16]. CSS preprocessors such as LESS [Les] or Syntactically Awesome Style Sheets (SASS) [Sas] were introduced in order to solve this issue (see section 4.1.2) and have become widely adopted by the industry [MT16]. However, as previously stated, modern web applications are many times dynamic and data-centred; it is not uncommon for the presentation of an interface (or at least of some of its elements) to depend on dynamic application-data. This is a concern that CSS preprocessors do not tackle, as there are limited ways of accessing the data of an application from within a style sheet;[4] often forcing presentation-logic to

[3] Working example (using AngularJS `v1.4.8`) at: `https://jsfiddle.net/YarnSphere/wknxwk8f`. We use AngularJS as an example framework for defining reactive interfaces because structures are defined using an extended HTML vocabulary—which should be simple to understand; note that other frameworks such as React could also be used in similar ways to achieve similar results.

[4] Recent CSS specifications allow element's attributes to be used as values in CSS properties via an `attr()` function; CSS custom properties may also be defined, possibly scoped to an element. This means that application-data must be directly bound to DOM elements to become visible in CSS.

be specified in JavaScript, together with behaviour-logic. Knowing this, how can reactive presentations that depend on application-data (dynamic presentations that automatically update as the data they depend on changes) be specified? This is one of the major topics of concern of this dissertation:

1.2 Reactive and Data-Dependent Presentation of Interfaces

Given the amount of complexity and number of technologies involved in the creation of modern client-focused web applications, a number of frameworks and languages have been created with the intent of bridging multiple client-side web technologies (see section 4.1.4). The bridge between HTML and JavaScript—between structure and behaviour—has been built in various languages such as Elm [Elm] or Ur/Web [Chl15] and frameworks like Angular [Angb], Meteor [Met], or React [Rea]—allowing for reactive and data-dependent interface structures to be built with relative ease.

These technologies typically support some sort of bridging with CSS—with the presentation. This is a feature that naturally reveals itself as a consequence of the already existent connection between HTML and CSS:

a) HTML elements may change their presentation via their `style` attribute. Building on top of the previous AngularJS example, this could be done by setting the `ng-style` attribute to `"task.done ? {textDecoration: 'line-through'} : {}"` in the `<span>` element with class `"task-text"`[5]: which would strikethrough each completed task;

b) CSS has access to attributes declared within HTML elements (typically classes are used), allowing for the definition of dynamic styles—provided that the element has the correct attributes set at each time. A possible way of defining the presentation of a task in CSS depending on whether it is or is not done (whether it has or has not the class `done`) is via:[6]

```
.task.done .task-text {
  text-decoration: line-through;
}
```

In AngularJS, this class could be set in the task's structure with something such as: `<li class="task"... ng-class="{done: task.done}">`.

[5] It is often considered a bad practice to set "inline-styles"; however, note that this code could be refactored by setting, for example, `ng-style` to `"Todos.style(task)"`, whilst defining the `style` function as: `task => task.done ? {textDecoration: "line-through"} : {}`.

[6] Working example at: https://jsfiddle.net/YarnSphere/q1xsyuq4; with AngularJS at: https://jsfiddle.net/YarnSphere/vm670bdo

With this in mind, we can understand how technologies that support the creation of reactive HTML values that depend on application-data, by extension, also support data-dependent reactive presentations. However, we presented two distinct ways of defining this bridge between structure/behaviour and presentation; which to use?

In late 2014, Christopher Chedeau—a member of Facebook's front-end infrastructure team—gave a presentation entitled "React: CSS in JS" [Che14] in which he proposes an approach to specify the presentation of an interface by means of JavaScript, *i.e.* using the strategy defined in item a); in this presentation Chedeau points out some of the problems that result from using CSS at a large scale—which would be the case if using the strategy in item b):

1. **Global namespace:** every identifier and class name is global in respect to CSS, making name collisions easy to encounter over time; this issue gave rise to the appearance of naming conventions such as those found in the Block, Element, Modifier (BEM) methodology [Bem];

2. **Dependencies:** when working with multiple CSS sources, each file should be imported only in pages where it is needed;

3. **Dead code elimination:** unused/redundant CSS rules output no errors; it is not uncommon for applications to end up with such rules in production [Hag+15];

4. **Minification:** names found in CSS cannot be minified as they must match their HTML counterpart (which may be dynamically set via JavaScript);

5. **Sharing constants:** as previously stated, it is common for CSS to depend on the state or data of the application, in which case values in the CSS must match their JavaScript counterparts—techniques must be used to keep these values synchronised;

6. **Non-deterministic resolution:** when a certain view requires more than one CSS file (and these files are loaded asynchronously), due to the way CSS works[7], rules appearing latest in the document may have priority (possibly making CSS dependent on the order the files are downloaded);

7. **Isolation:** Because all rules in CSS are global, rules defined with a certain element in mind may end up "leaking" to other elements; *i.e.* by specifying the presentation of a given view, one might inadvertently change the presentation of unrelated views.

This talk resulted in the creation of multiple libraries [Css] (*e.g.* Radium [Rad]) that embrace the spirit of what was proposed—writing CSS in JavaScript (see section 4.1.3). Yet, the proposed approach has some fatal flaws: there is no support for media queries; no

[7] When two rules have the same specificity—a numerical representation of the "importance" of a selector—the one appearing latest in the document has precedence.

support for CSS animations (using keyframes); and no support for selectors in general—resulting in no support for pseudo-classes (*e.g.* `:hover`, `:focus`). Whilst some of the existing libraries attempt to implement part of the missing features in JavaScript, certain aspects of CSS simply cannot be mimicked[8]; so the new question becomes: is it possible to support all CSS features whilst avoiding the problems enunciated by Chedeau? Is it possible to keep the mindset of the "CSS in JavaScript" philosophy and still support all CSS features?

As it turns out, it is possible; the main idea being: treat style sheets as first-class values defined in JavaScript and dynamically compile them to actual CSS. To avoid the isolation issue above mentioned, *apply* the style sheets to specific elements, *scoping* them to such elements in order to avoid *leaks*. This is the approach followed by libraries such as Aphrodite [Aph]. We explain this method in detail in section 3.5—as well as some of the issues with currently existent implementations.

1.3 A Language That Bridges Client-Side Web Technologies

So far we have given a broad description of the different layers of a web application, of the different technologies used in each layer, and how they can be used to specify the behaviour, structure, and presentation of user interfaces; yet, what is this dissertation all about?

This dissertation provides an attempt at defining a language-based approach to the problem of creating modern client-centred web applications. It presents Loom: a novel language that enables the definition of the behaviour, structure, and presentation of an interface whilst providing first-class support for reactivity. Loom supports HTML, CSS, and reactive values (signals) as first-class citizens, allowing the definition of data-dependent reactive interface structures and presentations by intermingling in a declarative manner the different kinds of values—done in a way that should feel familiar to nowadays web developers whilst attempting to avoid the problems of CSS at scale enunciated by Chedeau. Nevertheless, why another technology/language for the definition of web applications?

Loom was designed in response to certain flaws of state-of-the-art technologies:

- Whilst technologies have evolved in an attempt to bridge the different layers of what composes a web application, support for first-class definition of presentations—of style sheets—is still lacking, especially for reactive ones; in particular, current approaches provide such style sheets as JavaScript objects, representing most CSS syntax as strings; Loom provides CSS style sheets as first-class values with custom syntax and explicit semantics for handling reactivity (see section 2.4);

[8] As an example, the CSS pseudo-class `:visited`—which allows styling links that have been visited—cannot be mimicked in JavaScript. With privacy concerns in mind, there is no way of knowing, through JavaScript, whether a link has been visited.

- Many of the existent technologies that support the declarative definition of dynamic structures do so by extending HTML with special syntax/attributes to cause the structure to depend on data (*e.g.* `ng-if` or `ng-repeat` in AngularJS); Loom provides HTML elements as first-order values, which can be easily composed with all the expressivity of the programming language[9]—itself designed with the concern of defining dynamic structures in mind (see section 1.4);

- Most existent technologies depend on certain conventions to support reactivity (*e.g.* AngularJS' controllers and React's components: both with a notion of "state"). Loom provides first-class reactive values (signals) which are understood by HTML and CSS values alike: making Loom completely indifferent regarding how to structure an application;

- The vast majority of nowadays technologies—even when attempting to bridge behaviour, structure, and presentation—still require explicit creation of files in multiple languages (typically, at least an HTML file to mark the initial structure of the application is mandatory). Loom intends to allow the definition of a whole application with no losses regarding expressivity and without the need for HTML or CSS files to ever be (explicitly) created (see section 2.5).

Loom's key concepts, as will be possible to infer from subsequent sections, are not, by themselves, innovative—most of the concepts have already been studied; the innovation arises from the way in which we allow these concepts to interact with each other, allowing for what we argue to be an appropriate approach to the development of modern client-centred web applications.

Loom's main concepts—also first-class values in the language—are signals, HTML, and CSS:

Signals In the context of Functional Reactive Programming (FRP)—a declarative programming paradigm for working with mutable values—a signal is defined as "a value that changes over time"; regarding user interfaces, the concept of signal is useful to represent many values related to user interaction (*e.g.* the position of the cursor or the value in a text box). Loom provides first-class support for two types of signals: mutables and signal expressions. We name mutables the source for reactive computations; a mutable value is defined with an initial value and may be updated over time. On the other hand, a signal expression is a syntactical construct to declaratively combine and transform signals [Mai+10]—once defined, it may not be updated to a different expression. Signal expressions are defined with an arbitrary expression of the language: which may reference other signals. The value of the signal expression value is the result of evaluating its expression. Whilst evaluating, signal expressions create "dependencies" towards signals

[9] React's approach with JSX is similar, although their syntax extends JavaScript, which was not tailored to the task of building dynamic structures.

they access; conceptually, changes to these "dependencies" cause the signal's value to be updated—by reevaluating its expression.

HTML Loom supports HTML elements as first-class values: which may be composed to define the structure of a user interface. HTML values, in conformity with actual HTML elements, contain attributes and children: Loom allows them to be defined using the full expressivity of the language—itself designed with this task in mind. In order to support the declarative definition of reactive data-dependent structures, we extend the semantics of HTML by allowing signals to be used in (or as) attributes and in (or as) children. Conceptually, this may be seen as defining certain parts an interface's structure as dynamic, *i.e.* as changing over time. As previously mentioned, certain aspects of user interaction integrate well with signals: the value of a text box (in which a user may write) may be seen as a signal; Loom makes is possible for this kind of value to be two-way-bound—meaning that user input causes the underlying signal to update and that signal updates refresh the interface.

CSS To support the specification of the presentation of user interfaces, Loom provides CSS values as first-class. CSS values in Loom follow the semantics of actual CSS in regard to rules, selectors, and properties; however, and in similarity with CSS preprocessors, CSS rules may be nested and composed. The presentation of an interface may be specified by applying CSS values to HTML values—in contrast with the traditional approach, in Loom a style sheet is applied (and scoped) to an HTML element. In parallel with HTML values, CSS values may be used to define reactive data-dependent presentations by extending their semantics with support for signals. As such, Loom allows signals to be used as the value of CSS properties or as the body of CSS rules.

1.4 Design Principles

In this section, we present some of the principles that guided our design of the programming language. The first principle states that Loom ought to be interoperable with JavaScript, meaning that Loom should be able to easily access any value exported by a JavaScript module. The second principle concerns Loom's expressiveness in regard to HTML and CSS—which says that it should be possible to specify in the language anything that may be specified in its counterparts. The third and final design principle regards simplicity and consistency: Loom is a programming language that bridges multiple languages, each with its own specific syntax; however, it intends to make this integration whilst keeping the number of distinct concepts small and consistent with each other.

1.4.1 Interoperability With JavaScript

It is undeniable that most of the existent developer-oriented packages and frameworks in the web are targeted for the JavaScript language: npm—the default package manager for Node.js—at the time of this writing, contains over 400 thousand packages available, with over 8 billion downloads in the previous month, being the largest package ecosystem ever [Mod]. This makes interoperability with JavaScript a necessity for most developers using a new language. As such, Loom strives to be fully compatible with JavaScript: which implies being able to import JavaScript modules, as well as interact with any of their exported values with ease. With this in mind, our language was designed "around" JavaScript—in an attempt at exposing its good parts—which has the added benefit of making Loom feel similar to JavaScript for those familiar with it.

1.4.2 Expressiveness of HTML and CSS Values

Regarding the development of client-centred web applications, Loom's slogan may read as: "a language to rule them all". As conspicuous as it may sound, all it means is that Loom should be able to take care of the whole stack of technologies in an interface: structure, presentation, and behaviour. Whilst it should be possible to integrate Loom with existent technologies—as an example, an interface defined in Loom may use an external CSS style sheet—it should also be strictly optional. With this in mind, it is important that Loom be as expressive as the already existent solutions: this entails following HTML and CSS specifications to ensure that all functionality is kept, even if by different means; which brings us to:

1.4.3 Simplicity and Consistency

Loom was created in an attempt to bridge multiple client-side technologies in a single language—languages which are very distinct from one another, as they were built for altogether different purposes. As such, one of Loom's main concerns is in making this intermingling between technologies as smooth as possible. As an example, think of "conditional logic": Loom, as a programming language, should definitely have support for expressing conditionals; so how to go about defining conditional elements in an interface's structure (*i.e.* a part of an interface that should or should not be visible depending on some data)? Should HTML values have a specific way of expressing conditional logic, or should Loom provide the same construct for all kinds of conditional logic? It is here that we apply our principle of simplicity and consistency and opt for the latter option. Where possible, Loom should strive for keeping its number of core concepts small; as explained further ahead, this principle leads us to the adoption of expressions as the main building block of the language.

1.5 Introducing Loom

This section illustrates the core features of our language by means of an example: a simple task manager application where tasks may be added, deleted, set as done or active, and filtered depending on their completion status. We go over the example by defining the (mutable) data, the structure, and finally the presentation of the application. The full example may found in appendix A.1.

1.5.1 Representing Data as Signals

When declaring the variables that will contain the data of the application, a few concerns should be taken into account: is the data going to change over time? And if so, should an interface be updated depending on these changes? If the answer is positive, then this kind of data is a candidate in Loom to be defined as a signal.

Regarding the task manager application being defined, the main application data—the data that represents the state of the application—is: the *list of tasks*; and the *"state" of the filter* (which tasks to filter). Note that both of these may be seen as values that vary over time; in Loom we may define them as mutables using the `mut` keyword:

```
var tasks = mut []      // Start with an empty list of tasks
var toShow = mut "all" // One of "all", "done", or "active"
```

We previously mentioned that signals in Loom may be one of two kinds: mutables and signal expressions. Mutables are the source for reactive computations: we explicitly change their values on the occurrence of events. In this example, when a new task is added, `tasks` should be changed to a list containing the newly added task, together with all previous tasks.

So how can we define a task? A task may be seen as a record that contains some state: its *text*; and the *status of its completion*. To be able to distinguish between tasks, each task should also have a unique *identifier*. Note that whilst the identifier and text of a task are static, its completion status may change over time. We may create a function that—given the task's text and initial completion status—outputs a record representation of the task with:

```
var lastId = 0 // Generate unique identifiers
function task(text, done)
  ({id: lastId++, text: text, done: mut done})
```

A mutable's value may be accessed and assigned via a pointer-like syntax. A task may thus be added to or removed from the list of all tasks with the following functions:

```
function addTask(task)
  *tasks = *tasks.concat([task])

function removeTask(id)
  *tasks = *tasks.filter(task => task.id != id)
```

The only missing piece of data before we may define the interface of the application is the *list of tasks filtered by their completion status*. Note that this is also a value that changes over time; however, this time, it may be expressed from its relation with the list of tasks, with the completion status of each task, and with the current value of the filter. The list of filtered tasks may thus be defined as a signal expression, using the `sig` keyword:

```
var filteredTasks = sig *tasks.filter(task =>
  if (*toShow == "done") *(task.done)          // Show completed task
  else if (*toShow == "active") !*(task.done) // Show uncompleted task
  else true                                    // Show task regardless
)
```

This signal expression creates dependencies with all signals it accesses when it evaluates: `tasks`, `toShow`, and each accessed `task.done`. Updates to the dependencies cause the signal expression to reevaluate—always keeping the list of filtered tasks up-to-date.

1.5.2 Specifying the Structure of the Application

Now that we have specified the data of the application and some functions that mutate it, it's time to define its structure. We start by defining the structure of a single task by creating a function that—given the task's record representation—outputs its structure:

```
function taskStructure(task)
  <li.task> [
    <input.task-done {type: "checkbox", checked: task.done}/>
    <span.task-text> task.text
    <button.task-remove {onClick: () => removeTask(task.id)}> "Delete"
  ]
```

This example outlines some of the functionality we have previously mentioned regarding two-way bound values: `task.done`—a mutable—is applied to the `checked` attribute of a `checkbox`; this causes the `task.done` value to be updated whenever the user interacts with the check box. Further notice that we define an `onClick` event handler: causing the task to be removed from the list of all tasks when the `<button>` is pressed. The syntax and semantics of HTML values are explained in detail in section 2.3.

The remaining structure of the application may be specified with:

```
var newTaskText = mut "" // Text of the new task input box
var todoApp = <div#todo-app> [
  <h1> "Todos"
  <input.new-task {placeholder: "Add a task", value: newTaskText,
                   onKeyDown: evt => if (evt.key == "Enter") {
                                       addTask(task(*newTaskText, false))
                                       *newTaskText = ""
                                     }}/>
  <select.filter-tasks {value: toShow}>
    for (var filter in ["all", "done", "active"])
      <option {value: filter}> "Show ${filter}"
  <ul.tasks-list>
```

```
13      sig for (var task in *filteredTasks)
14        taskStructure(task)
15  ]
```

The most interesting aspect of the presented structure is arguably the usage of a signal to represent the children of the list of tasks—note that, in Loom, the `for` is a comprehension over a list that evaluates to a list (thus acting as a `map`). As we have previously mentioned, Loom allows for signals to be used virtually anywhere inside of an HTML value: in this example the signal expression maps each task of the list of filtered tasks to its HTML representation; updates to the this list cause the children of the element to be updated.

1.5.3 Data-Dependent Presentations

Finally, we change the presentation of tasks by applying a style sheet to each task. In this example, we intend to format the text of the task as strikethrough when the task is marked as complete; we may achieve this by defining a function that—given the task's record representation—outputs its desired style sheet:

```
function taskPresentation(task)
  css {
    |.task-text| {
      textDecoration: sig if (*(task.done)) "line-through" else "none"
    }
  }
```

These style sheets may then be applied to tasks with:

```
function taskStructure(task)
  <li.task {css: taskPresentation(task)}> ...
```

As we can see, Loom allows for signals to be used in CSS values with ease—making it possible to define style sheets that depend on application data and react to the changes on such data.

The presented example, as a whole, illustrates how Loom may be used to specify in a declarative manner the full structure, behaviour, and presentation of a web application.

1.6 Contributions

The contributions of this dissertation may seen as an attempt at solving each of the flaws enunciated in section 1.3 using a language-based approach, whilst being concerned with performance and with the issues raised in section 1.2 regarding the usage of CSS. We may thus summarise the contributions of our work as follows:

- We present Loom: a programming language built with the intent of easing the development of client-focused web applications. We define the three core first-class values of our language: signals, HTML, and CSS values;

- We explain how these three core building blocks may be composed in order to declaratively specify reactive and data-dependent interfaces—both structurally and in terms of presentation;
- We detail how we can take advantage of the modules of the language so that a web application may be specified in nothing but Loom;
- We explain how to efficiently render and mutate HTML values explicitly defined with dynamic parts—with signals—by taking advantage of the concept of virtual DOM;
- We propose a way of implementing scoped CSS values, attempting to mitigate the issues raised by Chedeau, whilst maintaing all the expressivity of CSS;
- We provide an initial implementation of the language's compiler (available at: `https://gitlab.com/loom-lang/loom`) which may be experimented with at: `https://loom-lang.gitlab.io/loom-playground`.

1.7 Structure of the Dissertation

The remainder of this document is structured to expand on the concepts introduced by this chapter. The content of the following chapters is thus as follows:

- Chapter 2 describes Loom: we explain the syntax and semantics of Loom's main features—informally addressed in this chapter—and relate them to existent relevant technologies;
- Chapter 3 focuses on some important implementation details, explaining the underlying technologies and how we built Loom on top of them—highlighting certain design decisions and limitations;
- Chapter 4 expands on the foundations on top of which our work was built: the current state-of-the-art in respect to technologies, techniques, and various key concepts related to this dissertation—possibly already briefly introduced in previous chapters;
- In chapter 5 we validate Loom by providing a comparison against alternatives using the TodoMVC [Tod] project as a base; we further benchmark Loom against popular frameworks—showing that Loom should work well in practice;
- Finally, in chapter 6 we reflect on our proposed language and describe the direction we expect it to follow;
- Appendix A lists the complete code for examples referenced in the dissertation.

Chapter 2

Programming Language

This chapter expands on the introductory one by presenting Loom in detail. Throughout this chapter we expand on the language's main features and the principles and foundations on top of which they were built. In particular, we explain the core of our programming language; detail the semantics and inspirations for Loom's reactive values (signals); go over Loom's first-class HTML values and their semantics; introduce important design decisions regarding CSS values; and finally, we explain how Loom's modules can be used to create fully working web applications.

Disclaimer Before we move on with the description of the core language, we would like to make a harsh observation regarding Loom: as stated previously, Loom was specified and developed with the main intent of mixing client-side web technologies—the key concepts for integrating these technologies being the notion of signals, HTML, and CSS values as first-class citizens. Most of the effort so far went into implementing such an integration in an efficient manner (more on this in chapter 3), effectively making this dissertation more of a *practical* rather than *theoretical* contribution. We expand on the consequences of this in more detail in chapter 6; however, in the context of this chapter, this means that Loom does not *yet* have a defined type system or set-in-stone operational semantics—something we believe to be of great importance in a programming language.

2.1 Core Language

Loom's main goal as a programming language is in aiding the creation of client-centred web applications. Targeting the web entails having JavaScript be output by Loom's compiler. Whilst the target compilation language shouldn't have a great influence on

Loom's design choices—with interoperability with JavaScript being one of our core design principles—the language became somewhat influenced by JavaScript. A few other influences to Loom were Scala [Scaa] and CoffeeScript [Cof].

Loom is thus a multi-paradigm programming language with support for imperative and functional programming styles. Figure 2.1 showcases a simplified version of Loom's concrete syntax where separator symbols, syntactic sugar, and certain non-relevant constructs are omitted.[1]

Although most of the keywords in Loom match their JavaScript counterparts, as we may observe—in similarity with languages such as Scala or CoffeeScript—Loom attempts to allow, when possible, constructs to be used as expressions. This brings us the benefit of being able to easily compose complex expressions—often desired when defining the structure of a user interface. Because the operational semantics of some of these expressions may not be obvious, we'll informally go over them:

- `if (e1) e2 else e3` is a typical *if then else* expression that, depending on the result of evaluating `e1`, evaluates to the evaluation of `e2` or `e3`. Note that `if (e1) e2` is a valid Loom expression and is equivalent to `if (e1) e2 else undefined`;

- All loops defined in Loom evaluate to an array where each element is the result of evaluating the loop's body in the respective step of the iteration. As an example, `for (var i in [1, 2, 3]) i * i` evaluates to `[1, 4, 9]`;

- `try e1 catch(err) e2 finally e3` is an expression that evaluates to the result of evaluating `e1` or `e2` depending on whether an error occurs during the evaluation of `e1`; the `finally` portion of the expression is only meaningful for side-effects. Similarly to the *if then else* expression, `try e1` is a valid Loom expression that is equivalent to `try e1 catch(err) undefined`. As an example, `try 1 catch(err) err` evaluates to `1` and `try throw "Error"catch(err) err` evaluates to the string `"Error"`;

- A block is an expression in Loom that evaluates to the result of evaluating its last statement after evaluating all previous statements. Note that variable declarations evaluate to `undefined`; function declarations evaluate to a value representing the function itself; and expression statements evaluate to the result of evaluating the expression. This means that `do {var x = 1; x;}` evaluates to `1`;

- The `...` operator that may be used in arrays and objects is called a *spread* operator and has the same semantics as the JavaScript counterpart[2]: it expands an expression where multiple elements (for arrays) or multiple properties (for objects) are

[1] The most notable syntactic sugar omitted from the grammar—used throughout the document in examples—regards the usage of block expressions after many of Loom's constructs. Such block expressions may have the `do` keyword omitted: which has the side effect of forcing extra parenthesis when an object is instead desired. Note that the same is done for the `css` keyword when nesting CSS values.

[2] In fact, spread object properties are, at the time of this writing, a stage 3 proposal for ECMAScript, see: https://github.com/sebmarkbage/ecmascript-rest-spread.

prog	::=	*topStat**	Program (sequence of top-level statements)
topStat	::=	`import` ID `from` STRING	Import default
	|	`import` `{` (ID (`as` ID)?)* `}` `from` STRING	Named import
	|	`export` `default` *exp*	Export default
	|	`export` *decl*	Export declaration
	|	*stat*	Statement
stat	::=	*decl*	Declaration
	|	*exp*	Expression statement
decl	::=	*varDecl*	Variable declaration
	|	`function` ID `(` ID* `)` *exp*	Function declaration
varDecl	::=	(`var` | `const`) (ID `=` *exp*)+	Variable declaration
exp	::=	`if` `(` *exp* `)` *exp* (`else` *exp*)?	Conditional expression
	|	`for` `(` *varDecl* `in` *exp* `)` *exp*	Array comprehension
	|	`for` `(` *varDecl* `of` *exp* `)` *exp*	Object comprehension
	|	`while` `(` *exp* `)` *exp*	While expression
	|	`try` *exp* (`catch` `(` ID? `)` *exp*)? (`finally` *exp*)?	Try expression
	|	`throw` *exp*	Throw expression
	|	`do` `{` *stat** `}`	Block expression
	|	*assig* `=` *exp*	Assignment
	|	⊖ *exp* | *exp* ⊕ *exp*	Unary/binary expression
	|	`[` (*exp* | `...` *exp*)* `]`	Array value
	|	`{` (ID `:` *exp* | `...` *exp*)* `}`	Object value
	|	*exp* `.` ID | *exp* `[` *exp* `]`	Selection
	|	`(` ID* `)` `=>` *exp*	Function expression
	|	*exp* `(` *exp** `)`	Function application
	|	`mut` *exp*	Mutable signal
	|	`sig` *exp*	Signal expression
	|	`*` *exp*	Signal dereferenciation
	|	`<` ID (`#` ID)? (`.` ID)* *exp*? (`/>` | `>` *exp*)	HTML value
	|	`css` `{` (ID `:` *exp* | `...` *exp* | `|` *sel** `|` *exp*)* `}`	CSS value
	|	ID	Identifier
	|	STRING | NUMBER | `true` | `false` | `null` | `undefined`	Literal
assig	::=	ID | *exp* `.` ID | *exp* `[` *exp* `]` | `*` *exp*	Assignable
sel	::=	*simpSel* ((␣ | `>` | `+`) *simpSel*)*	CSS selector
simpSel	::=	(`*` | `&` | ID)? (`#` ID | `.` ID | `:` ID (`(` ID `)`)?)*	Simple CSS selector

Figure 2.1: Simplified syntax of Loom in EBNF notation

expected. As an example, `[1, ...[2, 3], 4]` evaluates to the array `[1, 2, 3, 4]`; `{a: 1, ...{b: 2, c: 3}, c: 4}` evaluates to the object `{a: 1, b: 2, c: 4}`. This operator is useful as it enables a functional style of updating arrays/objects.

The semantics for other relevant expressions such as signal, HTML, and CSS values are explained in subsequent sections.

2.2 First-Class Reactivity

As previously mentioned in chapter 1, Loom supports signals as first-class values; but what are signals? Signals—initially introduced as behaviours—are a term used in the context of FRP to capture the temporal aspect of value mutability; they thus represent values that change over time (we go over the topic of reactive programming in more detail in section 4.2.2). Loom's signal values were heavily influenced by those of Scala.React [Mai+10] and Scala.Rx [Scab] and may be defined as one of two types: mutables (similar to `Var`s in Scala.React and React.Rx) and signal expressions (conceptually equivalent to `Signal`s in Scala.React and `Rx` in React.Rx). Let us look at each type in more detail:

Signal A signal in Loom may be seen as the common interface for both mutable values and signal expressions. All signals share the same core functionality: signals hold a *current value* (which in Loom may be accessed through the `*e` expression); signals may be *subscribed to* by other signals; and signals may be *observed*. Note that *subscribing* to a signal and *observing* a signal are two conceptually distinct actions: *subscriptions* define a dependency graph between signals—shortly, we will see that signal expressions implicitly subscribe to other signals; *observing* a signal implies running an arbitrary function whenever a signal's value changes—which may produce side-effects.

Mutable A `mut e` expression in Loom creates a mutable value—a value that may be seen as the *source* of a reactive computation. Changing the currently held value of a mutable may be done in Loom with a `*e1 = e2` expression (similarly to the `e1() = e2` expression in Scala.React and Scala.Rx). As previously mentioned, signals may subscribe to other signals; a mutable value is always a *source* in the dependency graph that such subscriptions represent: this means that an update to a mutable value conceptually causes all dependent signals to update.[3]

Signal expression Loom supports the composition of signals via signal expressions: using a `sig e` syntax. A signal expression, during the evaluation of its expression, implicitly and dynamically subscribes to signals whose values are accessed inside of its static scope. As an example, consider the declarations `var x = mut 1` and `var y = sig *x + 1`: `y` is a

[3] Note that we say *conceptually* because this behaviour is, in fact, implementation dependent (push vs. pull based propagation; see section 3.3)—it should, however, be transparent to the user.

signal expression whose value is `2` (`*y` evaluates to `2`) and whose expression accesses the value of `x`—becoming dependent of it. This means that an update to `x` such as `*x = 5` causes `*y` to evaluate to `6`. As another example, admit the same `x` and consider the function `var f = n => *x + n`: a signal such as `sig f(3)` will not subscribe to `x`—because `x` is not accessed in the signal's static scope. This behaviour is important in order to prevent the risk of adding unwanted dependencies to the signal expression. For a third example, and considering the same `x` definition, the signal `sig if (true) 5 else *x` does not depend on `x`—the signal never accesses the value of `x` during the evaluation of its expression: showcasing the dynamic detection of signal dependencies. Signal expressions are thus a powerful mechanism that allows a declarative composition of dependencies by taking advantage of all language constructs.

2.3 Dynamic Interfaces With First-Class HTML

Recall Loom's syntax for defining HTML values—which may more naturally be specified as:

html	::=	< ID (# ID)? (. ID)* *exp*? />	HTML value with no children
	|	< ID (# ID)? (. ID)* *exp*? > *exp*	HTML value with children

The expression `<t#i.c.d e1> e2` represents an HTML element with tag `t`, identifier `i`, and classes `c` and `d`; the result of evaluating `e1` represents the attributes of the element and of evaluating `e2` its children. As an example, consider:

```
<button#foo.bar.baz {type: "submit"}> [
  "Press me"
]
```

This HTML value represents the following HTML element:

```
<button id="foo" class="bar baz" type="submit">Press me</button>
```

Note that defining an HTML value with no children such as `<div/>` is equivalent, in Loom, to defining it with an empty array of children: `<div> []`.

Loom's semantics regarding HTML values are, as far as we are aware, unique in relation to other technologies due to the *kind* of values its inner expressions are allowed to evaluate to. In particular, HTML allows signals to be used virtually anywhere; fig. 2.2 showcases a possible simplified schema for the representation of an HTML value and the *kind* of values it expects in each of its fields.[4]

Though we have seen how HTML values may be *defined* in Loom—as well as what values they support—how are they effectively *rendered* to the screen? This is explained in detail in section 3.4 and briefly exposed in section 2.5; for now, let us assume that HTML values are *applied* to the DOM—making the DOM's structure match that of the applied

[4]Note that, as previously mentioned, Loom does not yet have a defined type system—being as dynamic as its target compilation language: JavaScript. However, we may see how future work involving the development of such a type system may force Loom's HTML values to follow this schema.

HTML ::= {tag: string, id: string?, classes: List[string], attrs: *Attrs*?, childr: *Childr*}

Attrs ::= Map[string, *Attr*] | Signal[Map[string, *Attr*]]

Attr ::= string | boolean | Function | Signal[string | boolean | Function]

Childr ::= List[*Child*] | Signal[List[*Child*]]

Child ::= (*HTML* | string)? | Signal[(*HTML* | string)?]

Figure 2.2: Possible simplified representation of the schema of an HTML value; the "?" notation indicates that the value may be *undefined*

HTML value. With this in mind, we can now dive into what it means for a signal to be part of an HTML value:

Conceptually, as signals are seen as values that change over time, signals used inside of HTML represent parts of the structure that change over time. This means that when a signal's value is updated, the part of the structure containing such signal is *updated*; if such structure is in fact being *rendered* on the screen, then the DOM consequently updates to match the signal's most recent value.

This updating of the DOM to make it match the structure of a signal's value effectively causes the *state* of the DOM to become consistent with that of the signal. Yet, are they *always* consistent? What about user interaction? In fact, user interaction (*e.g.* a user writing in a text box) produces changes to the *state* of the DOM—causing the interface's structure to get *out of sync* in respect to the one specified with Loom's HTML values. For this reason, Loom allows certain attributes of an element—those that may change due to user interaction (*e.g.* the `value` attribute of an `<input/>` element)—to be *aware* of user input: this can be done by setting the value of such attributes as a mutable signal—causing the signal to update whenever user input occurs. As an example, consider:

```
1 var text = mut "Initial value"
2 var elem = <input {value: text}/>
```

If `elem` is rendered on the screen, user interaction with the text box will propagate to the `text` mutable signal: conceptually keeping the DOM and the HTML value's state consistent at all times. This behaviour is typically defined as a *two-way binding* between the DOM and its *virtual* representation—in Loom's case, the HTML value.

2.4 CSS as a First-Class Citizen

CSS style sheets are, in Loom, first-class values that may be defined using the following (previously mentioned) syntax:

css ::= `css {` (ID `:` *exp* | `...` *exp* | `|` *sel** `|` *exp*)* `}` CSS value

sel ::= *simpSel* ((␣ | > | +) *simpSel*)* CSS selector

simpSel ::= (* | & | ID)? (# ID | . ID | : ID ((ID))?)* Simple CSS selector

Note that, for the sake of simplicity, the above grammar does not support all existent CSS selectors—supporting only a subset of them; however, Loom's actual grammar does.

As may be inferred from the grammar, Loom supports *nesting* of CSS rules; with semantics similar to those provided by CSS preprocessors (this is explained in more detail in section 4.1.2). As an example, consider:

```
css {
  color: "blue"
  |a > i, a:hover > b| {
    color: "green"
  }
  |&:focus| {
    color: "purple"
    |div| {
      color: "yellow"
    }
  }
}
```

This CSS value represents the following CSS style sheet, where `[scoped]` conceptually represents the element the CSS value was *applied* to (more on this below):

```
[scoped] {
  color: blue;
}
[scoped] a > i, [scoped] a:hover > b {
  color: green;
}
[scoped]:focus {
  color: purple;
}
[scoped]:focus div {
  color: yellow;
}
```

Another feature supported by Loom's CSS values is the ability to *import* other CSS values: the `...e` syntax, inspired by JavaScript's spread operator, expects `e` to be a CSS value and *imports* all of its properties and rules. This makes it possible to easily share common CSS properties by defining them elsewhere and importing them where needed. As a simple example, consider:

```
var red = css {
  backgroundColor: "red"
  border: "1px solid black"
}
var styleSheet = css {
  display: "block"
```

CSS ::= {props: SortedMap[string, *Val*], rules: SortedMap[List[*Selector*], *Body*]}

Val ::= number | string | Signal[number | string]

Body ::= *CSS* | Signal[*CSS*]

Figure 2.3: Possible representation of the schema of a CSS value; selectors' schema is not shown as it could get too complex—each type of selector would have its own schema; Sorted maps are used to represent insertion order

```
7   ...red // Import properties
8   border: "2px solid blue" // Override the property
9 }
```

This example showcases an important property of CSS: the order in which properties are defined matters—which allows for properties to be overridden.

Now that we have seen how CSS values are defined in Loom, how can they actually be used? In contrast with typical CSS style sheets, Loom's style sheets are *applied* (and *scoped*) to elements—thus avoiding the *leakage* of certain styles to unwanted parts of an interface (explained in more detail in section 3.5). CSS values may be applied to a given HTML value by using their `css` attribute. The above `styleSheet` may thus be applied to some `<div/>` HTML value with: `<div {css: styleSheet}/>`.

The final—and arguably most important—feature of Loom's style sheets lies in their ability to, in conformity with HTML values, support reactivity: allowing for the definition of dynamic data-dependent presentations. Figure 2.3 displays a simplified schema for the representation of a CSS value, highlighting the *kind* of values it expects in each of its fields. As shown, signals may be used as values of CSS properties and as the body for CSS rules—their semantics being as expected: they represent the parts of a style sheet that may change over time. HTML values rendered on the DOM that have CSS values *applied* to them will thus be able to have dynamic presentations, so long as their CSS contains such signals.

2.5 Modules: Creating a Web Application in a Single Language

A *recent* specification for JavaScript—ECMAScript 6 (ES6)—introduced the notion of *modules* in the language; with the principle of interoperability in mind, Loom's module system was designed with the same semantics as those defined in the specification. As such, Loom supports all of the available `import` and `export` syntactical constructs in ES6, even though fig. 2.1 only showcases the most useful ones.

Briefly, a file in Loom (also in ES6) is seen as a module; both *names* and a *default value* may be imported or exported by modules. As an example, consider a file containing the following:

```
1 import {foo, bar as baz} from "some-module"
```

```
import defaultValue from "other-module"
export var aNumber = 10
export default "some string"
```

This file imports `foo`—bound to local name `foo`—and `bar`—bound to local name `baz`—from `"some-module"`; as well as the default value—assigned to local name `defaultValue`—from `"other-module"`. The file further exposes the `aNumber` name and a default value containing the string `"some string"`.

In order for a web application to be launched, an HTML file has to be rendered by the browser: this typically forces users of a language or framework to create a "main" HTML file—commonly named `index.html`—to *launch* the application. Loom intends to allow the full specification of a web application in a single language; as such, we extend the semantics of our module system to allow the generation of these "main" HTML files.

This is done in Loom by exporting a *default* HTML value with an `html` tag. As an example, the following listing should generate an HTML file that may be launched:

```
var x = 10
export default <html> [
  <head> [
    <title> "Example"
  ]
  <body> [
    <span> x
  ]
]
```

Note Although intended as a core feature of the language, the current implementation of the Loom compiler does not *yet* generate HTML files as a result of exporting HTML values. This is due to Loom's mentioned lack of a type system: without which it is not possible to understand whether an exported value is indeed of type HTML value with an `html` tag.

Chapter 3

Implementation Details

Now that we have defined the language, it is time to detail how it was, in practice, implemented. This chapter showcases the implementation of Loom's compiler: we provide a JavaScript implementation of the compiler—making it possible to use it from within a web browser: Loom's playground takes advantage of this.

The compiler's source code may be found at: `https://gitlab.com/loom-lang/loom`.

Throughout the chapter, we detail how the main concepts of our language were implemented and the advantages of our approach over alternatives—as well as some of their limitations. In particular, we briefly introduce the architecture of our implementation; we expose our implementation of signals; we detail our implementation of HTML values—including how to efficiently integrate them with signals: allowing for performant DOM mutations; and finally, we explain our implementation of CSS values—including how they, in similarity with HTML, integrate with signals.

3.1 Architecture

In the general case, Loom's compiler is a tool that, when receiving a Loom program as input, transforms it into a JavaScript program. The compiler was implemented taking into consideration proper compiler design principles; fig. 3.1 displays the typical compilation process for a Loom program.

Such process may be described as follows: given a Loom program, we parse it into its Abstract Syntax Tree (AST) representation; we then compile it to an AST representation of JavaScript; this AST representation is then possibly transformed into an equivalent JavaScript AST; JavaScript source code is finally generated from the latest AST.

Note that, as previously mentioned, the compiler does not yet produce HTML files from exported HTML values; we expect this behaviour to be implemented after defining

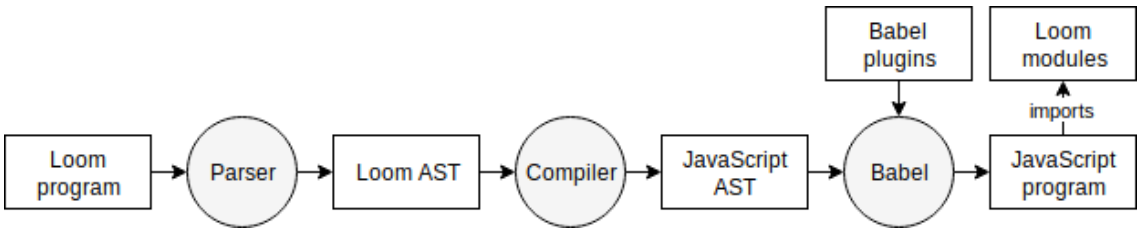


Figure 3.1: Loom's typical compilation process

and implementing a type system for the language.

The two final steps of the compilation are handled by Babel [Bab]. Babel is, in simple terms, a compiler from JavaScript to JavaScript: it is able to "understand" the most recent JavaScript syntax and compiles it to equivalent constructs using widely supported syntax. Babel further exposes its AST representation for JavaScript. From Loom's perspective, Babel is a useful library for two main reasons: we do not have to be concerned with the generation of actual JavaScript code—which is handled by Babel—only having to compile to Babel's AST representation of JavaScript; and we may compile Loom's constructs against the latest JavaScript specification without having to worry about platform support for it. In fact, because Babel's transformations work with the notion of plugins, users of Loom—by defining which plugins to use—may specify their programs to be compatible with whichever version of JavaScript they desire.

Regarding the parsing step of the compilation, the Loom compiler uses PEG.js [Maj] for the definition of Loom's grammar. PEG.js is a parser generator that, given a Parsing Expression Grammar (PEG), outputs a JavaScript parser. We define our grammar as a PEG instead of a more common Context-Free Grammar (CFG)—which, for example, Jison [Jis] understands—because some constructs in Loom would be difficult to express in a CFG without being ambiguous: PEGs just make them easier to specify.

In order to compile complex values such as signals, HTML, or CSS—whilst keeping the semantics defined in chapter 2—Loom's compiler uses *modules*: JavaScript files that we import from the generated JavaScript. These modules contain the logic required to implement Loom's semantics regarding these complex values.

3.2 Using Loom's Compiler

Before moving on with the details on how most of the language's concepts are implemented, let us first show how Loom's compiler may be used.

As previously mentioned, the source code for Loom's compiler may be found at `https://gitlab.com/loom-lang/loom`; it is also available for installation in npm via the `loom-lang` package (installed with `npm install -g loom-lang`).

Loom provides two ways of being interacted with: through a Command-Line Interface (CLI) and through a JavaScript Application Programming Interface (API).

The JavaScript API mainly exposes two functions: `parse` and `compile`. The first one receives a JavaScript string representing a Loom program and outputs the AST representation of said program; the second function takes a Loom program's AST as input as well as a set of options (*e.g.* Babel plugins to use; whether to output the JavaScript AST) and outputs the compiled JavaScript. Loom's playground (see section 3.6) takes advantage of this API in order to compile Loom programs in a browser, on the fly.

The CLI uses the JavaScript API to allow the compilation of files. At the time of this writing, Loom's CLI makes available the following options:

```
-c, --compile     output compiled code                          [boolean]
-e, --eval        evaluate compiled code                        [boolean]
-h, --help        display help message                          [boolean]
-i, --input       file used as input instead of STDIN            [string]
-o, --output      file used for output instead of STDOUT         [string]
-p, --parse       output parsed Loom AST                        [boolean]
-v, --version     display version number                        [boolean]
--cli             pass string from the command line as input     [string]
--js-ast          output compiled JavaScript AST (Babylon)      [boolean]
--repl            run an interactive Loom REPL                  [boolean]
--babel-presets   specify Babel presets to use in compilation     [array]
--babel-plugins   specify Babel plugins to use in compilation     [array]
```

As shown, Loom's compiler may be ran in interactive Read-Eval-Print-Loop (REPL) mode using the `--repl` flag.[1] This was implemented using Node.js' REPL module: each line of input is compiled as a Loom program; the generated JavaScript code is immediately evaluated and its value printed on the screen.

3.3 Implementing Signals

To honour the semantics specified in section 2.2, we provide signals as modules for Loom: this means that the generated JavaScript obtained from compiling a Loom program using signals will import the signals' module. In fact, we provide two distinct modules: one exporting a JavaScript class that represents mutable values; the other representing signal expressions. As an example, consider the following program in Loom:

```
var x = mut 10
var y = sig *x + 5
*x = 20
*y
```

The Loom compiler compiles the above program to the following (*prettified*) JavaScript:

```
import Mut from "loom-lang/mutable";
import Sig from "loom-lang/signal-expression";
let x = new Mut(10);
let y = new Sig(sig => {
  return x.subscribe(sig) + 5;
```

[1] Running Loom's compiler with no arguments also starts the interactive REPL mode.

```
6 });
7 x.setValue(20);
8 y.getValue();
```

The above mentioned modules are the ones imported in the JavaScript listing: `Mut` is a JavaScript class that represents a mutable value in Loom; `Sig` a JavaScript class representing signal expressions.

To understand the remaining generated code, let us first reveal some inner concepts regarding the signals' implementation. Recall the signals' core functionality: signals hold a *current value*; signals may be *subscribed to* by other signals; and signals may be *observed*. Furthermore: mutable values may be *assigned new values* over time; and a signal expression—during the evaluation of its expression—implicitly and dynamically subscribes to signals whose values are accessed inside its static scope. With this, we define the following pseudo-interfaces (with *pseudo-types*) for signals:

```
1  Signal[T] {
2    subscribe: Signal[?] => T,
3    observe: (T => Void) => {stop: () => Void},
4    getValue: () => T
5  }
6
7  Mutable[T] <: Signal[T] {
8    constructor: T => Mutable[T],
9    setValue: T => T
10 }
11
12 SignalExpression[T] <: Signal[T] {
13   constructor: (SignalExpression[T] => T) => SignalExpression[T]
14 }
```

Notice how signals may *subscribe* to other signals: the `subscribe` method on a signal receives the signal performing the subscription as an argument and returns the current value of the signal being subscribed.

A signal expression is constructed by providing a function whose body represents the actual *expression* of the signal and which receives the signal expression itself as an argument, *i.e.* the constructor calls the received function with `this` as argument.

Now we can understand the example showcased above for the compilation of `y`: the signal's expression, when executed, subscribes to `x`—who becomes responsible for notifying `y` whenever it changes.

However, we are yet to answer an important question regarding our implementation (previously raised in section 2.2): when `x` changes, `y` is notified; yet, when is `y`'s expression reevaluated? Two typical approaches follow from this question: a *push*-based approach would update `y` immediately; a *pull*-based approach updates `y` only when its value is effectively requested (in the example, `y` would only be reevaluated when calling `y.getValue()`). Both solutions have advantages and disadvantages (they both avoid/require needless recomputations in different situations); Loom opted for the *pull*-based

approach—the reason being that it allows us, as we will see in the following section, to define reactive HTML values that are only updated when actually being rendered in the DOM.

For the sake of completeness, there is yet another caveat in Loom's implementation of signals. Consider the following example with a signal expression `z` whose value is a function that returns the value of a signal it receives:

```
var w = mut 3
var z = sig a => *a
*z(w)
```

Loom will compile the above program to (omitting the imports):

```
let w = new Mut(3);
let z = new Sig(sig => {
  return a => {
    return a.subscribe(sig);
  };
});
z.getValue()(w);
```

If implemented naïvely, this example would cause `z` to subscribe to arbitrary signals whenever the function obtained from evaluating its expression is called with a new signal: we call this *leaking*. Yet again, recall our defined semantics for signal expressions: a signal expression—*during the evaluation of its expression*—implicitly and dynamically subscribes to signals whose values are accessed inside its static scope. This means that signal expressions should only ever subscribe to signals *whilst* their expressions are evaluating. This is why our implementation keeps track of whether the signal expression is currently being evaluated; if it is not, calling the `subscribe` method is innocuous, behaving as a `getValue` call.

3.4 HTML Values and the Virtual DOM

In similarity with signals, Loom compiles HTML values that follow the semantics specified in section 2.3 with the aid of modules. As an example, consider the following Loom program:

```
var x = mut "Hello world!"
var elem = <div#foo> [
  <span.bar {title: "baz"}> [x]
  <img {src: "http://url.com/image.jpg"}/>
]
elem.renderTo(document.getElementById("foo"))
*x = "Goodbye!"
```

Which Loom's compiler transforms into:

```
import Mut from "loom-lang/mutable";
import VElem from "loom-lang/virtual-element";
```

```
let x = new Mut("Hello world!");
let elem = new VElem("div", "foo", [], {}, [
  new VElem("span", null, ["bar"], {title: "baz"}, [x]),
  new VElem("img", null, [], {src: "http://url.com/image.jpg"}, [])
]);
elem.renderTo(document.getElementById("foo"));
x.setValue("Goodbye!");
```

As shown, Loom's HTML values are compiled into `VElem`s—virtual elements—that take the tag name, identifier, list of classes, attributes, and children of the respective HTML value as constructing arguments. We will go over virtual elements and the meaning of *virtual* shortly; let us first explain the `renderTo` portion of the code:

We have previously left in the open how exactly HTML values *render* on the screen. Rendering an element on the screen is the premise behind `renderTo` which—given an actual DOM node[2]—renders the virtual element *on top* of the given node. Assuming that the previous example was compiled into an "example.js" file and that we have the following HTML document:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <div id="foo"></div>
    <script src="./example.js"></script>
  </body>
</html>
```

Then, the `<div id="foo">` element rendered from the HTML document would be mutated to match the structure that `elem` represents: in this case, two new DOM elements—a `<span>` and an `<img>`—would be created and appended to the `<div>` with all respective attributes and children. Note that, although the `<span>` contains a signal as a child, it would render to: `<span class="bar" title="baz">Hello world!</span>`, *i.e.* the signal's value is accessed when rendering the element.

The main questions that arise when implementing HTML values—bound by the previously introduced semantics—involve how to update a structure. In the example, `x` is a signal whose value mutates at some point; how can changes to a signal affect what is displayed? And how can it be done efficiently?

Before answering these questions, it is worth pointing out a few facts: in practice, a signal inside an HTML value may evaluate to an arbitrarily complex structure (as opposed to the simple text in this example); DOM mutations are expensive, rerendering the whole interface or even a given *section* of it whenever a value that is part of such a *section* changes is impractical (see section 4.2.3): DOM manipulations should thus be kept to a minimum.

[2] It might appear like we are using the words *element* and *node* interchangeably; although not very important to the explanation, note that a DOM element (*e.g.* a `<div>`) is a *type* of DOM node; other types include DOM text nodes and comment nodes.

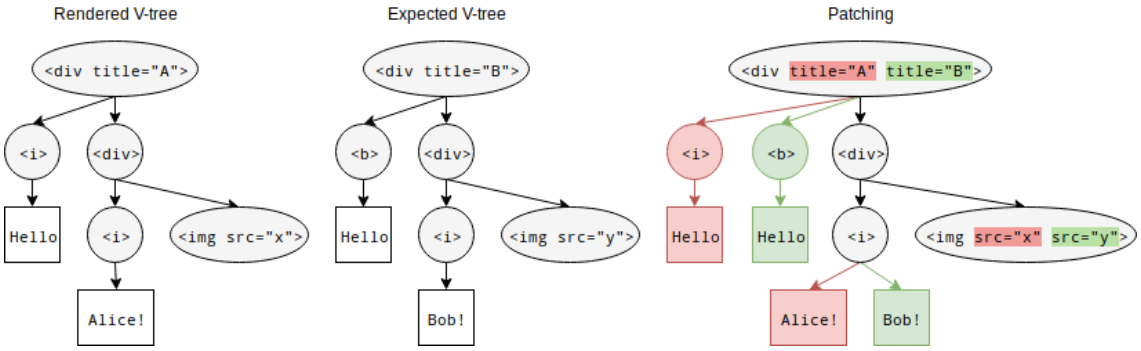


Figure 3.2: Patching of virtual trees

This is a problem that many other technologies have solved by introducing the notion of a *virtual* DOM: a structure that is a representation of the actual DOM and that allows efficient manipulation of its components. Conceptually, when an update is made to the virtual DOM, it is *diffed* against its older representation: updating the actual DOM only where this *diffing* detects changes. As one might infer, Loom's `VElem`s represent virtual elements in a virtual DOM representation.

Internally, Loom uses a library that goes by the name of Snabbdom [Sna]—a performant virtual DOM library which was low-level enough for us to adapt it to our needs: integrating it with signals. We actually *fork* Snabbdom to make it compatible with our implementation of virtual nodes but, at its core, Snabbdom still provides us a single API function: `patch`. This function takes two virtual nodes, the first one already *rendered* on the actual DOM (*i.e.* its structure represents an actual DOM structure) and a second one with the *expected* structure for the DOM (the representation of what the DOM should become) and patches the actual DOM by diffing both virtual nodes and updating it accordingly.

Figure 3.2 showcases a patching between two arbitrary virtual trees where the coloured sections of the figure represent the DOM manipulations performed by the algorithm to go from the first tree to the second. As shown, the algorithm attempts to minimise the number of DOM mutations by only updating what effectively changed.

However, how does Loom allow signals to be used within HTML values? Conceptually, our virtual elements (when rendered) *observe* any signals they may contain (in attributes or children); `patch`ing them against their previous value whenever the signals update. This approach allows us to only `patch` the sub-trees affected by a signal mutation.

As an example, consider the following Loom program and assume that `elem` has been rendered on the screen:

```
1 var person = <i> "Alice"
2 var elem = <div> [
3   <span> "Hello"
4   mut person
```

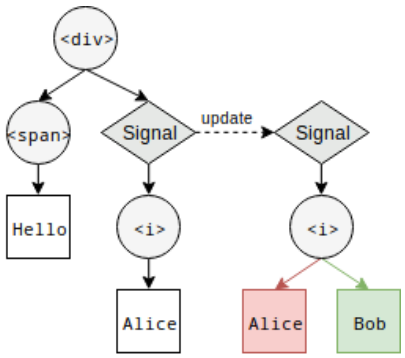


Figure 3.3: Sub-tree patching with signals

```
]
elem.renderTo(document.getElementById("foo"))
*person = <i> "Bob"
```

As shown in fig. 3.3, the signal update causes the sub-tree contained within the signal to be rerendered (using the `patch` function).

3.5 Supporting CSS Values

This section goes over the last topic left to discuss: the implementation of CSS values that follow the semantics defined in section 2.4. With this in mind, consider the following example which showcases the creation of an interface with a reactive style sheet:

```
var foo = mut "yellow"
var bar = css {display: "inline-block"}
var baz = css {
  color: foo
  backgroundColor: "green"
  |&#x > div.y:hover| {
    color: "blue"
  }
  ...bar
}
var x = <div#x {css: baz}> [
  "Hello",
  <div.y> ["World!"]
]
x.renderTo(document.getElementById("x"))
*foo = "purple"
```

This example creates and renders a `<div>` element with some children; the element has a style sheet applied to it. Note that this style sheet contains a signal (`foo`) within, associated with the `color` property. Loom's compiler, given the above input, will output the following (*prettified*) JavaScript code:

```
import Mut from "loom-lang/mutable"
```

```
import CSS from "loom-lang/css";
import CompSel from "loom-lang/composite-selector";
import SimpSel from "loom-lang/simple-selector";
import VElem from "loom-lang/virtual-element";
let foo = new Mut("yellow");
let bar = new CSS({kind: "properties",
                   properties: [["display", "inline-block"]]});
let baz = new CSS(
  {kind: "properties", properties: [["color", foo],
                                    ["backgroundColor", "green"]]},
  {kind: "rule", selectors: [
     new CompSel(">",
       new SimpSel({kind: "parent"}, {kind: "id", name: "x"}),
       new SimpSel({kind: "tag", name: "div"}, {kind: "class", name: "y"},
                   {kind: "pseudoClass", name: "hover", argument: null}))],
   body: new CSS({kind: "properties", properties: [["color", "blue"]]})},
  {kind: "import", imported: bar}
);
let x = new VElem("div", "x", [], {css: baz}, [
  "Hello"
  new VElem("div", null, ["y"], {}, ["World!"]),
]);
x.renderTo(document.getElementById("x"));
foo.setValue("purple");
```

Although this example is a bit harder to digest, it should not be difficult to map each of Loom's constructs to their JavaScript counterpart. Once again, we use modules to aid with the compilation: in particular, we define classes for representing CSS values and selectors.

`SimpSel` represents a *simple selector*—a selector without combinators; `CompSel` represents a *composite selector*—a composition of two selectors (the left one always *simple*) via a combinator (in the example, the `>` combinator).

From this example, a few important questions arise: how to apply a style sheet to an element so that it becomes scoped to it? How to handle signals inside style sheets?

Typically, regarding the first question, frameworks that support the dynamic creation of style sheets render the whole style sheet in the global scope—though first prefixing each selector with a custom class (this is the case for frameworks such as Fela [Fel] and Aphrodite [Aph]). They then add said class to each element against which the style sheet should be applied. If not being careful with the specificity of certain selectors, this may lead to an unwanted behaviour where rules of parent style sheets override more locally defined rules. As an example, consider the following Loom program:

```
var outerCSS = css {
  |#red, #blue| {
    color: "blue"
  }
}
var innerCSS = css {
```

```
  color: "red"
}
var elem = <div#z {css: outerCSS}> [
  <div#red {css: innerCSS}> "Red"
  <div#blue> "Blue"
]
elem.renderTo(document.getElementById("z"))
```

If both `outerCSS` and `innerCSS` are rendered in the global scope, then the DOM would look something like the following, where both `<div>`s will be blue (because the outer rule is more specific than the inner one):

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .outer #red, .outer #blue {
        color: blue;
      }
      .inner {
        color: red;
      }
    </style>
  </head>
  <body>
    <div id="z" class="outer">
      <div id="red" class="inner">Red</div>
      <div id="blue">Blue</div>
    </div>
  </body>
</html>
```

These frameworks usually work around this issue by limiting the kind of selectors that may be used in their style sheets—at the expense of expressivity. Typically, they do not allow descendant selectors to be used.

In order to abide by the semantics designed in the previous chapter—which were purposely defined in a way that avoids any expressivity losses—in Loom, we reuse the idea of prefixing each selector with a class (though we use an attribute selector) together with an experimental technology named *scoped styles*: which automatically have the scoping properties we desire.

In terms of the second question—how to handle signals inside style sheets—we take advantage of a recent CSS feature called *custom properties*, often referenced as CSS variables. This feature enables the usage of variables within CSS—variables whose values may be defined in elements. Most other existent frameworks require the rendering of CSS to handle this sort of dynamism within their style sheets.

As such, the following is how the element identified by `x` in the first example should *appear* in the DOM after being rendered:

```
<div id="x" data-loom-css-0 style="--loom-prop-0-0: yellow;">
  <style scoped>
    [data-loom-css-0] {
      color: var(--loom-prop-0-0);
      background-color: green;
      display: inline-block;
    }
    [data-loom-css-0]#x > div.y:hover {
      color: blue;
    }
  </style>
  Hello
  <div class="y">World!</div>
</div>
```

A mutation of the signal's value only requires the update of the `--loom-prop-0-0` custom attribute.

However, there are some issues with the proposed approach (which we intend to work on in the future; see also section 6.1):

- Scoped styles are currently only supported in Firefox—with most other browsers seemingly not interested in implementing the feature—this means that scoped style sheets become available *globally*. The web is moving in the direction of supporting *web components* which *do* support the scoping of styles—in the future, Loom may try to use this approach. Note that, whilst other browsers don't support the scoping of styles, most style sheets defined in Loom will still work properly: only certain rules with high-specificity selectors (such as the previously shown example) may *override* rules defined more locally (this makes Loom behave alike to most existent solutions);
- We place a `<style>` element on the children of HTML elements with applied style sheets: this ruins the behaviour of certain CSS selectors such as: `:first-child`, `:nth-child`, *etc.*;
- Custom properties need to be polyfilled for compatibility with older browsers;
- We have no efficient way of dealing with whole rules that are themselves signals such as `css {|div|someSignal}`: the mutation of this signal causes the rerendering of the whole style sheet.

  However, there are common situations where such rules are useful. As an example, think of the task manager application from chapter 1. We intend to style the text of a task if it is done; doing nothing when it is not done. This can be achieved via a CSS value such as the following:

```
css {
  |.task-text| sig
    if (*(task.done))
```

```
      css {
        textDecoration: "line-through"
        color: "gray"
      }
    else
      css {}
}
```

Yet, as we have explained, the toggling of the `done` status of a task will cause the whole style sheet to rerender—which may be highly inefficient. Because this is such a common pattern, we introduce a syntactical construct for expressing the above showcased example—we introduce an `:if()` pseudo-selector:

```
css {
  |.task-text:if(task.done)| {
    textDecoration: "line-through"
    color: "gray"
  }
}
```

This pseudo-selector accepts either a boolean or a signal that evaluates to a boolean; its semantics are equivalent to those of the original formulation. However, this construct can be implemented efficiently. The following shows the structure of a done task, showcasing how the `if()` pseudo-selector translates to CSS:

```
<li class="task" data-loom-css-0 data-loom-if-1-0>
  <style scoped>
    [data-loom-if-1-0][data-loom-css-0] .task-text {
      text-decoration: line-through;
      color: gray;
    }
  </style>
  <input class="task-done" type="checkbox">
  <span class="task-text">Understand Loom's CSS values</span>
</li>
```

Toggling the `done` status of the task simply requires toggling the `data-loom-if-1-0` attribute of the element in order to update its presentation.

3.6 Loom's Playground

So that users may easily experiment with our language, we make available a playground to interact with Loom. This playground, itself implemented in Loom—with a few examples to choose from—may be found at: `https://loom-lang.gitlab.io/loom-playground`. Its source code may be found at: `https://gitlab.com/loom-lang/loom-playground`.

Figure 3.4 displays the Graphical User Interface (GUI) of the playground. Note that the playground does not support the usage of multiple files; as such, examples which would normally be spread across multiple files are listed as a single one.

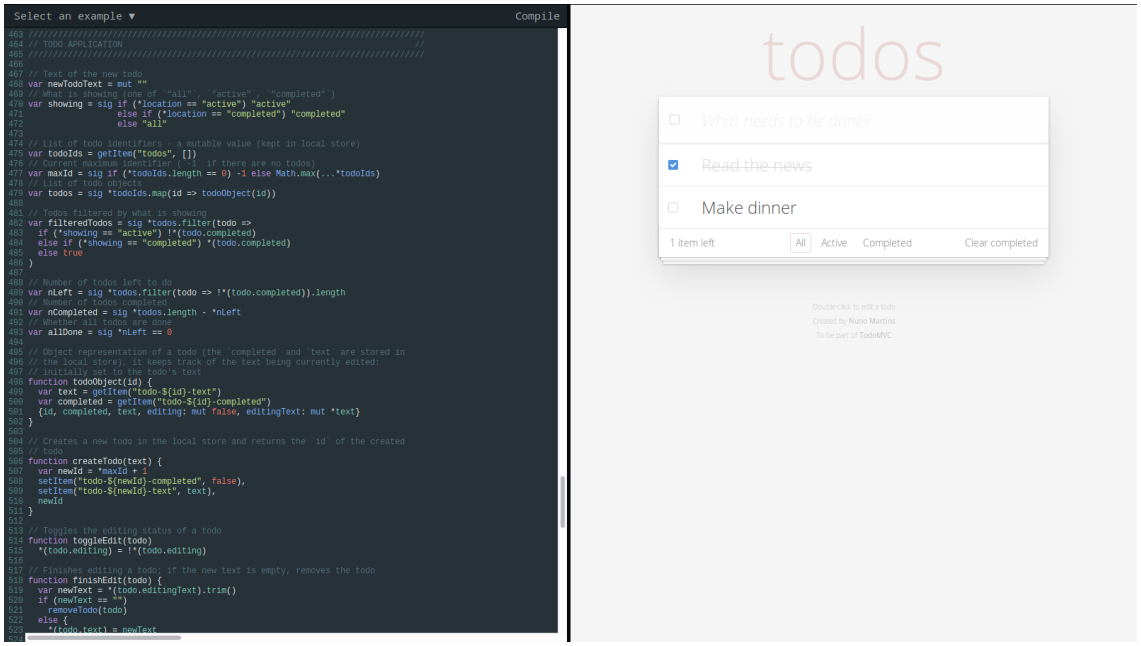


Figure 3.4: Loom's Playground GUI

Further note that the page rendered on the right side is the result of an `export default` in the Loom program on the left.

Chapter 4

Related Work

Throughout this chapter we expose some of the technologies, techniques and overall work that is related, in one way or another, to the creation of a unified web language focused on client-centred applications. In specific, we dive into the realm of web template engines, which provide useful insight on how HTML values that depend on data can be specified; we look into CSS preprocessors, which provide an extension to CSS style sheets with interesting useful semantics; we explore already existent languages and frameworks that bridge different web technologies, explaining how these bridges are built; we discuss modularity in the context of the web and its implications; and finally, we present the notion of reactive programming, discussing why it is being adopted in the web, and what techniques are used to make some of its aspects more efficient.

4.1 Background and Foundations

4.1.1 Web Template Engines

HTML was created with the idea of static documents in mind; as such, it is not suited for the definition of the structure of nowadays interfaces that, more often than not, depend on some sort of external data. In order to work around the limitations of HTML, web template engines were created: they use a processor that allows them to generate HTML documents, whilst possibly being parametrised. These engines have been widely used on the server side of applications in order to generate server-side dynamic structures that depend on some external data, such as content from an HTML form or from the state of the server. More recently, they began being applied in the client-side, thus being able to produce client-side dynamic structures.

Client-side dynamic structures change their content on the occurrence of some event, such as time or user interaction (e.g. click of a button). These sort of changes occur

by means of DOM manipulation, only made possible with the aid of a programming language with access to the DOM API (typically JavaScript). Template engines started to be adopted on the client side of applications in order to avoid the specification of a web page's dynamic elements in nothing but JavaScript, which is often either verbose, or based on the creation of HTML elements via strings.[1] As such, engines must typically offer ways of being integrated with JavaScript—providing functions such as `render`, which typically receive data as arguments and produce the rendered template as an HTML string.

Embedded JavaScript (EJS) [Ejs], Pug [Pug], and Handlebars [Han] are examples of popular template engines that integrate well with JavaScript; the study of these engines was useful in order to understand the main features of languages tailored to allow the definition of dynamic structures—which Loom supports. Bellow we find an example of how a template may look like (using Pug):

```
#todo-app
  h1 Todos
  input.new-task(placeholder="Add a task")
  if tasks.length > 0
    ul.tasks-list
      each task in tasks
        li.task
          input.task-done(type="checkbox" checked=task.done)
          span.task-text=task.text
  else
    #no-tasks No tasks to show.
```

Other engines typically support a similar set of features. The presented example shows some of the most common features of these engines: the usage of data variables (`tasks` array in the example) allows the definition of templates that vary depending on the application's data; loops and conditionals allow the specification of the template logic, where what is shown depends on the data of the application.

Other common features of template engines include syntax simplifications for common operations (in the Pug example `#todo-app` is equivalent to `div(id="todo-app")` and `li.task` to `li(class="task")`; as previously seen, Loom borrowed this idea); and inheritance, where a template may extend another.

4.1.2 CSS Preprocessors

CSS preprocessors are tools that, provided code written in some specified language, compile it to CSS — they have become widely popular among web developers as a way of bypassing some of the limitations of plain CSS [MT16]. SASS [Sas], Less [Les], and Stylus [Sty] are examples of popular CSS preprocessors [Mar14].

The study of existent CSS preprocessors was important, in our context, in order to understand what a language that features CSS values as first-class citizens is able to provide

[1] With this being said, there are still some viable alternatives based solely on JavaScript, of which HyperScript [Hyp] is a prime example.

"out-of-the-box", as well as what constructs must be added to the language in order to support some of the most useful/popular features of current preprocessors; as such, this section provides a concise description of the main capabilities of current popular CSS preprocessors.

One of the main reasons behind developers' use of CSS preprocessors instead of plain CSS is the ability to declare identifiers. Identifiers naturally eliminate the common need for repeating CSS literal values throughout a style sheet—such as colours, paddings, or widths; they are the simplest example of a feature that is automatically available in a programming language that supports CSS values.

CSS preprocessors further allow arithmetic and the use/definition of other functions. They often support multiple data types: from numbers (e.g. `17`, `4em`, `18px`), strings, and colours (e.g. `red`, `#ccc`, `rgb(0,0,100)`) to lists of values, separated by spaces or commas (e.g. `10px 0 5px 1px`, `Arial, Helvetica, sans-serif`). These functionalities should also appear naturally in a programming language containing CSS values, although it might not be trivial to determine appropriate ways of specifying the values of CSS properties (due to the nature of CSS syntax, e.g. some lists of values are represented with commas, whilst others have spaces)—Loom currently requires strings to be used to represent most CSS values; in the future, more thinking has to go into how to better integrate CSS with the language. As an example, consider the following style sheet written in SASS:

```
1 $fontColor: black;
2 #content {
3   color: $fontColor;
4   a {
5     color: lighten($fontColor, 20%);
6     &:hover { color: lighten($fontColor, 10%); }
7   }
8 }
```

Other preprocessors typically support a similar set of features, albeit with different syntax. In addition to allowing the usage of identifiers (`$fontColor` in the example) and functions (in the example, `lighten` is a function that takes a colour and a percentage as arguments and produces a new lighter colour), most preprocessors offer the ability to nest rules. The example presents such a nesting: the styles for `a` are applied when `a` is a child of `#content`. Furthermore, since `&` references the parent selector, `&:hover` represents the selector `a:hover`. The usage of nested rules when writing style sheets has become popular due to its intrinsic resemblance to the way in which HTML is specified. Indeed, by looking at style sheets written in such a way, developers are more easily able to discern a rough approximation of the structure of an interface. In Loom's case, it is also arguably the most natural way of composing a style sheet—compare it with the scoped CSS style sheets exposed in section 3.5.

Other useful features of popular CSS preprocessors include the ability to extend rules, to deal with imports, and to interpolate selectors and property names. Rule extension makes it possible for a rule to extend all properties of another rule, whilst adding new

properties; Loom supports this via the splats operator (`...`) in CSS values. Imports are supported natively by CSS; preprocessors, however, extend their functionality in order to import other files written within the same preprocessor language. In Loom, importing in unrelated with CSS; it is a concern of the module system which allows any value in Loom to be imported or exported (obviously including CSS values). Interpolating selectors and property names refers to the usage of variables inside them (e.g. `p.#{$name}{margin-#{$dir}:10px;}`). Although not introduced in section 2.4, Loom supports the interpolation of property names but not of selectors: selectors are parsed by Loom and syntactically checked by its parser (see section 6.1 as to why this may be of importance in the future).

4.1.3 CSS in JavaScript

Christopher Chedeau's talk—"React: CSS in JS"—introduced in section 1.2, enunciates a set of problems encountered when dealing with CSS in a large scale: problems which aren't solved by the adoption of CSS preprocessors.

Chedeau thus proposes an approach where JavaScript is used to define the presentation of elements of an interface. This proposal gave rise to the appearance of a plethora of JavaScript libraries [Css]: Radium [Rad] and Aphrodite [Aph] being two commonly mentioned ones.

Most of the created libraries attempt to provide an easy way for styles to be defined in JavaScript, whilst avoiding some of the issues with Chedeau's approach: it is hard/impossible to support certain CSS features with inline styles. Some of these features include: media queries, browser states (CSS pseudo-classes such as `:hover`, `:focus`, `:visited`), and keyframes.

As far as we are aware, there are two main types of "CSS in JS" libraries, in regard to their implementation: those that attempt to follow Chedeau's proposal and implement most styles as inline (this is the case for Radium); and those that, keeping the idea of inline-styles in mind, compile the styles generated in JavaScript to actual CSS: this is what Aphrodite and Fela [Fel] do behind the scenes—from where Loom got its inspiration.

The latter approach has the advantage of having no limits regarding what may be supported from CSS: since the JavaScript compiles down do CSS, all CSS functionality should be available to the user.

4.1.4 Bridging Web Technologies

Creating a bridge between some of the technologies that are common on web applications is something that has already been done, and is still being done by a number of different languages and frameworks. This section presents some of them, explaining their concerns and what can be learned (in the context of this dissertation) from what they provide. We also show that none of them attempt to offer first-class support for style sheets in order to support reactive data-dependent presentations.

Opa [Opa; RT10] is a statically typed unified web programming language whose main intent is in providing a framework suited for rapid and secure web development. It allows the specification of a whole application in a single language, with automation of client/server calls. This language has a clearly distinct goal from Loom's: it is mainly concerned with closing the gap between client, server, and database—focusing on aspects such as security and transparency. As such, they do not offer, for instance, built-in support for reactive computations. Yet, there are some shared concerns, namely in the unification of multiple languages into one: the Opa language provides Extensible Hypertext Markup Language (XHTML) as a data-type, with special syntax support; similarly, it also provides CSS as a data-type.

An example of a function in Opa that produces an XHTML paragraph (adapted from [Bin+13]) is as follows:

```
function helloWorld() {
  style = css {color: white; background: blue; padding: 10px;}
  <p style={style}>Hello world</>
}
```

This example showcases Opa's syntax for XHTML and CSS, which are intended to look just like the real thing.

CSS in Opa, however, is a data-structure that is either registered and served to the clients, becoming immutable, or is applied to XHTML values through their `style` attribute, as shown in the example. This means that, aside from manipulating the `style` attributes of XHTML elements, there is no way of creating dynamic presentations of an interface using CSS as a data-structure.

Another language recently developed for the web is Elm [CC13; Elm] — an increasingly popular strict functional programming language that compiles to JavaScript. Unlike Opa, and similarly to our language, Elm is mainly concerned with the development of client-side reactive applications. Elm supports both HTML and CSS as data-types: values of such types are produced via functions offered by the language.

A fully working example of a tasks-manager application using Elm may be found in appendix A.2. For a simpler example, the following code shows how a simple HTML element representing a user profile (with the user's picture and name) may be built using Elm (adapted from [Cza14]):

```
profile : User -> Html
profile user =
  div [class "profile"]
    [ img [src user.picture] []
    , span [style [("color", "red")]] [text user.name]
    ]
```

`div`, `img`, and `span` are functions that take a list of attributes and a list of HTML elements and produce a new HTML element. Similar functions exist for all the tags defined in the HTML5 specification. All of these are actually helpers that use the `node` function:

a function of type `String -> List Attribute -> List Html -> Html` (e.g. the `div` function is implemented as `div = node "div"`). HTML attributes are themselves specified via functions; `style` is an example of such a function and takes a list of pairs of strings as argument. This is the typical way of using CSS directly in the Elm language: by being applied to an element's `style` attribute. Thanks to the language's reactive nature (explained in 4.2.2) these styles may be dynamically updated on the event of some data-change: causing the presentation of a page to consequently update.

In contrast with Loom, Elm does not support the creation of style sheets that allow the full expressivity of CSS: at least not first-class ones with support for reactivity.

A third language that attempts to close the gap between multiple web technologies is Ur/Web [Chl15]. Ur/Web is a unified web language, resembling Opa, that supports HTML and Structured Query Language (SQL) queries as first-class values, whilst supporting reactivity in a way similar to Elm. However, it seems to lack support for primitive CSS values (again, reactivity in terms of presentation is still possible through HTML element's `style` attribute).

Angular [Angb], Meteor [Met], Ractive [Rac], and React [Rea] are examples of frameworks built with the intent of bridging web technologies. What these frameworks have in common is that, instead of defining a new language that supports values such as HTML or CSS, they offer ways of better integrating already existent technologies.

Angular, for instance, extends HTML with a set of directives that allows it to work as a template engine with reactive properties. In fact, a similarity between these frameworks is their concern with the definition of reactive structures (see section 4.2.2).

4.2 Methods and Techniques

4.2.1 Gradual typing

Gradual typing is a type system that supports both static and dynamic typing. It is a type system commonly applied on top of languages that were originally designed to be dynamic. This is the case for JavaScript, on top of which extensions such as TypeScript [Typ] or Flow [Flo], with support for gradual typing, were defined.

As previously mentioned, this methodology for adding (optional) static typing to JavaScript comes with the advantage of not losing support for already available packages (e.g. obtained through *npm*). This is not the case for other languages that compile to JavaScript whilst forcing static typing, such as Opa or Elm. In fact, in order to use external (JavaScript) packages in these languages, communication with the language must be performed through a provided API.

Even though not yet implemented in Loom, this provides us with insight on how we may, in the future, benefit from type systems whilst keeping the principle of interoperability with JavaScript in mind.

4.2.2 Reactive Programming

A common problem encountered when writing client-side web applications is the monitoring of a certain value (e.g. the width of the browser's window; some text input in a form field; or the current time) with the intent of updating some other value whenever the first one changes. Multiple patterns express this idea: polling and comparing, where the value is checked periodically and an update occurs when a change in the value is found; events, where an event is emitted every time the value changes and whoever is interested listens for such events in order to update; bindings, where values are represented by an object of some interface that allows the binding between such objects, thus forming a dependency graph that causes values to be automatically updated when a value they depend on changes. A different approach, that has become increasingly popular, is reactive programming.

Reactive programming is a declarative style of programming: this means that the programmer specifies what is supposed to happen, rather than how. Modern spreadsheet programs provide an example of reactive programming: cells that contain formulae such as `=A1+B1` that depend on other cells are updated whenever the values of those other cells change. This approach abstracts away the inner propagation of changes through the data flow.

Technologies such as Angular, Meteor, React, and Elm have popularised the usage of this technique on the web in order to create reactive interfaces. However, whilst frameworks such as Angular, Meteor, or React often force a set of conventions such as models, views, controllers, or components to be adopted, languages like Elm or Ur/Web offer an approach based on FRP.

FRP is a declarative programming paradigm for working with time-varying values, known as signals [Wan+01]. A survey on the existent set of libraries that work with signals [Mog] separates the usage of signals in two distinct categories: with combinators, or as signal expressions. Combinator libraries, such as Flapjax [Mey+09] or Elm, combine signals via functions, in order to produce new signals. Signal expressions allow the definition of signals as arbitrary expressions of the host language. These expression may depend on other signals, automatically updating whenever their value changes. This is the approach followed by Loom; also worth noting is the fact that combinator functions may still be built on top of mutable values and signal expressions.

4.2.3 Virtual DOM

Regarding the usage of reactive techniques, an important observation must be made when updating interfaces in the context of the web: DOM updates are not cheap. This is true for reactive structures specified as signals, or in any other ways. This means that it is not wise to update a whole page whenever some value that the page depends on changes. In fact, not only is it not cheap, it may compromise user experience (*e.g.* if the user has some text box selected, an update to the DOM causes the mentioned text box to lose

focus). A popular approach to work around these issues lies in the usage of a virtual DOM structure: virtual DOMs allow for lightweight updates of the structure of a page by calculating the difference between two virtual structures, and patching the real DOM with only the elements that actually changed [Cza14; Vir].

Yet, why the need for a virtual DOM? The above problems could be solved using tradicional DOM manipulations; likely, even more efficiently. The real benefit of a virtual DOM is that the DOM manipulations are automated and abstracted whilst still being performant. Doing this sort of manual management requires keeping track of what has changed and what has not, in order to avoid updating large portions of a structure that do not require an update—a process typically error prone. This is why most of nowadays frameworks that provide some way of defining reactive structures in a declarative manner use the virtual DOM techniques: this includes Elm, React, Vue.js, and many others.

Loom's virtual DOM implementation, as previously mentioned, is built on top of Snabbdom—a lightweight low-level library which has been shown to be performant in practice (see section 5.2).

Chapter 5

Validation

In this chapter we enunciate some of the ways with which we validate Loom: we implement the TodoMVC application, making it possible to compare Loom against other technologies; and we explain how our underlying technologies have been shown performant in practice.

5.1 TodoMVC

The TodoMVC is a project that offers the same tasks-manager application implemented in a plethora of different languages and frameworks [Tod]. Thanks to this project, it is possible to effectively compare multiple technologies by having a common background application. Figure 5.1 displays the expected interface of a TodoMVC application; our working version is available online at: `https://loom-lang.gitlab.io/loom-todomvc`.

In order to be able to compare Loom against other technologies, we implement the TodoMVC application with all required specifications.[1] Appendix A.3 contains the main code for our implementation of the application. It is worth noting that, as opposed to all other TodoMVC applications, all of the interface's presentation has been specified in Loom. The full source code may be found at: `https://gitlab.com/loom-lang/loom-todomvc`.

We leave to the reader the task of comparing our implementation with other implementations of the application. It is our belief that Loom's implementation is both concise and intuitive—though we might be biased. Regarding performance, we found that our application performs as well as all others for a small number of tasks; however, at about

[1] The full list of specifications may be found at: `https://github.com/tastejs/todomvc/blob/master/app-spec.md`

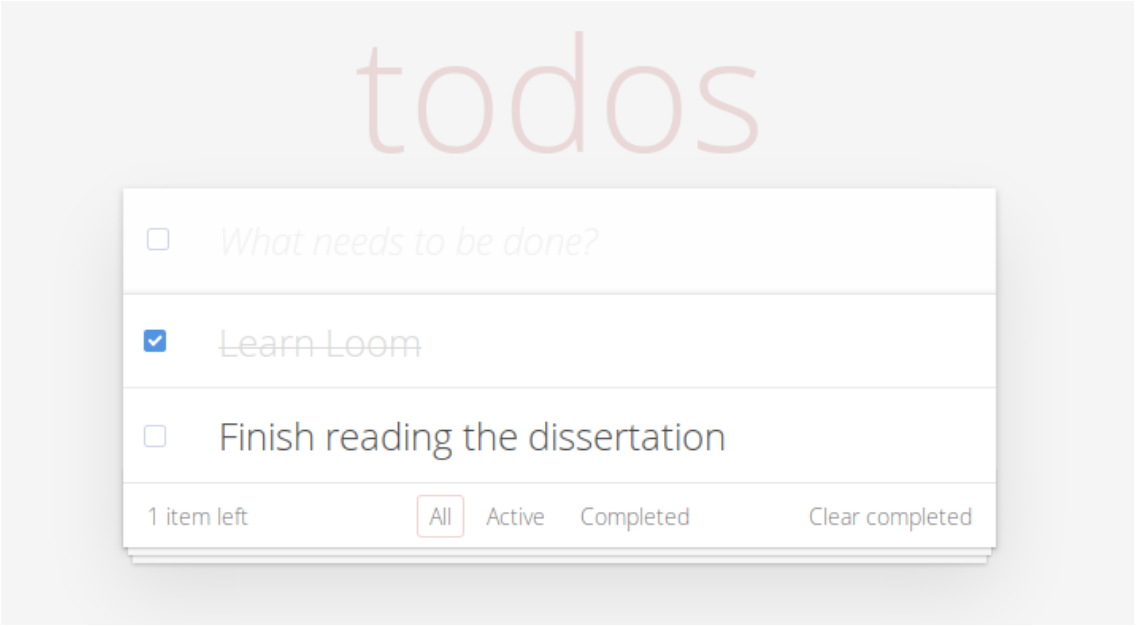


Figure 5.1: TodoMVC application (the displayed one was implemented in Loom—thought all of them look alike)

two hundred tasks, it starts to lag: this is due to our functional approach to the implementation of the application without having decent underlying data structures to support it, as well as currently unoptimised CSS support. This can be solved in two ways: either rethink Loom with native support for immutable data-structures tailored for a more functional-style of programming; or change the implementation of the application so that it does not create a new HTML value for each task every time the list of tasks is updated. In chapter 6 we see that this is an issue we intend to work on in the future.

5.2 Benchmarks

In order to benchmark our implementation of Loom's patching algorithm, we extend an already existent "JS Framework Benchmark" [Js]. This project contains implementations of multiple benchmarks for a plethora of different frameworks; as such, we provide an implementation of the benchmarks in Loom, for comparison.

The benchmarks in question consist on the creation of a large table with randomized entries and measuring the duration of various operations:

- **Create rows:** duration of creating 1000 rows;
- **Replace all rows:** duration of updating all 1000 rows of the table (with 5 warm-up iterations);

- **Partial update:** duration of updating the text of every 10th row (with 5 warm-up iterations);
- **Select row:** duration of highlighting a row in response to a click on it (with 5 warm-up iterations);
- **Swap rows:** duration of swapping 2 rows on a 1000 rows' table (with 5 warm-up iterations);
- **Remove row:** duration of removing a row (with 5 warm-up iterations);
- **Create many rows:** duration of creating 10000 rows;
- **Append rows to large table:** duration of adding 1000 rows to a table of 10000 rows;
- **Clear rows:** duration of clearing the table filled with 10000 rows;
- **Startup time:** duration for loading and parsing the JavaScript code and rendering the page;
- **Ready memory:** memory usage after page load;
- **Run memory:** memory usage after adding 1000 rows.

For all benchmarks, the duration was measured with the rendering time included. Figure 5.2 shows a comparison between Loom's results and those of two of the most popular frameworks for the client-side development of web applications: React and Angular.

As may be seen, Loom is currently not as optimised as the frameworks we are comparing it with; yet, the results also show that Loom performs very well at certain benchmarks: this is especially the case for the benchmarks where Loom takes advantage of its sub-tree patching (benchmarks 3 and 4).

We believe that Loom's poor results in benchmarks 5 and 6 are not a consequence of the employed patching algorithm, but rather of the fact that Loom requires an extra step to perform the actions: to swap two rows, Loom requires the creation of a new list that is a copy of the initial one with both rows swapped; to remove a row, Loom creates a new list that is a copy of the first one without the row to remove. Both Angular and React simply mutate the already existent list without creating a new one.

Benchmark 8, on the other hand, likely showcases the differences in terms of optimizations between Loom and the remaining libraries: which become apparent for very large structures.

With these results in mind, we can still say that Loom is currently performant enough in practice—though there is obviously still room for improvement. In all fairness, however, the presented benchmarks do not take into account the patching of CSS values—which will likely cause yet another drop in performace; as we mention in the following chapter, measuring and minimising this impact will be left for future work.

Duration in milliseconds

	Angular v4.1.2	Loom	React v15.5.4
Create rows Duration to create 1000 rows	415.87 ± 7.66 (1.03)	574.99 ± 25.56 (1.43)	401.98 ± 11.60 (1.00)
Replace all rows Duration to update all 1000 rows of the table	423.07 ± 9.66 (1.06)	521.55 ± 33.47 (1.31)	398.38 ± 10.44 (1.00)
Partial update Time to update the text of every 10th row	106.94 ± 5.10 (1.04)	102.95 ± 3.74 (1.00)	112.18 ± 5.72 (1.09)
Select row Duration to highlight a row in response to a click on it	5.97 ± 2.77 (1.00)	8.06 ± 2.11 (1.00)	8.66 ± 1.40 (1.00)
Swap rows Duration to swap 2 rows on a 1000 rows' table	30.55 ± 3.16 (1.08)	146.15 ± 12.11 (5.15)	28.41 ± 3.00 (1.00)
Remove row Duration to remove a row	116.05 ± 6.94 (1.02)	233.09 ± 14.19 (2.05)	113.69 ± 4.45 (1.00)
Create many rows Duration to create 10000 rows	4136.59 ± 60.09 (1.09)	4567.05 ± 48.96 (1.20)	3801.59 ± 59.65 (1.00)
Append rows to large table Duration for adding 1000 rows to a table of 10000 rows	568.13 ± 47.30 (1.00)	1986.86 ± 75.14 (3.50)	593.71 ± 29.21 (1.05)
Clear rows Duration to clear the table filled with 10000 rows	889.46 ± 21.64 (1.06)	1242.00 ± 20.31 (1.48)	839.27 ± 24.41 (1.00)
Startup time Time for loading, parsing, and starting up	181.61 ± 22.48 (1.35)	134.13 ± 22.96 (1.00)	168.72 ± 22.95 (1.26)
Slowdown geometric mean	**1.07**	**1.62**	**1.04**

Memory allocation in MBs

	Angular v4.1.2	Loom	React v15.5.4
Ready memory Memory usage after page load	5.13 ± 0.06 (1.36)	3.78 ± 0.00 (1.00)	4.99 ± 0.12 (1.32)
Run memory Memory usage after adding 1000 rows	10.71 ± 0.07 (1.11)	10.05 ± 0.01 (1.04)	9.69 ± 0.05 (1.00)

Figure 5.2: Loom's benchmarks compared with React and Angular; each value is followed by: ± standard deviation (slowdown = duration / fastest)

Chapter 6

Conclusions

The idea for Loom originated during the development of the client-side of some application in which many (dynamic) parts of the interface's presentation depended on application-data. We were, at the time, working with JavaScript on the client-side and Node.js on the server-side, both using the same base language—we wondered whether it would be possible to do the same for HTML and CSS: keeping all the functionality of each language whilst benefitting from the expressivity of JavaScript.

As such, so far our efforts were put into making Loom work (we followed a very end-oriented approach): we wanted to validate our idea by having a working compiler for the language that allowed us to specify performant reactive interfaces in an easy manner—this is what we believe we have achieved. We consider our implementation of the Loom compiler a very good start towards the premise of this dissertation: bridging client-side web technologies in a single programming language. Our work will thus function as ground work for the many possible improvements that we follow up by enumerating.

6.1 Future Directions

However complete our implementation of the language's compiler may be, time was limited and, as such, there are plenty of improvements to be made (both to the compiler and to the language itself). The following list showcases what we believe will be part of Loom's future:

- We intend to formally define the language: its basic constructs and behaviour regarding signals, HTML and CSS values, and the way they interact should be properly specified;
- We will explore the usage of immutable data-types for representing lists, records, and other structures of importance—we are currently compiling them to JavaScript

arrays and objects; with interoperability with JavaScript being one of our principles, this requires understanding how the JavaScript counterparts will be supported;

- We believe that a type system greatly improves the reliability of a language; as such, we intend to incorporate one in Loom: this includes a more in-depth study of existent type systems currently used in the context of the web to guarantee interoperability with JavaScript;
- The server-side rendering of web pages is important; we will likely explore the usage of Loom in the server-side of an application—which would be possible by using Node.js. This involves providing means for generating HTML and CSS documents on the fly from Loom's HTML and CSS values. Other interesting ideas in this area involve the creation of seamless connections between client and server by means of signals and web sockets, achieving similar results to those of Meteor [Met].
- Signals in Loom currently follow a *pull*-based approach with regard to *when* they update. It is possible to optimise these updates—avoiding needless recomputations—by mixing both *push* and *pull* philosophies in a *push-pull*-based approach.
- CSS in Loom requires some tweaking: although it works, its performance should be further optimised for the case where we declare the body of a rule as a signal. Another concern regards browser compatibility: currently, as previously mentioned, CSS only works *properly* in Firefox—albeit it's current behaviour is not exactly *disastrous* in other browsers. Obvious improvements to CSS involve the straightforward implementation of certain features that are not currently in Loom simply because we did not have the time to do it: key-frame support and media queries being the two main ones.
- Still in the topic of CSS, we intend to explore better ways of integrating Loom with value-types supported by CSS, i.e. support specifying units in numbers: such as `px`, `%`, `em`, etc. Once again, this is something that involves rethinking the language itself.
- Regarding CSS and type systems, because we have both HTML and CSS values as first-class citizens in Loom, we may explore ways of identifying unused styles in style sheets—a problem in nowadays applications [Hag+15].
- Although Loom's virtual DOM implementation should work well in practice, as shown in the benchmarks, there are still ways of making it better—especially because we have the freedom of compiling HTML values in any way we see fit. Libraries such as Inferno [Inf]—possibly the fastest currently existent virtual DOM library—take advantage of these kind of optimisations to achieve high levels of performance.
- At last (but not least), we intend to benchmark Loom's CSS values against both: virtual DOM libraries (to understand the impact of using "CSS in JS" *vs.* normal

CSS documents) and against other "CSS in JS" libraries (*e.g.* Fela—to compare implementation performance).

Bibliography

[Bin+13] H. Binsztok, A. Koprowski, and I. Swarczewskaja. *Opa: Up and Running*. "O'Reilly Media, Inc.", 2013. ISBN: 978-1449328856.

[Chl15] A. Chlipala. "Ur/Web: A Simple Model for Programming the Web". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 153–165. DOI: 10.1145/2676726.2677004. URL: http://doi.acm.org/10.1145/2676726.2677004.

[CC13] E. Czaplicki and S. Chong. "Asynchronous functional reactive programming for GUIs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 411–422. DOI: 10.1145/2462156.2462161. URL: http://doi.acm.org/10.1145/2462156.2462161.

[Gar10] J. J. Garrett. *Elements of user experience, the: user-centered design for the web and beyond*. Pearson Education, 2010.

[Hag+15] M. Hague, A. W. Lin, and C.-H. L. Ong. "Detecting Redundant CSS Rules in HTML5 Applications: A Tree Rewriting Approach". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/2858965.2814288.

[Mai+10] I. Maier, T. Rompf, and M. Odersky. *Deprecating the Observer Pattern*. Tech. rep. 2010.

[Mar14] A. Mardan. "Applying Stylus, Less, and Sass". In: *Pro Express. js*. Springer, 2014, pp. 181–183.

[Maz16] D. Mazinanian. "Refactoring and migration of cascading style sheets: towards optimization and improved maintainability". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 1057–1059. DOI: 10.1145/2950290.2983943. URL: http://doi.acm.org/10.1145/2950290.2983943.

[MT16] D. Mazinanian and N. Tsantalis. "An empirical study on the use of CSS preprocessors". In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 168–178.

[Mey+09] L. A. Meyerovich, A. Guha, J. P. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. "Flapjax: a programming language for Ajax applications". In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. 2009, pp. 1–20. DOI: 10.1145/1640089.1640091. URL: http://doi.acm.org/10.1145/1640089.1640091.

[Mog] R. Mogk. "Reactive Interfaces: Combining Events and Expressing Signals".

[RT10] D. Rajchenbach-Teller. "Opa: Language support for a sane, safe and secure web". In: *Proceedings of the OWASP AppSec Research* 2010 (2010).

[Wan+01] Z. Wan, W. Taha, and P. Hudak. "Real-time FRP". In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: ACM, 2001, pp. 146–156. ISBN: 1-58113-415-0. DOI: 10.1145/507635.507654. URL: http://doi.acm.org/10.1145/507635.507654.

Webography

[Anga] *Angular webpage*. URL: https://angularjs.org (visited on 03/24/2017).

[Angb] *Angular webpage*. URL: https://angular.io (visited on 03/24/2017).

[Aph] *Aphrodite repository*. URL: https://github.com/Khan/aphrodite (visited on 03/24/2017).

[Bab] *Babel webpage*. URL: http://babeljs.io (visited on 03/24/2017).

[Bem] *BEM naming convention*. URL: https://en.bem.info/methodology/naming-convention (visited on 03/24/2017).

[Che14] C. Chedeau. *React: CSS in JS*. Nov. 2014. URL: https://speakerdeck.com/vjeux/react-css-in-js (visited on 03/24/2017).

[Cof] *CoffeeScript webpage*. URL: http://coffeescript.org (visited on 03/24/2017).

[Cza14] E. Czaplicki. *Blazing Fast HTML, Virtual DOM in Elm*. July 2014. URL: http://elm-lang.org/blog/blazing-fast-html (visited on 03/24/2017).

[Ejs] *EJS webpage*. URL: http://stylus-lang.com (visited on 03/24/2017).

[Elm] *Elm webpage*. URL: http://elm-lang.org (visited on 03/24/2017).

[Fel] *Fela webpage*. URL: http://fela.js.org (visited on 03/24/2017).

[Flo] *Flow webpage*. URL: http://flowtype.org (visited on 03/24/2017).

[Han] *Handlebars webpage*. URL: http://handlebarsjs.com (visited on 03/24/2017).

[Hyp] *HyperScript repository*. URL: https://github.com/dominictarr/hyperscript (visited on 03/24/2017).

[Inf] *Inferno webpage*. URL: https://infernojs.org (visited on 03/24/2017).

[Jis] *Jison webpage*. URL: http://zaa.ch/jison (visited on 03/24/2017).

[Js] *JS framework benchmark*. URL: https://github.com/krausest/js-framework-benchmark (visited on 06/12/2017).

[Les] *Less webpage*. URL: http://lesscss.org (visited on 03/24/2017).

[Com] *List of languages that compile to JS*. URL: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS (visited on 03/24/2017).

[Maj] D. Majda. *PEG.js webpage*. URL: https://pegjs.org (visited on 03/24/2017).

[Met] *Meteor webpage*. URL: https://www.meteor.com (visited on 03/24/2017).

[Mod] *Module Counts*. URL: http://www.modulecounts.com (visited on 03/24/2017).

[Opa] *Opa webpage*. URL: http://opalang.org (visited on 03/24/2017).

[Pug] *Pug webpage*. URL: https://pugjs.org (visited on 03/24/2017).

[Rac] *Ractive.js webpage*. URL: http://www.ractivejs.org (visited on 03/24/2017).

[Rad] *Radium webpage*. URL: http://formidable.com/open-source/radium (visited on 03/24/2017).

[Css] *React: CSS in JS techniques comparison*. URL: https://github.com/MicheleBertoli/css-in-js (visited on 03/24/2017).

[Rea] *React webpage*. URL: https://facebook.github.io/react (visited on 03/24/2017).

[Sas] *SASS webpage*. URL: http://sass-lang.com (visited on 03/24/2017).

[Scaa] *Scala webpage*. URL: http://www.scala-lang.org (visited on 03/24/2017).

[Scab] *Scala.Rx repository*. URL: https://github.com/lihaoyi/scala.rx (visited on 03/24/2017).

[Sna] *Snabbdom repository*. URL: https://github.com/snabbdom/snabbdom (visited on 03/24/2017).

[Sty] *Stylus webpage*. URL: http://stylus-lang.com (visited on 03/24/2017).

[Tod] *TodoMVC webpage*. URL: http://todomvc.com (visited on 03/24/2017).

[Typ] *TypeScript webpage*. URL: http://www.typescriptlang.org (visited on 03/24/2017).

[Vir] *virtual-dom repository*. URL: https://github.com/Matt-Esch/virtual-dom (visited on 03/24/2017).

[Vue] *Vue.js webpage*. URL: https://vuejs.org (visited on 03/24/2017).

Appendix A

Full Examples

A.1 Simple Todo Application in Loom

This section showcases the full code for the example presented in section 1.5. This example creates a tasks-manager application that allows adding, removing, and filtering tasks by their completion status; as well as toggling each task's status. The interface may be observed in fig. A.1 with the full code for specifying it being:

```
var lastId = 0 // Unique identifier generator

// Initial list of tasks
var tasks = mut [task("Read the Introduction", true),
                 task("Learn Loom", true),
                 task("Learn EVERYTHING", false)]

var toShow = mut "all" // One of "all", "done", or "active"
```

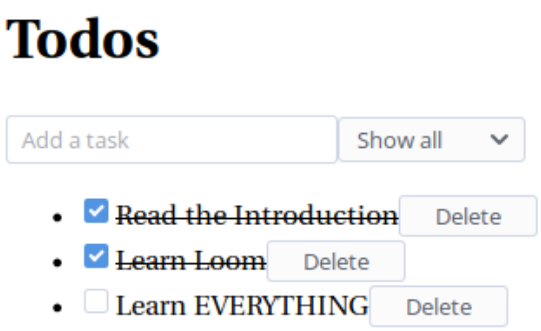


Figure A.1: Interface for the simple tasks-manager application defined in Loom as rendered by Firefox

```
9
10 // Tasks filtered by what should be shown
11 var filteredTasks = sig *tasks.filter(task =>
12   if (*toShow == "done") *(task.done)          // Show completed task
13   else if (*toShow == "active") !*(task.done) // Show uncompleted task
14   else true                                    // Show task regardless
15 )
16
17 var newTaskText = mut "" // Text of the new task input box
18
19 function task(text, done)
20   ({id: lastId++, text: text, done: mut done})
21
22 function addTask(task)
23   *tasks = *tasks.concat([task])
24
25 function removeTask(id)
26   *tasks = *tasks.filter(task => task.id != id)
27
28 function taskPresentation(task)
29   css {
30     |.task-text| {
31       textDecoration: sig if (*(task.done)) "line-through" else "none"
32     }
33   }
34
35 function taskStructure(task)
36   <li.task {key: task.id, css: taskPresentation(task)}> [
37     <input.task-done {type: "checkbox", checked: task.done}/>
38     <span.task-text> task.text
39     <button.task-remove {onClick: () => removeTask(task.id)}> "Delete"
40   ]
41
42 var todoApp = <div#todo-app> [
43   <h1> "Todos"
44   <input.new-task {placeholder: "Add a task", value: newTaskText,
45                    onKeyDown: evt => if (evt.key == "Enter") {
46                                        addTask(task(*newTaskText, false))
47                                        *newTaskText = ""
48                                      }}/>
49   <select.filter-tasks {value: toShow}>
50     for (var filter in ["all", "done", "active"])
51       <option {value: filter}> "Show ${filter}"
52   <ul.tasks-list>
53     sig for (var task in *filteredTasks)
54       taskStructure(task)
55 ]
56
57 // Create an HTML file with the application
58 export default <html> [
```

```
59    <head>
60      <title> "Simple todo application"
61    <body>
62      todoApp
63  ]
```

A.2 Simple Todo Application in Elm

In this section we list the code required to build a simple tasks-manager application in Elm. The application allows the creation of new tasks and the toggling of their completion status:

```
1  import Html exposing (..)
2  import Html.Attributes exposing (..)
3  import Html.Events exposing (..)
4  import Json.Decode as Json
5
6  main: Program Never Model Msg
7  main = Html.beginnerProgram {model = {nextId = 0, tasks = [],
8                                        newTaskText = ""},
9                               view = view, update = update}
10
11 -- MODEL
12 type alias Model = {nextId: Int, tasks: List Task, newTaskText: String }
13 type alias Task = {id: Int, text: String, done: Bool}
14
15 newTask: Int -> String -> Bool -> Task
16 newTask id text done = {id = id, text = text, done = done}
17
18 -- UPDATE
19 type Msg = UpdateNewTaskText String | AddTask | ToggleDone Int Bool
20
21 update: Msg -> Model -> Model
22 update msg model = case msg of
23   UpdateNewTaskText text -> {model | newTaskText = text}
24   AddTask -> {model | nextId = model.nextId + 1,
25                       tasks = model.tasks ++
26                               [newTask model.nextId model.newTaskText False],
27                       newTaskText = ""}
28   ToggleDone id done ->
29     let updateTask task = if task.id == id then {task | done = done} else task
30     in {model | tasks = List.map updateTask model.tasks}
31
32 -- VIEW
33 view: Model -> Html Msg
34 view model =
35   div [id "todo-app"] [
36     h1 [] [text "Todos"],
37     input [class "new-task", placeholder "Add a task",
```

```
38            value model.newTaskText, onInput UpdateNewTaskText,
39            onEnter AddTask] [],
40      ul [class "tasks-list"] <|
41        List.map taskView model.tasks
42    ]
43
44  taskView: Task -> Html Msg
45  taskView task =
46    li [class "task"] [
47      input [class "task-done", type_ "checkbox", checked task.done,
48             onClick (ToggleDone task.id (not task.done))] [],
49      span [class "task-text", style [("text-decoration",
50              if task.done then "line-through" else "none")]]
51        [text task.text]
52    ]
53
54  onEnter msg = let isEnter code = if code == 13 then Json.succeed msg
55                                   else Json.fail "Not enter"
56                in on "keydown" (Json.andThen isEnter keyCode)
```

A.3 TodoMVC Application in Loom

This section showcases the *main* code for the implementation of the TodoMVC tasks-manager application in Loom. It implements all of the project's required features whilst having all style sheets specified in Loom.

To avoid filling too many pages, we hide the CSS specification from the listing (as it was mostly a translation from the original CSS to Loom's syntax); we also omit the implementation details for the router and web storage modules. In any case, as previously mentioned, the complete source code may be found at: https://gitlab.com/loom-lang/loom-todomvc.

An observation: as future work, we will update the web storage module to support arbitrary data structures that may hold signals—making the example even simpler to understand and the usage of web storage a lot more straightforward.

This being said; the example follows:

```
1  import appStyle from "./styles/app-style"
2  import todoStyle from "./styles/todo-style"
3  import location from "./utils/router"
4  import {getItem, setItem} from "./utils/web-storage"
5
6  // Text of the new todo
7  var newTodoText = mut ""
8  // What is showing (one of `"all"`, `"active"`, `"completed"`)
9  var showing = sig if (*location == "active") "active"
10                   else if (*location == "completed") "completed"
11                   else "all"
12
```

```
13 // List of todo identifiers - a mutable value (kept in local store)
14 var todoIds = getItem("todos", [])
15 // Current maximum identifier (`-1` if there are no todos)
16 var maxId = sig if (*todoIds.length == 0) -1 else Math.max(...*todoIds)
17 // List of todo objects
18 var todos = sig *todoIds.map(id => todoObject(id))
19
20 // Todos filtered by what is showing
21 var filteredTodos = sig *todos.filter(todo =>
22   if (*showing == "active") !*(todo.completed)
23   else if (*showing == "completed") *(todo.completed)
24   else true
25 )
26
27 // Number of todos left to do
28 var nLeft = sig *todos.filter(todo => !*(todo.completed)).length
29 // Number of todos completed
30 var nCompleted = sig *todos.length - *nLeft
31 // Whether all todos are done
32 var allDone = sig *nLeft == 0
33
34 // Object representation of a todo (the `completed` and `text` are stored in
35 // the local store), it keeps track of the text being currently edited:
36 // initially set to the todo's text
37 function todoObject(id) {
38   var text = getItem("todo-${id}-text")
39   var completed = getItem("todo-${id}-completed")
40   {id, completed, text, editing: mut false, editingText: mut *text}
41 }
42
43 // Creates a new todo in the local store and returns the `id` of the created
44 // todo
45 function createTodo(text) {
46   var newId = *maxId + 1
47   setItem("todo-${newId}-completed", false),
48   setItem("todo-${newId}-text", text),
49   newId
50 }
51
52 // Toggles the editing status of a todo
53 function toggleEdit(todo)
54   *(todo.editing) = !*(todo.editing)
55
56 // Finishes editing a todo; if the new text is empty, removes the todo
57 function finishEdit(todo) {
58   var newText = *(todo.editingText).trim()
59   if (newText == "")
60     removeTodo(todo)
61   else {
62     *(todo.text) = newText
```

```
63      toggleEdit(todo)
64    }
65  }
66
67  // Cancels the edit (the editing text is reset to the todo's text)
68  function cancelEdit(todo) {
69    *(todo.editingText) = *(todo.text)
70    toggleEdit(todo)
71  }
72
73  // Adds a new todo to the list of todos when the text isn't empty
74  function addTodo() {
75    var text = *newTodoText.trim()
76    if (text != "") {
77      *todoIds = [...*todoIds, createTodo(text)]
78      *newTodoText = ""
79    }
80  }
81
82  // Removes a todo from the list of todos
83  function removeTodo(todo)
84    *todoIds = *todoIds.filter(id => id != todo.id)
85
86  // Sets all todos to either completed or not completed, depending on the
87  // status of `allDone`
88  function toggleAll() {
89    var status = !*allDone
90    for (var todo in *todos)
91      *(todo.completed) = status
92  }
93
94  // Removes all completed todos from the list of todos
95  function clearCompleted()
96    *todoIds = *todos.filter((todo) => !*(todo.completed)).map(todo => todo.id)
97
98  // Converts a todo object to its HTML representation
99  function todoHtml(todo)
100   <li {key: todo.id, css: todoStyle(todo)}> [
101     <div.view> [
102       <input.toggle {type: "checkbox", checked: todo.completed} />
103       <label {onDblClick: () => toggleEdit(todo)}> todo.text
104       <button.destroy {onClick: () => removeTodo(todo)} />
105     ]
106     sig if (*(todo.editing)) // Show input box when the todo is being edited
107       <input.edit {value: todo.editingText
108                    onBlur: () => finishEdit(todo)
109                    onKeyDown: e => if (e.key == "Enter") finishEdit(todo)
110                                    else if (e.key == "Escape") cancelEdit(todo)
111                    onInsert: (el) => el.focus()} />
112   ]
```

```
113
114 // Main view of the application
115 var todoApp = <div> [
116   <header.header> [
117     <h1> "todos"
118     // Add a new todo
119     <input.new-todo {value: newTodoText, autoFocus: true
120                      placeholder: "What needs to be done?"
121                      onKeyDown: e => if (e.key == "Enter") addTodo()} />
122   ]
123   <section.main> [
124     // Checkbox to toggle all (shown only when there are todos)
125     sig if (*todos.length > 0)
126       <input.toggle-all {type: "checkbox", checked: allDone
127                          onClick: toggleAll} />
128     // List of todos
129     <ul.todo-list> sig *filteredTodos.map(todoHtml)
130   ]
131   // Show footer only when there are todos
132   sig if (*todos.length > 0)
133     <footer.footer> [
134       <span.todo-count> [ // Number of todos left to do
135         <strong> nLeft
136         sig " item${if (*nLeft == 1) "" else "s"} left"
137       ]
138       <ul.filters> {
139         // We are using classes to style each button, we could instead have a
140         // style sheet for them
141         var setSelected = s => ({class: sig if (*showing == s) "selected"})
142         [
143           <li> <a {href: "#",           ...setSelected("all")      }> "All"
144           <li> <a {href: "#active",     ...setSelected("active")   }> "Active"
145           <li> <a {href: "#completed", ...setSelected("completed")}> "Completed"
146         ]
147       }
148       // Show button to clear completed only when there is a completed todo
149       sig if (*nCompleted > 0)
150         <button.clear-completed {onClick: clearCompleted}> "Clear completed"
151     ]
152 ]
153
154 // Exports the "main" page
155 export default <html {lang: "en"}> [
156   <head> [
157     <meta {charset: "utf-8"} />
158     <title> "Loom - TodoMVC"
159   ]
160   <body {css: appStyle}> [
161     <section.todoapp> todoApp
162     <footer.info> [
```

```
163         <p> "Double-click to edit a todo"
164         <p> ["Created by "
165              <a {href: "https://gitlab.com/nunocastromartins"}> "Nuno Martins"]
166         <p> ["To be part of ", <a {href: "http://todomvc.com"}> "TodoMVC"]
167       ]
168     ]
169 ]
```