The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/195010

Please be advised that this information was generated on 2020-10-17 and may be subject to change.
Purely Functional Web Components

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op vrijdag 12 oktober 2018
om 10:00 uur precies
door

László Domoszlai

geboren op 8 september 1977
te Boedapest (Hongarije)
Promotoren:

Prof. dr. ir. M.J. Plasmeijer
Prof. dr. Z. Horváth (Eötvös Loránd University, Boedapest, Hongarije)

Manuscriptcommissie:

Prof. dr. R.T.W. Hinze
Prof. dr. J.T. Jeuring (Universiteit Utrecht)
Dr. C. Grelck (Universiteit van Amsterdam)
## Contents

1  Introduction ........................................... 1
   1.1  Functional programming .......................... 1
   1.2  Web programming ................................. 2
   1.3  Task oriented programming and iTasks ............ 3
   1.4  The scope and organization of this thesis ......... 4

I  Client side evaluation ........................... 11

2  Implementing a non-strict purely functional language in JavaScript ................... 13
   2.1  Introduction ....................................... 13
   2.2  The SAPL programming language and interpreter .... 15
   2.3  A JavaScript based implementation for SAPL .......... 17
   2.4  Benchmarks ........................................ 25
   2.5  Applications ....................................... 29
   2.6  Related work ....................................... 31
   2.7  Conclusion and future work ....................... 32

3  Compiling Haskell to JavaScript through Clean’s core .................................. 35
   3.1  Introduction ....................................... 35
   3.2  Introduction into SAPL ............................. 37
   3.3  Introduction into the STG language ................. 39
   3.4  Discussion ......................................... 42
   3.5  Transforming STG to SAPL ......................... 42
   3.6  Interoperability issues ............................ 47
   3.7  Benchmarks ........................................ 49
   3.8  Related work ....................................... 53
   3.9  Conclusion and future work ....................... 53
7.3 Compositional Interactive Images ..................................... 147
7.4 Implementation ............................................................ 154
7.5 Related work ............................................................... 160
7.6 Conclusions and future work ............................................ 162

IV On the side 163

8 Parametric lenses: change notification for bidirectional lenses 165
  8.1 Introduction .............................................................. 165
  8.2 Introduction to Lenses .................................................. 169
  8.3 Introduction to Parametric Lenses .................................... 170
  8.4 Introduction to Parametric Views .................................... 171
  8.5 Lens combinators ....................................................... 176
  8.6 First example: Self service storage .................................. 178
  8.7 Joining multiple sources ............................................... 181
  8.8 Second example: Filtering ships ..................................... 182
  8.9 Revised second example ............................................... 186
  8.10 Related Work ........................................................... 188
  8.11 Conclusion and Future Work ........................................ 190

Bibliography 193

Summary 207

Samenvatting 209

Acknowledgements 211

Curriculum Vitae 213
1 Introduction

There have been two important long-term trends in computing technology in the last decade.

First, the World Wide Web, or shortly, the Web, became the dominant application and development platform, and it is still gaining importance. Even some of the most prevalent desktop applications, e.g. word processors and spreadsheets, have been moved to the web in the past few years. Web applications also had to become richer to compete with the user experience provided by mature desktop applications. Richer web applications tend to use more client-side application logic, making JavaScript, the only commonly available client-side programming language, one of the most used [5, 3] programming languages today.

Secondly, functional programming has been on the rise. Half a century after the first functional programming language, LISP, appeared, most major imperative programming languages adopt functional features, the ranks of functional programming languages have been increasing on programming language popularity indexes [6], and functional programming became part of the basic curriculum in Computer Science [1].

This thesis presents my research on the coupage of these areas, focusing, in particular, on how to use a functional programming language for multiple tiers of the web programming stack. This thesis consists of a selection of articles in the fields of functional programming, covering a range of topics including the compilation of a lazy functional language to JavaScript and the design of a single-language, multi-tier web component architecture embedded in the iTASK framework [113].

1.1 Functional programming

Functional programming is a declarative programming paradigm where programs are executed by evaluating expressions instead of executing statements in a fixed order, in contrast with imperative programming languages.

Purely functional languages, the group of functional languages which are the most interesting, have the important feature that functions are side-effect
free, evaluating a function can have only the effect of computing its result. This means that pure functions, using the same arguments, always evaluate to the same result (they are said to be referentially transparent), what makes their order of execution irrelevant and also makes it easier to reason about the behavior of programs.

Another characteristic feature of functional languages is the utilization of higher-order functions, functions that can take other functions as arguments or return a function as result. This feature, along with lazy evaluation (an evaluation strategy which delays the evaluation of an expression until its value is needed), is believed to be the source of advanced modularity and argued to be even more important than referential transparency [78].

Finally, one can observe a very close collaboration in the research of pure functional languages with the research of advanced type systems (which is in part due to the fact that it is hard to type side-effects). The result of this collaboration is the group of functional languages our research is focused on, the so called strongly-typed, lazy, purely functional languages.

What makes these languages interesting is that referential transparency and the strong type system eliminate a major sources of bugs, while advanced modularity, fostered by “the power and elegance of higher-order functions and lazy evaluation” [78], results in very concise programs.

1.2 Web programming

Web programming is a term that refers to the development of websites for the internet. The architecture of web applications is based on a client-server model. Websites are running on remote computers, called the web servers, uniquely identified by their URLs (Uniform Resource Locator). A user can request the service of a web server on his or her workstation (on the client) by the URL of the website, using a web browser.

The web browser is the actual platform of the client-side of the web architecture. It is basically a mediator between the user and the server maintaining communication and providing a presentation platform. The latter means that the web browser is able to display rich static content given by the declarative HyperText Markup Language (HTML) [4], which can be enhanced with dynamic behavior by embedding pieces of JavaScript in the HTML markup.

JavaScript [2] is a dynamically typed, imperative programming language executed by the web browser, and the only language that is supported by all main browsers. This is a major difference between programming the server or the client-side. While on the server one can use whichever language one wants to use, including functional ones, on the client, programming in JavaScript is inevitable.

JavaScript, however, is infamous for its many dubious language features,
including weak typing, an extensive and error prone syntax and the dependence on global variables. As Douglas Crockford, one of the authors of the JavaScript language, says in the introduction of his book titled JavaScript: The Good Parts [30]:

Most programming languages contain good and bad parts, but JavaScript has more than its share of the bad, having been developed and released in a hurry before it could be refined.

1.3 Task oriented programming and iTasks

Task Oriented Programming [113, 92] (TOP) is a paradigm for developing multi-user, distributed web-applications. The iTask system [113], or iTasks, is a TOP framework that offers a domain specific language which is embedded in the strongly typed, lazy, purely functional programming language Clean [114, 126].

In the TOP paradigm, the unit of application logic is a task. Tasks are descriptions of interactive persistent units of work that maintain a typed value. When a task is executed, it has a persistent value which may change over time reflecting the current state of the work has taken place. This task value can be observed by other tasks and tasks can also work on shared data using an abstraction called Shared Data Sources (SDS) [113].

In iTasks complex multi-user interactions can be programmed in a declarative style by defining the work that has to be accomplished. The definition is given on a high level of abstraction and does not require the programmer to provide any user interface definition. The iTask system uses generic programming [73, 12] and a hybrid static-dynamic type system [129, 128] to generate the user interface. From the programmer’s perspective this is achieved at two levels. At the most basic level, the iTasks engine can be asked to generate a graphical user interface (GUI) for any conceivable first order model type. iTasks uses a predefined set of primitive user interface elements to generate the GUI, a client-side editor, for the given type, after which it dynamically creates an associated primitive task. On the higher level, additional user interface elements are generated automatically as tasks are combined together. These elements reflect the actual combinators in use and express the “flow” of the application.

In summary, TOP offers four core concepts for software developers:

1. Tasks, which are abstractions of the work that needs to be performed.
2. Shared Data Sources (SDS), which are abstractions of information that is shared between tasks.
3. Combinator functions that compose tasks and SDSs into more complex tasks and SDSs.
4. Generic interaction with the users.

1.4 The scope and organization of this thesis

The chapters of this thesis consist of mostly unmodified versions (with the exception of Chapter 3, which was partially rewritten to improve its readability) of peer-reviewed and published articles. This results in some redundancy, most notably the introduction sections of some of the chapters may overlap.

The research of this thesis can be divided into four groups; the chapters are organized into four parts accordingly. Part I introduces a technique for transcompiling a lazy functional language to JavaScript. Based on this compilation scheme, Part II describes a web component architecture that enables the development of custom, interactive web components in a single language manner. In Part III, two non-trivial use cases of this component architecture are shown. Finally, Part IV contains one additional paper that did not fit any of the other research topics.

1.4.1 Part I

The only commonly available programming language in the web browsers is JavaScript. Thus, in order to be able to develop client-side applications in a functional language, one first has to find a way to execute such a language in a browser. There are three general options to do that:

1. Developing a browser plug-in

This is the most intrusive, thus, less favorable way to execute a functional language in a browser. It has two main problems. On the technical side, browser plug-ins are written in native code, thus must be designed to run on any combination of operating system and processor architecture. Furthermore, different browsers have different plug-in APIs, which adds one more layer of complexity. Because of these, it becomes overwhelmingly hard to maintain any widely available browser plug-in. Another problem is that browser plugins do not work out of the box, they need to be installed by the user, something that people tend to be reluctant to do. On the other hand, as plug-ins are native, they are fast. Still, this is a highly obsolete technique, but it was used in the early days [80].

2. Developing an interpreter

Another approach is to use JavaScript to interpret the functional language. Unfortunately, Javascript is not fast enough for this purpose. The overhead caused by parsing the source code and maintaining the AST is unacceptable for most purposes. It can be alleviated by developing a byte
code interpreter in JavaScript instead of an AST one. Still, the absence of certain language constructs (e.g. gotos, most importantly computed gotos) makes it impossible to develop a fast, modern threaded interpreter instead of a much slower, switch-based one [50].

3. Developing a compiler

This approach results in the fastest code execution, and it is also transparent from the perspective of the user. It is especially favorable as the dynamic nature of JavaScript enables the direct compilation of many high abstraction language constructs.

In Chapter 2 we introduce a transcompilation scheme from Clean to JavaScript using an intermediate language called SAPL [39]. SAPL is a typeless, extended lambda calculus with explicit lazy semantics that can be generated by the Clean compiler. The compilation scheme describes how to compile the language constructs, and it also explains how to implement lazy semantics by representing thunks with JavaScript arrays.

Utilizing an intermediate language like SAPL, has multiple advantages. One of these is to avoid a phenomena called the explosion of code. Transcompiling a functional program to JavaScript can result in a huge amount of code, especially when generic programming techniques are also used in the source code. This increases the page load time and the memory consumption of the browser. However, most of the time, only a part of the application is actually required for the execution, which can be exploited by compiling to JavaScript on demand from the intermediate language.

Another advantage, or side-effect, of using an intermediate language is that it can be used as a target of multiple source languages. In Chapter 3 this idea is explored by developing a series of program transformations to convert one of the core languages of the flagship Haskell compiler, GHC [100], to SAPL.

Contribution

Chapter 2 and Chapter 3 are based on the papers “L. Domoszlai, E. Bruèl and J.M. Jansen. Implementing a non-strict purely functional language in JavaScript” [39] and “L. Domoszlai and R. Plasmeijer. Compiling Haskell to JavaScript through Clean’s core” [44], respectively. My contribution to Chapter 2 is the development of the compiler (except for an early feasibility study) and writing the main part of the paper. Most of Chapter 3 is my own contribution.

Further development

With these papers, the development of the SAPL language and the SAPL to JavaScript compiler just started. Both have been significantly upgraded to
Chapter 1

achieve better performance and to reduce memory consumption (e.g. variables can be annotated now with type hints to enable better code generation, and special constructs are introduced handling records more efficiently) and now this technology is part of the latest Clean compilers. Furthermore, based on Chapter 3, the SAPL compiler infrastructure project was born [38], which implements a full-fledged GHC to JavaScript and Clean to JavaScript compiler (including run-time environments). At the moment of writing, a completely new compiler backend is being developed that utilizes its own memory management instead of being based on the garbage collector of JavaScript. This way the execution speed of the transcompiled programs is hoped to be improved significantly.

1.4.2 Part II

The Clean to JavaScript compiler is only the bottom layer of the web development stack. It enables the execution of a Clean program in a browser, but the compiled program is restricted to very basic user interactions, let alone creating rich content. To fill the gaps, in Chapter 4 we integrated it with the iTask framework using a technique called Tasklets [45].

The iTask system is developed for the construction of distributed systems where users work together on the internet. In iTasks the unit of application logic is a task. Tasks are descriptions of interactive persistent units of work that maintain a typed value. The framework offers a domain specific language embedded into the lazy functional language Clean for defining applications by combining tasks. From the mere declarative specification a complete multi-user web application is generated. Although the generated nature of the user interface (UI) entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom UI building blocks. By integrating it with the Clean to JavaScript compiler, we made a framework that generates the backbone of the application from a declarative specification, but still enables the development of custom, tailor made, interactive web components when required.

The web components are developed in a single language manner in Clean using a Model-View-Controller (MVC) approach, where the behavior of the application is encoded in Clean written event handler functions. The run-time environment, that comes with the iTask framework, seamlessly mediates JavaScript events to Clean written event handler functions. These event handlers are also state transition functions and they can interact with the browser using the exact same method as IO is done in Clean, by utilizing uniqueness typing [19].

Tasklets are implemented as special primitive tasks. This has the advantage that by being able to wrap any task into a Tasklet, tasks can be executed in the browser, effectively moving computation from the server to the client.
However, tasks are not designed to be interactive, they are only supposed to produce values. This imposes a limitation on the usefulness of Tasklets.

To overcome the limitations of Tasklets, in Chapter 5 we proposed a new web components architecture called *Editlets* [42], which is based on iTask editors instead of tasks. Editlets are type driven and enable two-way communication between the clients and the server in a way that minimizes communication overhead by sending and receiving actual changes only. Furthermore, they are also able to work on shared data (another limitation of Tasklets) as the Editlet infrastructure enables the use of an arbitrary conflict resolution strategy when data collision detected.

**Contribution**

Chapter 4 and Chapter 5 are based on the papers “L. Domoszlai and R. Plasmeijer. *Tasklets: Client-Side Evaluation for iTasks*” [45] and “L. Domoszlai, B. Lijnse and R. Plasmeijer. *Editlets: Type-based, Client-side Editors for iTasks*” [42], respectively. My contribution to Chapter 4 and 5 is the first implementation (except some iTask integration issues) and most of the writing.

**Further development**

Editlets have become a vital part of the iTask architecture to such an extent that in its recent versions Editlets replace built-in editors. Editlets were further improved to be composable and are thus able to generate the whole user interface from these type-driven, user-definable components.

As for Tasklets, Editlets made them obsolete, except for one important use case: Editlets offer all of the functionality of Tasklets with the exception of the feature to be able to run an arbitrary task on the client-side. This feature was even improved in Tasklets later on. In its first version the set of tasks that could be executed in a browser was limited to the ones not using shared data, because shared data was implemented simply as shared memory in the memory space of the iTask server application. To be able to run such tasks in a browser, we had to publish an interface to the shared data that is accessible by the clients. In the most recent, distributed versions of iTask, distributed servers also use this interface to share data between each other.

As for future work, Editlets are mostly considered finished, but we found at least one of its features worthwhile to introduce for other abstractions as well. This feature is the ability to communicate in terms of changes. We want Shared Data Sources to be read and written in terms of changes to reduce communication overhead and for smoother integration with Editlets.
To prove the usability of Tasklets and Editlets, two papers, one for Tasklets and one for Editlets, were published with advanced use cases.

Chapter 6 explains a non-traditional use case for Tasklets. It introduces an extension that, based on Tasklets, turns iTasks into an application server and a rapid development environment for client side web applications written in Clean. With this extension one can develop libraries in Clean to be used in traditional web applications. The user of such a library simply asks the iTasks server for the library. The server on demand compiles the Clean code to JavaScript, ships the source code to the browser and deploys it. In the browser the developer uses the functions of the library just as they were normal JavaScript functions. The client-side run-time environment overloads the dynamic typing capabilities of the Clean language to ensure run-time type safety of the parameters passed to the functions of the library, an important feature that was integrated with Editlets later on to provide type safety at the boundaries of the native and generated JavaScript.

Chapter 7 provides a more traditional use case for Editlets. It introduces a purely compositional library for creating interactive user interface components based on Scalable Vector Graphics (SVG) \[32\]. The graphics library is integrated with iTasks based on Editlets in such a way that one can easily switch between the generic form-like GUIs and graphics-based user interfaces. In this paper we developed a game, called Ligretto, to test the capabilities of the Editlet based on this SVG library.

**Contribution**

Chapter 6 and Chapter 7 are based on the papers “L. Domoszlai and T. Kozsik. *Clean Up the Web! - Rapid Client-Side Web Development with Clean*” [41] and “P. Achten, J. Stutterheim, L. Domoszlai and R. Plasmeijer. *Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics*” [11], respectively. My contribution to Chapter 6 is the development of the framework and the examples, and writing parts of the paper. As for Chapter 7, my primary contribution was to improve the Editlet infrastructure and the JavaScript compiler that enabled the development of the SVG editlet.

**Further development**

Editlets, besides their standard applications, are also used for some advanced use cases, e.g. to generate blueprints in the Tonic \[121\] library. Blueprints are flowchart-like graphical representation of the code that can be static or dynamic. Static blueprints simply take use of the aforementioned SVG library to display the graphical representation of the code. Dynamic blueprints, however,
are not only representations of the code, but include dynamic information as they can highlight the part of a blueprint that is currently being executed, enable step by step execution, and so on. Thus dynamic blueprints, from the perspective of Editlets, are more interesting as they take advantage of the whole feature set of the Editlet architecture, including two way communication with the server and advanced user interactions.

1.4.4 Part IV

Chapter 8 introduces parametric lenses. Parametric lenses were inspired by the Incidone application [94] while I was working on Editlets. The main screen of the application displays a map showing the ships moving around a user-specified part of the Dutch coast. To avoid polling the server by the clients, the application utilizes server-side push messages using HTML5 WebSockets [133] to keep the client-sides updated. However, there can be several thousands ships around the Dutch coast at any given time, and there can be many clients, thus it is very important to send notifications for the movements of only those ships that are actually displayed on a given client.

As the problem described above is a very common computational pattern, I wanted to find a general, reusable solution. For creating an abstract view by focusing on a specific domain of the underlying data, lenses [58] offered a well-known solution. Lenses, however, did not provide a way for change notifications.

In Chapter 8 we present a general extension to lenses as a solution for this general problem. In this extension, called parametric lenses, lenses are partially defunctionalized to extract a first-order parameter, the focus domain. The focus domain groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain is mapped to the output domain by the lens. This additional focus information will enable to read, update and observe specific parts of the underlying data.

Contribution

Chapter 8 is based on the paper “L. Domoszlai, B. Lijnse and R. Plasmeijer. Parametric Lenses: Change Notification for Bidirectional Lenses” [43]. Both the paper and the development of the related software are my own contribution.

Further development

Parametric lenses were fully integrated with the iTask system and it is in daily use.
Part I

Client side evaluation
Implementing a non-strict purely functional language in JavaScript

This paper describes an implementation of a non-strict purely functional language in JavaScript. This particular implementation is based on the translation of a high-level functional language such as Haskell or Clean into JavaScript via the intermediate functional language SAPL. The resulting code relies on the use of an evaluator function to emulate the non-strict semantics of these languages. The speed of execution is competitive with that of the original SAPL interpreter itself and better than that of other existing interpreters.

2.1 Introduction

Client-side processing for web applications has become an important research subject. Non-strict purely functional languages such as Haskell and Clean have many interesting properties, but their use in client-side processing has been limited so far. This is at least partly due to the lack of browser support for these languages. Therefore, the availability of an implementation for non-strict purely functional languages in the browser has the potential to significantly improve the applicability of these languages in this area.

Several implementations of non-strict purely functional languages in the browser already exist. However, these implementations are either based on the use of a Java Applet (e.g. for SAPL, a client-side platform for Clean [79, 80]) or a dedicated plug-in (e.g. for HaskellScript [102] a Haskell-like functional language). Both these solutions require the installation of a plug-in, which is often infeasible in environments where the user has no control over the configuration of his/her system.

2.1.1 Why switch to JavaScript?

As an alternative solution, one might consider the use of JavaScript. A JavaScript interpreter is shipped with every major browser, so that the installation of a plug-in would no longer be required. Although traditionally perceived as being slower than languages such as Java and C, the introduction of JIT compilers for JavaScript has changed this picture significantly. Modern im-
implementations of JavaScript, such as the V8 engine that is shipped with the Google Chrome browser, offer performance that sometimes rivals that of Java.

As an additional advantage, browsers that support JavaScript usually also expose their HTML DOM through a JavaScript API. This allows for the association of JavaScript functions to HTML elements through the use of event listeners, and the use of JavaScript functions to manipulate these same elements.

This notwithstanding, the use of multiple formalisms complicates the development of Internet applications considerably, due to the close collaboration required between the client and server parts of most web applications.

### 2.1.2 Results at a glance

We implemented a compiler that translates SAPL to JavaScript expressions. Its implementation is based on the representation of unevaluated expressions (thunks) as JavaScript arrays, and the just-in-time evaluation of these thunks by a dedicated evaluation function (different from the `eval` function provided by JavaScript itself).

Our final results show that it is indeed possible to realize this translation scheme in such a way that the resulting code runs at a speed competitive with that of the original SAPL interpreter itself. Summarizing, we obtained the following results:

- We realized an implementation of the non-strict purely functional programming language Clean in the browser, via the intermediate language SAPL, that does not require the installation of a plug-in.

- The performance of this implementation is competitive with that of the original SAPL interpreter and faster than that of many other interpreters for non-strict purely functional languages.

- The underlying translation scheme is straightforward, constituting a one-to-one mapping of SAPL onto JavaScript functions and expressions.

- The implementation of the compiler is based on the representation of unevaluated expressions as JavaScript arrays and the just-in-time evaluation of these thunks by a dedicated evaluation function.

- The generated code is compatible with JavaScript in the sense that the namespace for functions is shared with that of JavaScript. This allows generated code to interact with JavaScript libraries.
2.1.3 Organization of the paper

The structure of the remainder of this paper is as follows: we start with introducing SAPL, the intermediate language we intend to implement in JavaScript in Section 2.2. The translation scheme underlying this implementation is presented in Section 2.3. We present the translation scheme used by our compiler in two steps. In step one, we describe a straightforward translation of SAPL to JavaScript expressions. In step two, we add several optimizations to the translation scheme described in step one. Section 2.4 presents a number of benchmark tests for the implementation. A number of potential applications is presented in Section 2.5. Section 2.6 compares our approach with that of others. Finally, we end with our conclusions and a summary of planned future work in Section 2.7.

2.2 The SAPL programming language and interpreter

SAPL stands for Simple Application Programming Language. The original version of SAPL provided no special constructs for algebraic data types. Instead, they are represented as ordinary functions. Details on this encoding and its consequences can be found in [79]. Later a Clean like type definition style was adopted for readability and to allow for the generation of more efficient code (as will become apparent in Section 2.3).

The syntax of the language is the following:

\[
\begin{align*}
\langle \text{program} \rangle & ::= \{ \langle \text{function} \rangle \mid \langle \text{type} \rangle \} + \\
\langle \text{type} \rangle & ::= :: \langle \text{ident} \rangle \ '=>' \langle \text{ident} \rangle \ (\langle \text{ident} \rangle\ )^* \{ | \langle \text{ident} \rangle \ (\langle \text{ident} \rangle\ )^* \}^* \\
\langle \text{function} \rangle & ::= \langle \text{ident} \rangle \ (\langle \text{ident} \rangle\ )^* \ '=>' \langle \text{let-expr} \rangle \\
\langle \text{let-expr} \rangle & ::= [ \text{let} \ '\langle \text{let-defs} \rangle \ '\in' ] \langle \text{main-expr} \rangle \\
\langle \text{let-defs} \rangle & ::= \langle \text{ident} \rangle \ '=>' \langle \text{application} \rangle \{ , \langle \text{ident} \rangle \ '=>' \langle \text{application} \rangle \}^* \\
\langle \text{main-expr} \rangle & ::= \langle \text{select-expr} \rangle \mid \langle \text{if-expr} \rangle \mid \langle \text{application} \rangle \\
\langle \text{select-expr} \rangle & ::= '\text{select}' \ (\langle \text{factor} \rangle \{ (\langle \text{lambda-expr} \rangle \mid \langle \text{let-expr} \rangle \ )'\}^*) + \\
\langle \text{if-expr} \rangle & ::= '\text{if}' \ (\langle \text{factor} \rangle \ (\langle \text{let-expr} \rangle ') \ (\langle \text{let-expr} \rangle ') \\
\langle \text{lambda-expr} \rangle & ::= '\\' \langle \text{ident} \rangle + '=>' \langle \text{let-expr} \rangle
\end{align*}
\]

15
Chapter 2

\[
\langle \text{application} \rangle ::= \langle \text{factor} \rangle \langle \text{factor} \rangle^*
\]

\[
\langle \text{factor} \rangle ::= \langle \text{ident} \rangle \mid \langle \text{literal} \rangle \mid \text{'}(\langle \text{application} \rangle \text{')'}
\]

An identifier can be any identifier accepted by Clean, including operator notations. For literals characters, strings, integer or floating-point numbers and boolean values are accepted.

We illustrate the use of SAPL by giving a number of examples. We start with the encoding of the list data type, together with the \texttt{sum} function.

\[
\text{:: List} = \text{Nil} \mid \text{Cons x xs}
\]

\[
\text{sum xxs} = \text{select xxs 0 (λ x xs} = x + \text{sum xs)}
\]

The \texttt{select} keyword is used to make a case analysis on the data type of the variable xxs. The remaining arguments handle the different constructor cases in the same order as they occur in the type definition (all cases must be handled separately). Each case is a function that is applied to the arguments of the corresponding constructor.

As a more complex example, consider the \texttt{mappair} function written in Clean, which is based on the use of pattern matching:

\[
\text{mappair f Nil zs} = \text{Nil}
\]

\[
\text{mappair f (Cons x xs) Nil} = \text{Nil}
\]

\[
\text{mappair f (Cons x xs) (Cons y ys)} = \text{Cons (f x y) (mappair f xs ys)}
\]

This definition is transformed to the following SAPL function (using the above definitions for \texttt{Nil} and \texttt{Cons}).

\[
\text{mappair f as zs} = \text{select as Nil (λ x xs} = \text{select zs Nil (λ y ys} = \text{Cons (f x y) (mappair f xs ys))}
\]

SAPL is used as an intermediate formalism for the interpretation of non-strict purely functional programming languages such as Haskell and Clean. The Clean compiler includes a SAPL back-end that generates SAPL code. Recently, the Clean compiler has been extended to be able to compile Haskell programs as well [126].

2.2.1 Some remarks on the definition of SAPL

SAPL is very similar to the core languages of Haskell and Clean. Therefore, we choose not to give a full definition of its semantics. Rather, we only say something about its main characteristics and give a few examples to illustrate these.

The only keywords in SAPL are \texttt{let}, \texttt{in}, \texttt{if} and \texttt{select}. Only constant (non-function) \texttt{let} expressions are allowed that may be mutually recursive (for creating cyclic expressions). They may occur at the top level in a function and at the top level in arguments of an \texttt{if} and \texttt{select}. \texttt{λ}-expressions may
2.3 **A JavaScript based implementation for SAPL**

Section 2.1 motivated the choice for implementing a SAPL interpreter in the browser using JavaScript. Our goal was to make the implementation as efficient as possible.

Compared to Java, JavaScript provides several features that offer opportunities for a more efficient implementation. First of all, the fact that JavaScript is a *dynamic* language allows both functions and function calls to be generated at run-time, using the built-in functions `eval` and `apply`, respectively. Second, the fact that JavaScript is a dynamically *typed* language allows the creation of heterogeneous arrays. Therefore, rather than building an interpreter, we have chosen to build a compiler/interpreter hybrid that exploits the features mentioned above.

Besides these, the evaluation procedure is heavily based on the use of the `typeof` operator and the runtime determination of the number of formal parameters of a function which is another example of the dynamic properties of the JavaScript language.

For the following SAPL constructs we must describe how they are translated to JavaScript:

- literals, such as booleans, integers, real numbers, and strings;
- identifiers, such as variable and function names;
- function definitions;
- constructor definitions;
- let constructs;
- applications;
- select statements;
- if statements;
- built-in functions, such as `add`, `eq`, etc.

**Literals**

Literals do not have to be transformed. They have the same representation in SAPL and JavaScript.
Identifiers

Identifiers in SAPL and JavaScript share the same namespace, therefore, they need not to be transformed either.

However, the absence of block scope in JavaScript can cause problems. The scope of variables declared using the `var` keyword is hoisted to the entire containing function. This affects the `let` construct and the $\lambda$-expressions, but can be easily avoided by postfixing the declared identifiers to be unique. In this way, the original variable name can be restored if needed.

With this remark we will neglect these transformations in the examples of this paper for the sake of readability.

Function definitions

Due to JavaScript’s support for higher-order functions, function definitions can be translated from SAPL to JavaScript in a straightforward manner:

\[
T[\text{j} f x_1 \ldots x_n = \text{body}] = \text{function } f(x_1, \ldots, x_n) \{ T[\text{body}] \}
\]

So SAPL functions are mapped one-to-one to JavaScript functions with the same name and the same number of arguments.

Constructor definitions

Constructor definitions in SAPL are translated to arrays in JavaScript, in such a way that they can be used in a `select` construct to select the right case. A SAPL type definition containing constructors is translated as follows:

\[
T[\text{j} :: \text{typename} = \ldots | C_k x_{k0} \ldots x_{kn} | \ldots] = \ldots \text{function } C_k(x_{k0}, \ldots, x_{kn}) \{ \text{return} [k, \ 'C_k', x_{k0}, \ldots, x_{kn}]; \} \ldots
\]

where $k$ is a positive integer, corresponding to the position of the constructor in the original type definition. The name of the constructor, ‘$C_k$’, is put into the result for printing purposes only. This representation of the constructors together with the use of the `select` statement allows for a very efficient JavaScript translation of the SAPL language.

Let constructs

Let constructs are translated differently depending on whether they are cyclic or not. Non-cyclic lets in SAPL can be translated to `var` declarations in JavaScript as follows:

\[
T[\text{j} let \ x = e \ in \ b] = \text{var } x = T[\text{j} e]; T[\text{j} b]
\]

Due to JavaScript’s support for closures, cyclic lets can be translated from SAPL to JavaScript in a straightforward manner. The idea is to take any occurrences of $x$ in $e$ and replace them with:
Implementing a non-strict purely functional language in Java-Script

function () { return x; }

This construction relies on the fact that the scope of a JavaScript closure is the whole function itself. This means that after the declaration the call of this closure will return a valid reference. In Section 2.3.1 we present an example to illustrate this.

Applications

Every SAPL expression is an application. Due to JavaScript’s eager evaluation semantics, applications cannot be translated from SAPL to JavaScript directly. Instead, unevaluated expressions (or *thunks*) in SAPL are translated to arrays in JavaScript:

$$\langle x_0 \ x_1 \ldots \ x_n \rangle = [\underbrace{\langle x_0 \rangle}, \underbrace{\langle x_1 \rangle}, \ldots, \underbrace{\langle x_n \rangle}]$$

Thus, a thunk is represented with an array of two elements. The first one is the function involved, and the second one is an array of the arguments. This second array is used for performance reasons. In this way one can take advantage of the JavaScript *apply()* method and it is very straightforward and fast to join such two arrays, which is necessary to do during evaluation.

select statements

A *select* statement in SAPL is translated to a *switch* statement in JavaScript as follows:

$$\langle \text{select } f \ (\lambda x_0 \ldots \ x_n = b) \ldots \rangle$$

= 

```javascript
var _tmp = Sapl.feval(\[f\]);
switch(_tmp[0]) {
    case 0: var x0 = _tmp[2], \ldots, xn = _tmp[n+2];
            \[b\];
            break;
    \ldots
}
```

Evaluating the first argument of a *select* statement yields an array representing a constructor (see above). The first argument in this array represents the position of the constructor in its type definition, and is used to select the right case in the definition. The parameters of the $\lambda$- expression for each case are bound to the corresponding arguments of the constructor in the *var* declaration (see also examples).
Chapter 2

if statements

An if statement in SAPL is translated to an if statement in JavaScript:

```javascript
if (p) {
  t;
} else {
  f;
}
```

This translation works, because booleans in SAPL and JavaScript have the same representation.

Built-in functions

SAPL defines several built-in functions for arithmetic and logical operations. As an example, the `add` function is defined as follows:

```javascript
function add(x, y) {
  return Sapl.feval(x) + Sapl.feval(y);
}
```

Unlike user-defined functions, a built-in function such as `add` has strict evaluation semantics. To guarantee that their arguments are in normal form when the function is called, the function `Sapl.feval` must be applied to them (see Section 2.3.2).

2.3.1 Examples

The following definitions in SAPL:

```sapl
:: List = Nil | Cons x xs

ones = let os = Cons 1 os in os
fac n = if (eq n 0) 1 (mult n (fac (sub n 1)))
sum xxs = select xxs 0 (λx xs = add x (sum xs))
```

are translated to the following definitions in JavaScript:

```javascript
function Nil() {
  return [0, 'Nil'];
}
function Cons(x, xs) {
  return [1, 'Cons', x, xs];
}
function ones() {
  var os = Cons(1, function() {
    return os;
  });
  return os;
}
function fac(n) {
  if (Sapl.feval(n) == 0) {
    return 1;
  } else {
    return Sapl.feval([mult, [n, [fac, [[sub, [n, 1]]]]]]);
  }
}
```

20
Implementing a non-strict purely functional language in Java-Script

```javascript
function sum(as) {
  var _tmp = Sapl.feval(as);
  switch (_tmp[0]) {
    case 0: return 0;
    case 1: var x = _tmp[2], xs = _tmp[3];
           return Sapl.feval([add, [x, [sum, [xs]]]]);
  }
}
```

The examples show that the translation preserves the structure of the original definitions.

### 2.3.2 The `feval` function

To emulate SAPL’s non-strict evaluation semantics for function applications, we represented unevaluated expressions (thunks) as arrays in JavaScript. Because JavaScript treats these arrays as primitive values, some way is needed to explicitly reduce thunks to normal form when their value is required. This is the purpose of the `Sapl.feval` function. It reduces expressions to weak head normal form. Further evaluation of expressions is done by the printing routine. `Sapl.feval` performs a case analysis on an expression and undertakes different actions based on its type:

#### Literals

If the expression is a literal or a constructor, it is returned immediately. Literals and constructors are already in normal form.

#### Thunks

If the expression is a thunk of the form `[f, [xs]]`, it is transformed into a function call `f(xs)` with the JavaScript `apply` function, and `Sapl.feval` is applied recursively to the result (this is necessary because the result of a function call may be a boxed value).

Due to JavaScript’s reference semantics for arrays, thunks may become shared between expressions over the course of evaluation. To prevent the same thunk from being reduced twice, the result of the call is written back into the array. If this result is a primitive value, the array is transformed into a boxed value instead. Boxed values are represented as arrays of size one. Note that in JavaScript, the size of an array can be altered in-place.

If the number of arguments in the thunk is smaller than the arity of the function, it cannot be further reduced (is already in normal form), so it is returned immediately. Conversely, if the number of arguments in the thunk is larger than the arity of the function, a new thunk is constructed from the result of the call and the remainder of the arguments, and `Sapl.feval` is applied iteratively to the result.
Chapter 2

Boxed values

If the expression is a boxed value of the form \([x]\), the value \(x\) is unboxed and returned immediately (only literals and constructors can be boxed).

Curried applications

If the expression is a curried application of the form \([[f, [xs]], [ys]]\), it is transformed into \([f, [xs ++ ys]]\), and Sapl.feval is applied iteratively to the result.

More details on evaluation

For the sake of deeper understanding we also give the full source code of feval:

```javascript
feval = function (expr) {
    var y, f, xs;
    while (1) {
        if (typeof(expr) === "object") { // closure
            if (expr.length === 1) return expr[0]; // boxed value
            else if (typeof(expr[0]) === "function") { // application -> make call
                f = expr[0]; xs = expr[1];
                if (f.length === xs.length) { // most often occurring case
                    y = f.apply(null, xs); // turn chunk into call
                    expr[0] = y;
                    expr.length = 1; // adapt size
                } else if (f.length < xs.length) { // less likely case
                    y = f.apply(null, xs.splice(0, f.length));
                    expr[0] = y; // slice of arguments
                } else
                    return expr; // not enough arguments
            } else if (typeof(expr[0]) === "object") { // curried app -> uncurry
                y = expr[0];
                expr[0] = y[0];
                expr[1] = y[1].concat(expr[1]);
            } else
                return expr; // constructor
        } else if (typeof(expr) === "function") // function
            expr = [expr, []];
        else // literal
            return expr;
    }
}
```
2.3.3 Further optimizations

Above we described the compilation scheme from SAPL to JavaScript, where unevaluated expressions (thunks) are translated to arrays. The Sapl.feval function is used to reduce thunks to head normal form when their value is required. For ordinary function calls, our measurements indicate that the use of Sapl.feval is more than 10 times slower than doing the same call directly. This constitutes a significant overhead. Fortunately, a simple compile time analysis reveals many opportunities to eliminate unnecessary thunks in favor of such direct calls. Thus, expressions of the form:

Sapl.feval([f, [x1, ..., xn]])

are replaced by:

f(x1, ..., xn)

This substitution is only possible if f is a function with known arity at compile-time, and the number of arguments in the thunk is equal to the arity of the function. It can be performed wherever a call to Sapl.feval occurs:

- The first argument to a select or if;
- The arguments to a built-in function;
- Thunks that follow a return statement in JavaScript.

As an additional optimization, arithmetic operations are inlined wherever they occur. With these optimizations added, the earlier definitions of sum and fac are now translated to:

```javascript
function fac(n) {
    if (Sapl.feval(n) == 0) {
        return 1;
    } else {
        return Sapl.feval(n) * fac(Sapl.feval(n) - 1);
    }
}

function sum(as) {
    var _tmp = Sapl.feval(as);
    switch(_tmp[0]){
        case 0: return 0;
        case 1: var x = _tmp[2], xs = _tmp[3];
            return Sapl.feval(x) + sum(xs);
    }
}
```

As an additional optimization, arithmetic operations are inlined wherever they occur. With these optimizations added, the earlier definitions of sum and fac are now translated to:
Moreover, let’s consider the following definition of the Fibonacci function, \( \text{fib} \), in SAPL:

\[
\text{fib } n = \text{if } (\text{gt } 2 \text{ } n) \ 1 \ (\text{add } (\text{fib } (\text{sub } n \ 1)) \ (\text{fib } (\text{sub } n \ 2)))
\]

This is translated to the following function in JavaScript:

```javascript
function fib(n) {
    if (2 > Sapl.feval(n)) {
        return 1;
    } else {
        return (fib([n-1]) + fib([n-2]));
    }
}
```

A simple strictness analysis reveals that this definition can be turned into:

```javascript
function fib(n) {
    if (2 > n) {
        return 1;
    } else {
        return (fib(n - 1) + fib(n - 2));
    }
}
```

The calls to `feval` are now gone, which results in a huge improvement in performance. Indeed, this is how \( \text{fib} \) would have been written, had it been defined in JavaScript directly. In this particular example, the use of eager evaluation did not affect the semantics of the function. However, this is not true in general. For the use of such an optimization we adopted a Clean like strictness annotation. Thus, the above code can be generated from the following SAPL definition:

\[
\text{fib } !n = \text{if } (\text{gt } 2 \text{ } n) \ 1 \ (\text{add } (\text{fib } (\text{sub } n \ 1)) \ (\text{fib } (\text{sub } n \ 2)))
\]

But strictly defined arguments also have their price. In case one does not know if an argument in a function call is already in evaluated form, an additional wrapper function call is needed that has as only task to evaluate the strict arguments:

```javascript
function fib$eval(a0) {
    return fib(Sapl.feval(a0));
}
```

As a possible further improvement, a more thorough static analysis on the propagation of strict arguments could help to avoid some of these wrapper calls.

Finally, the SAPL to JavaScript compiler provides simple tail recursion optimization, which has impact on not only the execution time, but also reduces stack use.
The optimizations only affect the generated code and not the implementation of `feval`. In the next section an indication of the speed-up obtained by the optimizations is given.

### 2.4 Benchmarks

In this section we present the results of several benchmark tests for the JavaScript implementation of SAPL (which we will call Sapljs) and a comparison with the Java Applet implementation of SAPL. We ran the benchmarks on a MacBook 2.26 MHz Core 2 Duo machine running MacOS X10.6.4. We used Google Chrome with the V8 JavaScript engine to run the programs. At this moment V8 offers one of the fastest platforms for running Sapljs programs. However, there is a heavy competition on JavaScript engines and they tend to become much faster. The benchmark programs we used for the comparison are the same as the benchmarks we used for comparing SAPL with other interpreters and compilers in [79]. In that comparison it turned out that SAPL is at least twice as fast (and often even faster) as other interpreters like Helium[70], Amanda[22], GHCi and Hugs[81]. Here we used the Java Applet version for the comparison. This version is about 40% slower than the C version of the interpreter described in [79] (varying from 25 to 50% between benchmarks), but is still faster than the other interpreters mentioned above. The Java Applet and JavaScript version of SAPL and all benchmark code can be found at [40].

We briefly repeat the description of the benchmark programs here:

1. **Prime Sieve** The prime number sieve program, calculating the 2000th prime number.
2. **Symbolic Primes** Symbolic prime number sieve using Peano numbers, calculating the 160th prime number.
3. **Interpreter** A small SAPL interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 30th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating $\text{fib}\ 35$.
5. **Match** Nested pattern matching (5 levels deep) repeated 160000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 1000 times.
7. **Sorting** Tree Sort (3000 elements), Insertion Sort (3000 elements), Quick Sort (3000 elements), Merge Sort (10000 elements, merge sort is much faster, we therefore use a larger example)
8. **Queens** Number of placements of 11 Queens on a 11 x 11 chess board.
9. **Knights** Finding a Knights tour on a 5 x 5 chess board.

10. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree, repeated 100 times.

11. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 3500 lines Prolog program.

For sorting a list of size $n$ a source list is used consisting of numbers 1 to $n$. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

The benchmarks cover a wide range of aspects of functional programming: lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage. The programs were chosen to run at least for a second, if possible. This helps eliminating start-up effects and gives the JIT compiler enough time to do its work. In many cases the output was converted to a single number (e.g. by summing the elements of a list) to eliminate the influence of slow output routines.

### 2.4.1 Benchmark tests

We ran the tests for the following versions of SAPL:

- SAPL: the Java Applet version of SAPL;

- Sapljs: the Sapljs version including the normal form optimization, the inlining of arithmetic operations and the tail recursion optimization. The strictness optimization is only used for the fib benchmark;

- Sapljs nopt: the version not using these optimizations.

We also included the estimated percentage of time spent on memory management for the Sapljs version. The results can be found in Figure 2.1.
2.4.2 Evaluation of the benchmark tests

Before analysing the results we first make some general remarks about the performance of Java, JavaScript and the SAPL interpreter which are relevant for a better understanding of the results. In general it is difficult to give absolute figures when comparing the speeds of language implementations. They often also depend on the platform (processor), the operating system running on it and the particular benchmarks used to compare. Therefore, all numbers given should be interpreted as global indications.

According to the language shoot-out site [27] Java programs run between 3 and 5 times faster than similar JavaScript programs running on V8. So a reimplementaton of the SAPL interpreter in JavaScript is expected to run much slower as the SAPL interpreter.

We could not run all benchmarks as long as we wished because of stack limitations for V8 JavaScript in Google Chrome. It supports a standard (not user modifiable) stack of only 30k at this moment. This is certainly enough for most JavaScript programs, but not for a number of our benchmarks that can be deeply recursive. This limited the size of the runs of the following benchmarks: Interpreter\(^1\) all sorting benchmarks, and the Prolog and Parser Combinator benchmark. Another benchmark that we used previously, and that could not be ran at all in Sapljs is: twice twice twice inc 0.

For a lazy functional language the creation of thunks and the re-collection of them later on, often takes a substantial part of program run-times. It is therefore important to do some special tests that say something about the speed of memory (de-)allocation. The SAPL interpreter uses a dedicated memory management unit (see [79]) not depending on Java memory management. The better performance of the SAPL interpreter in comparison with the other interpreters partly depends on its fast memory management. For the JavaScript implementation we rely on the memory management of JavaScript itself. We did some dedicated tests that showed that memory allocation for the Java SAPL interpreter is about 5-7 times faster than the JavaScript implementation. Therefore, we included an estimation of the percentage of time spent on memory management for all benchmarks ran in Sapljs. The estimation was done by counting all memory allocations for a benchmark (all creations of thunks) and multiplying it with an estimation of the time to create a thunk, which was measured by a special application that only creates thunks.

Results

The Fibonacci and Interpreter benchmarks run (30 and 2 times resp.) significantly faster in Sapljs than in the SAPL interpreter. Note that both these

\(^1\)The latest version of Chrome has an even more restricted stack size. We can now run Interpreter only up to the 18th prime number.
benchmarks profit significantly from the optimizations with Fibonacci being more than 100 times faster and Interpreter almost 7 times faster than the non-optimized version. The addition of the strictness annotation for Fibonacci contributes a factor of 3 to the speed-up. With this annotation the compiled Fibonacci program is equivalent to a direct implementation of Fibonacci in JavaScript and does not use feval anymore. The original Sapl interpreter does not apply any of these optimizations. The Interpreter benchmark profits much (almost a factor of 2) from the tail recursion optimization that applies for a number of often used functions that dominate the performance of this benchmark.

Symbolic Primes, Match, Queens and Knights run at a speed comparable to the Sapl interpreter. Hamming and Sort are 40 percent slower, Primes and Prolog are 80 percent slower. Parser Combinators is the worst performing benchmark and is almost 4 times slower than in Sapl.

All benchmarks benefit considerably from the optimizations (between 1.5 and 120 times faster), with Fibonacci as the most exceptional.

The Parser Combinators benchmark profits only modestly from the optimizations and spends relatively much time in memory management operations. It is also the most ‘higher order’ benchmark of all. Note that for the original Sapl interpreter this is one of the best performing benchmarks (see [79]), performing at a speed that is even competitive with compiler implementations. The original Sapl interpreter does an exceptionally good job on higher order functions.

We conclude that the Sapljs implementation offers a performance that is competitive with that of the Sapl interpreter and therefore with other interpreters for lazy functional programming languages.

Previously [79] we also compared Sapl with the GHC and Clean compilers. It was shown that the C version of the Sapl interpreter is about 3 times slower than GHC without optimizer. Extrapolating this result using the figures mentioned above we conclude that Sapljs is about 6-7 times slower than GHC (without optimizer). In this comparison we should also take into account that JavaScript applications run at least 5 times slower than comparable C applications. The remaining difference can be mainly attributed to the high price for memory operations in Sapljs.

2.4.3 Alternative memory management?

For many Sapljs examples a substantial part of their run-time is spent on memory management. They can only run significantly faster after a more efficient memory management is realized or after other optimizations are realized. It is tempting to implement a memory management similar to that of the Sapl interpreter. But this memory management relies heavily on representing graphs by binary trees, which does not fit with our model for turning
thunks into JavaScript function calls which depends heavily on using arrays to represent thunks.

2.5 Applications

Developing rich client-side applications in Clean

We can use the Sapljs compiler to create dedicated client-side applications in Clean that make use of JavaScript libraries. We can do this because JavaScript and the code generated by Sapljs share the same namespace. In this way it is possible to call functions within SAPL programs that are implemented in JavaScript. The Sapljs compiler does not check the availability of a function, so one has to rely on the JavaScript interpreter to do this. Examples of such functions are the built-in core functions like add and eq, but they can be any application related predefined function.

Because we have to compile from Clean to SAPL before compiling to JavaScript, we need a way to use functions implemented in JavaScript within Clean programs. Clean does not allow that programs contain unknown functions, so we need a way to make these functions known to the Clean compiler. This can be realized by defining holes, unspecified functions, using the undef function provided by the standard Clean environment. An undef expression matches any type, so it can be used to check if the written code is syntactically and type correct, without finishing the whole application. E.g., example is a function with 2 integer arguments and an integer result with an implementation only in JavaScript.

\[
\text{example :: Int Int } \rightarrow \text{ Int}
\]
\[
\text{example } = \text{ undef}
\]

We adapted the Clean to SAPL compiler not to generate code for functions with an undefined body. In this way we have created a universal method to reference functions defined outside the Clean environment.

We used this technique to define a library in Clean for manipulating the HTML DOM at the client side. The following Clean code gives a demonstration of its use:

\[
\text{import StdEnv, SaplHtml}
\]
\[
\text{onKeyUp :: !HtmlEvent !*HtmlDocument } \rightarrow \text{ */(HtmlDocument, Bool)}
\]
\[
\text{onKeyUp e d}
\]
\[
\# (d, str) = \text{getDomAttr d "textarea" "value"}
\]
\[
\# (d, str) = \text{setDomAttr d "counter" "innerHTML" (toString (size str))}
\]
\[
= (d, True)
\]
It is basically a definition of a piece of HTML using arrays and ADTs defined in the SaplHtml module. What is worth to notice here are the definitions of the event handler function and the DOM manipulating functions, `getDomAttr` and `setDomAttr`, which are also defined in SaplHtml, but are implemented in JavaScript using the above mentioned technique. The two parameters of the event handler function are effectively the related JavaScript Event and Document objects, respectively.

Compiling the program to JavaScript and running it returns the following string, which is legal HTML:

```html
<div><textarea id="textarea" rows="15" cols="50"
  onKeyUp="Sapl.execEvent ( event , 'onKeyUp$eval')">
  
</textarea>
</div>

The event handler call is wrapped by the `Sapl.execEvent` function which is responsible for passing the event related parameters to the actual event handler. Including this string into an HTML document along with the generated JavaScript functions we get a client side web application originally written in Clean. Despite this program is deliberately very simple, it demonstrates almost all the basics necessary to write any client side application. Additional interface functions, e.g. calling methods of a JavaScript object, can be found in the SaplHtml module.

**iTTask integration**

Another possible application is related to the iTTask system [111]. iTask is a combinator library written in Clean, and is used for the realization of web-based dynamic workflow systems. An iTask application consists of a structured collection of tasks to be performed by users, computers or both.

To enhance the performance of iTTask applications, the possibility to handle tasks on the client was added [112], accomplished by the addition of a simple `OnClient` annotation to a task. When this annotation is present, the iTTask runtime automatically takes care of all communication between the client and server parts of the application. The client part is executed by the SAPL interpreter which is available as a Java applet on the client.

However, the approachability of JavaScript is much better compared to Java. The Java runtime environment, the Java Virtual Machine might not...
even be available on certain platforms (on mobile devices in particular). Besides that, it exhibits significant latency during start-up. For these reasons, a new implementation of this feature is recommended using Sapljs instead of the SAPL interpreter written in Java. Several feature were made to foster this modification:

- The SAPL language was extended with some syntactic sugar to allow distinguishing between constructors and records.

- Automatic conversion of data types like records, arrays, etc, between SAPL and JavaScript was added. In this way full interaction between SAPL and existing libraries in JavaScript became possible.

- Automatic conversion of JSON data structures to enable direct interfacing with all kinds of web-services was added.

2.6 Related work

Client-side processing for Internet applications is a subject that has drawn much attention in the last years with the advent of Ajax based applications. Earlier approaches using JavaScript as a client-side platform for the execution of functional programming languages are Hop [96, 116], Links [28] and Curry [69].

Hop is a dedicated web programming language with a HTML-like syntax build on top of Scheme. It uses two compilers, one for compiling the server-side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. The application essentially runs on the client and may call services on the server. Syntactic constructions are used for indicating client and server part code. In [96] it is shown that a reasonably good performance for client-side functions in Hop can be obtained. However, contrary to Haskell and Clean, both Hop and the below mentioned Links are strict functional languages, which simplifies their translation to JavaScript considerably.

Links [28] and its extension Formlets is a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. Client-server communication is implemented using Ajax technology, like this is done in the iTask system.

Curry offers a much more restricted approach: only a very restricted subset of the functional-logic language Curry is translated to JavaScript to handle client-side verification code fragments only.

A more recent approach is the Flapjax language [103], an implementation of functional reactive programming in JavaScript. Flapjax can be used either
as a programming language, compiling to JavaScript, or as a JavaScript library. Entire applications can be developed in Flapjax. Flapjax automatically tracks dependencies and propagates updates along dataflows, allowing for a declarative style of programming.

An approach to compile Haskell to JavaScript is YCR2JS [65] that compiles YHC Core to JavaScript, comparable to our approach compiling SAPL to JavaScript. Unfortunately, we could not find any performance figures for this implementation.

Another, more recent approach, for compiling Haskell to JavaScript is HS2JS [76], which integrates a JavaScript backend into the GHC compiler. A comparison of JavaScript programs generated by this implementation indicate that they run significantly slower than their Sapljs counterparts.

### 2.7 Conclusion and future work

In this paper we evaluated the use of JavaScript as a target language for lazy functional programming languages like Haskell or Clean using the intermediate language SAPL. The implementation has the following characteristics:

- It achieves a speed for compiled benchmarks that is competitive with that of the SAPL interpreter and is faster than interpreters like Amanda, Helium, Hugs and GHCi. This is despite the fact that JavaScript has a 3-5 times slower execution speed than the platforms used to implement these interpreters.

- The execution time of benchmarks is often dominated by memory operations. But in many cases this overhead could be significantly reduced by a simple optimization on the creation of thunks.

- The implementation tries to map SAPL to corresponding JavaScript constructs as much as possible. Only when the lazy semantics of SAPL requires this, an alternative translation is made. This opens the way for additional optimizations based on compile time analysis of programs.

- The implementation supports the full Clean (and Haskell) language, but not all libraries are supported. We tested the implementation against a large number of Clean programs compiled with the Clean to SAPL compiler.

#### 2.7.1 Future work

We have planned the following future work:
- Implement a web-based Clean to SAPL (or to JavaScript) compiler (experimental version already made).

- Experimenting with supercompilation optimization by implementing a SAPL to SAPL compiler based on whole program analysis.

- Encapsulate JavaScript libraries in a functional way, e.g. using generic programming techniques.

- Attach client-side call-backs written in Clean to iTask editors. It can be implemented using Clean-SAPL dynamics [80] which make it possible to serialize expressions at the server side and execute them at the client side.

- Use JavaScript currying instead of building thunks. Our preliminary results indicate that using JavaScript currying would be significantly slower, but further investigation is needed for proper analysis.
3 Compiling Haskell to JavaScript through Clean’s core

Crossing the borders of languages by letting them cooperate on source code level has enormous benefits as different languages have distinct language features and useful libraries to share. This is particularly true for the functional programming world where languages are in constant development being the target of active research. There already exists a double-edged compiler frontend for the lazy functional languages Haskell and Clean which enables the interoperation of features of both languages. This paper presents a series of program transformations to solve the same problem at another level by transforming STG, one of the core languages of the flagship Haskell compiler GHC, to SAPL, one of the core languages of Clean. By this transformation (1) we have made the existing Clean to JavaScript compiler available for Haskell as well; (2) using this compiler, under certain limitations, one can mix Clean and Haskell code as they are compiled to the same target code using the same run-time system and calling convention; (3) one can more easily compare the core code generated by the two compilers and measure their execution times.

3.1 Introduction

Functional languages tend to make use of one or more core languages, usually a kind of enriched lambda calculus, that are generated at different stages of their compiler pipeline [83, 54, 13, 84]. Usually, the last one of the pipeline serves as the source language of the code generator, while the others represent an intermediate transformation. Many times these languages are also exploited as the source of compilation to alternative target platforms, e.g. SAPL to JavaScript [39], or as the input of some tool, e.g. a theorem prover [105].

The latter suggests a further natural use case: by transforming a core of one language into the core of another language, all the compilers and tools associated with the target core would be immediately available for the source language. Transforming core languages also has the advantage that the program being transformed is already type checked, and complex syntax is factored out. This idea is accomplished at a different level of abstraction by van Groningen et al as they made a double-edged compiler frontend for Clean to be able to compile Haskell code as well [126]. In that project, however, the primary goal
was to provide interoperability of the language features and libraries for both languages and was limited to Haskell98 code, which comprises only a limited feature set.

In this paper we investigate the translation of STG to SAPL. We chose STG [68], one of the core languages of the flagship Haskell compiler GHC, as the source language of the transformation, because of the many interesting language features of GHC. Strictly speaking the STG language is not the core language of the GHC compiler. There exists a *GHC Core* which is a very small, explicitly-typed language. However, STG is the last step of a series of intermediate representations (it is the language of the so called Spineless *Tagless G*-machine (STG) to which GHC compiles), thus it benefits from all the optimizations of the GHC compiler. Our target language is SAPL, one of the core languages of Clean that is the platform of an efficient interpreter technology [79] and JavaScript compiler [39].

Our primary goal is to develop a reliable transformation technique which enables mixing SAPL code of different sources under minimal restrictions. In Figure 3.1 an overview of the Clean and Haskell compilers is presented. It shows how the SAPL language fits into the picture: both compiler frontends
Compiling Haskell to JavaScript through Clean’s core

have been modified to convert their cores (internal data structures) into SAPL. Later on SAPL, coming from both sources, can be compiled to JavaScript, so the same run-time system and calling convention is used for both languages, thereby the foundations of interoperability have been made.

The remainder of this paper is structured as follows: in Section 3.2 and Section 3.3 the SAPL and STG languages are presented briefly. After some general remarks on the soundness of the transformation in Section 3.4, we introduce the transformation steps in Section 3.5. In Section 3.6 interoperability issues are discussed, then benchmarks are presented in Section 3.7. It is followed by a discussion of related work in Section 3.8 and concluding remarks in Section 3.9.

3.2 Introduction into SAPL

SAPL stands for Simple Application Programming Language. This is the target of the transformation, a core, lazy, purely-functional language. In the following, some basic examples are presented to give an intuition about the language. Consider the Clean code of the following factorial and summation functions:

\[
\begin{align*}
\text{fact} & \ 0 = 1 \\
\text{fact} & \ n = n \times \text{fact}(n-1) \\
:: \ \text{List} \ a & = \text{Nil} \ | \ \text{Cons} \ a \ (\text{List} \ a)
\end{align*}
\]

\[
\begin{align*}
\text{sum} \ \text{Nil} & = 0 \\
\text{sum} \ (\text{Cons} \ x \ xs) & = x + \text{sum} \ xs
\end{align*}
\]

The equivalent function definitions in SAPL are the following:

\[
\begin{align*}
\text{fact} & \ n = \text{if} \ (\text{eq} \ n \ 0) \ 1 \ (\text{mult} \ n \ (\text{fact} \ (\text{sub} \ n \ 1))) \\
:: & \ \text{Nil} \ | \ \text{Cons} \ x \ xs \\
\text{sum} \ xxss & = \text{select} \ xxss \ 0 \ (\lambda \ x \ xs = x + \text{sum} \ xs)
\end{align*}
\]

The if and select constructs are used to pattern match on primitive and algebraic types, respectively. The language also requires the declaration of data constructors and their arities (but not the type) starting with the :: token.

Finally, the following mappair function written in Clean is a more complex example which also uses pattern matching:

\[
\begin{align*}
:: & \ \text{List} \ a = \text{Nil} \ | \ \text{Cons} \ a \ (\text{List} \ a) \\
\text{mappair} \ f \ \text{Nil} \ ys & = \text{Nil} \\
\text{mappair} \ f \ (\text{Cons} \ x \ xs) \ \text{Nil} & = \text{Nil} \\
\text{mappair} \ f \ (\text{Cons} \ x \ xs) \ (\text{Cons} \ y \ ys) & = \text{Cons} \ (f \ x \ y) \ (\text{mappair} \ f \ xs \ ys)
\end{align*}
\]

This definition is transformed to the following SAPL function:
:: Nil | Cons x xs
mappair f xs ys
= select xs Nil (λx xs =
select ys Nil (λy ys = Cons (f x y) (mappair f xs ys))

SAPL is obtained from Clean by removing type information and syntactic sugar (this is done after the type checking phase of the compilation, SAPL programs generated from Clean are type correct). This includes the contraction of partial functions by the means of SAPL pattern matching contracts if and select, as it is illustrated in these examples.

SAPL was originally constructed as a language of its own implementing language features to enable efficient interpretation [79]. Later, a Clean like type definition style and other language features were adopted for readability and to allow for the generation of efficient JavaScript code [39]. Currently SAPL source code can be generated using the Clean compiler, thus, considering the intermediate nature of SAPL, it can be regarded as a core language of Clean.

The formal definition of the language is given in [39]. In this chapter we restrict ourselves to a somewhat simplified version of the SAPL syntax in Figure 3.2 (e.g. for readability, syntax for parentheses is omitted):

\[
\begin{align*}
d &::= :: C_1 [\vec{v}_1] | \ldots | C_n [\vec{v}_n] & \text{(Algebraic Type Definition)} \\
f &::= v [\vec{v}] = e & \text{(Function Definition)} \\
& \mid v := e & \text{(Constant Applicative Form)} \\
e &::= \text{select } s \vec{l} & \text{(Select Expression)} \\
& \mid \text{let } \vec{b} \text{ in } s & \text{(Local Definition)} \\
& \mid s & \text{(Simple Expression)} \\
b &::= v = s & \text{(Binding)} \\
s &::= \text{if } s_c s_t s_f & \text{(If Expression)} \\
& \mid v [\vec{s}] & \text{(Application)} \\
& \mid C [\vec{s}] & \text{(Constructor)} \\
& \mid L & \text{(Literal)} \\
l &::= \lambda \vec{v} = s & \text{(Lambda Expression)} \\
& \mid s & \text{(Simple Expression)}
\end{align*}
\]

Figure 3.2: Core syntax of SAPL.

**Select Expressions** Select expressions, inspired by Church encoding [87], are intended to perform pattern matching on data constructors. The
select keyword is used to make a case analysis on the data type of its first argument. To accomplish this, the first argument is reduced to *head normal form* before pattern matching. The remaining arguments handle the different constructor cases in the same order as they occur in the type definition (all cases must be handled separately). Each case is a function that is applied to the arguments of the corresponding constructor. If for a particular data constructors a case alternative is not provided, the primitive `nomatch` function must be used to trigger a run-time error if necessary.

**If Expressions** An *if* expression is used to perform case analysis on a primitive value. Just like the *Select* expression, it is strict in its first argument, in its *predicate*. It returns its second or third argument depending on the value of the predicate after reduction.

**Lambda Expressions** Only the arguments of a *select* expression can contain lambda expressions in SAPl. All other nested lambda expression must be lifted to the top-level.

**Let Expressions** Only constant (non-function) *let* expressions are allowed that may be mutually recursive (for creating cyclic expressions).

**Applications** SAPl, in contrast to STG, does not require applications to be saturated, and arguments of applications are allowed to be other applications (compound applications).

**Function Definitions** Function definitions can be split into two kinds. SAPl distinguishes normal functions and *constant applicative forms* (CAF). CAFs cannot have any arguments and the `':='` notation is used for them.

**Algebraic Type Definitions** SAPl requires explicit algebraic type definitions. The constructor indexes are used by the *select* construct, while the arity of data constructors carries important information for the code generator.

### 3.3 Introduction into the STG language

The *Shared Term Graph* (STG) language, one of the core languages of GHC, the language of the *Spineless Tagless G*-machine, is the source of the transformation. To have an intuition of the language, let us consider the source code of the summation function of the previous section. In Figure 3.4, the STG counterpart of the factorial example is also given.
Chapter 3

\[
f ::= b \quad \text{(Function Definition)}
\]

\[
b ::= v = l \quad \text{(Binding)}
\]

\[
l ::= [\vec{v}] \lambda \pi [\vec{v}_a] \rightarrow e \quad \text{(Lambda Form)}
\]

\[
\pi ::= u | n \quad \text{(Update Flag)}
\]

\[
e ::= \text{let } \vec{b} \text{ in } e \quad \text{(Local Definition)}
| \text{letrec } \vec{b} \text{ in } e \quad \text{(Local Recursion)}
| \text{case } e \text{ of } A \quad \text{(Case Expression)}
| v [\vec{a}] \quad \text{(Application)}
| C [\vec{a}] \quad \text{(Constructor)}
| L \quad \text{(Literal)}
\]

\[
a ::= v \quad \text{(Variable)}
| L \quad \text{(Literal)}
\]

\[
A ::= \vec{A}_a [D] \quad \text{(Algebraic Alternatives)}
| \vec{A}_p [D] \quad \text{(Primitive Alternatives)}
| D \quad \text{(Default Only Alternative)}
\]

\[
A_a ::= C [\vec{v}] \rightarrow e \quad \text{(Algebraic Alternative)}
\]

\[
A_p ::= L \rightarrow e \quad \text{(Primitive Alternative)}
\]

\[
D ::= v \rightarrow e \quad \text{(Default Alternative)}
| \text{default } \rightarrow e
\]

---

**Figure 3.3:** Core syntax of STG.

\[
\text{Main.sum =}
\{\} \lambda n \{\text{ds_sL7}\} \rightarrow
\text{case ds_sL7 of}
\text{Main.Nil }\{\} \rightarrow \text{GHC.Types.I# }\{0\}
\text{Main.Cons }\{x_s\_lb, x_s\_Lc\} \rightarrow
\text{let sat_sMu }= \{x_s\_Lc\} \lambda u \{\} \rightarrow \text{Main.sum x_s\_Lc}
\text{in } \text{GHC.Num.}+ \{\text{GHC.Num.fNumInt}, x_s\_lb, sat_sMu\}
\]

Conceptually, the STG language is an enriched lambda calculus, furthermore, it can be regarded as a variant of administrative normal form (ANF) [57] as (1) it allows only constants and variables to serve as arguments of function applications (flat applications), and (2) it requires the result of a non-trivial expression to be assigned to a let-bound variable or returned from a function.

To highlight the essence of the language, in Figure 3.3, a slightly simplified,
Compiling Haskell to JavaScript through Clean’s core

Main.fact = \{\} \lambda \{ds_sCK\} \rightarrow 
  \textit{case } ds_sCK \textit{ of} 
    \textit{GHC.Types.I#} \{dsl_sCN\} \rightarrow 
      \textit{case } dsl_sCN \textit{ of} 
        \textit{default } \rightarrow 
          \textit{let } sat_sDc = \{ds_sCK\} \lambda \{\} \rightarrow 
            \textit{let } sat_sDd = \{ds_sCK\} \lambda \{\} \rightarrow 
                \textit{in } GHC.Types.I# \{1\} 
          \textit{in } GHC.Num.* \{GHC.Num.fNumInt, ds_sCK, sat_sDc\} 
          0 \rightarrow GHC.Types.I# \{1\} 

Figure 3.4: The STG code of the factorial function generated by GHC version 7.2.1 (no optimizations).

core syntax of the original STG definition [68] is presented; newer GHC versions tend to generate rather different and extended syntax instead. All the STG examples in this chapter are generated by GHC 7.2.1 and then converted to the equivalent original syntax for better understanding. In the following we highlight the salient characteristics of the language, the full syntax along with well-defined operational semantics is given in [68].

Case Expressions Case expressions are used to perform pattern matching. In contrast to SAPL, it handles both algebraic and primitive data types and this is the only construct where a data value is ever forced (evaluated) in STG. Case alternatives may contain a default branch as can be seen in Figure 3.4, line 6.

Let Expressions Let expressions begin with either the \textit{let} or \textit{letrec} keywords. Let expressions can be mutually recursive and always bind variables to constructs resulting in a \textit{closure} in the heap: lambda abstractions, constructor applications or other \textit{let} expressions.

Lambda expressions Lambda expressions contain special hints for the code generator, the so called update flag. This flag is the first character (denoted by $\pi$ in Figure 3.3) after the backslash which indicates a lambda expression. Before the update flag, a list of free variables are given; the arguments of lambda abstractions are listed after the update flag. If there are no arguments, that lambda expression simply denotes a \textit{thunk}, a suspended computation. For further information about the connection of lambdas, thunks and update flags please refer to [68].
Applications  STG, in accordance with ANF, requires applications to be saturated, and the arguments of applications must be atoms (variables or literals), compound applications are not allowed.

Function Definitions  Function definitions are named lambda expressions in STG, that is the \( f = \{ \} \lambda \pi \{ a_1 a_2 \ldots a_n \} \rightarrow \) body form is used instead of the \( f \ a_1 \ a_2 \ldots \ a_n = \) body style.

3.4 Discussion

Before explaining the actual transformation steps, it is important to discuss the limitations of the transformation. SAPL has a definite call-by-need evaluation strategy, basically all closures (thunks and values) are created implicitly at let bindings and function arguments, and they are forced at pattern matching (in the if and select expressions) and in primitive functions.

STG takes a different approach. Closures are always created explicitly at let bindings, and the only place they are ever forced is in the case expression. Primitive values can be passed to functions unboxed and primitive operations actually expect their arguments in that form (primitive values are boxed by wrapping them in a data constructor, this is how strictnes is handled in STG, see Section 3.7). That is, STG has a finer control over what and when is forced than SAPL.

As a result, the transformed SAPL program may be lazier than the original STG one, thus we must declare that the transformation is valid only in an effect-free environment. The impact of the transformations on non-pure functions is out of the scope of this chapter, and requires further investigation.

3.5 Transforming STG to SAPL

In this section we present a series of program transformations which, by applying them on an STG program, result in SAPL code. We use the same simple factorial function to illustrate the steps as before:

\[
\text{fact} :: \text{Int} \rightarrow \text{Int} \\
\text{fact} \ 0 = 1 \\
\text{fact} \ n = n \times \text{fact} \ (n - 1)
\]

We use the STG code of this function from Figure 3.4. First, the abstract syntax tree (AST) of the SGT program is produced and the following rewrite rules are applied step by step in the given order. This technique is similar to compilation by program transformation [86, 85], but the code resulted from the intermediate steps is a mixture of the source and target languages.
Transformation of named lambda expressions

Function definitions are named lambda expressions in STG which can be easily turn into SAPL function definitions by the following rewrite rule (update flags are ignored):

\[ f = \{ \} \lambda \pi \{ x_1 \ldots x_n \} = b \Rightarrow f \ x_1 \ldots x_n = b \]

This rule becomes slightly different in the case of an empty argument list, because according to the semantics of Haskell, top-level functions with an empty argument list are just CAFs:

\[ f = \{ \} \lambda \pi \{ \} = b \Rightarrow f := b \]

Extraction of algebraic type definitions

In SAPL algebraic data type definitions must be explicitly provided. In contrast, STG code does not contain explicit type definitions, but a pre-generated function for every data constructor. Fortunately, even if this information is not part of the STG language directly, it is available in the compiler. This is the option we chose.

A pure STG solution would be to discover groups of constructors by defining an equivalence relation. We can say that two constructors are in the same group (have the same type) iff they are both alternatives of the same `case` expression. Computing the transitive closure of the so gained groups we eventually would get a valid grouping of data constructors.

Elimination of forcing-only `case` expressions

Since only `case` expressions force evaluation in STG, sometimes, instead of pattern matching, it is used to explicitly force the evaluation of a thunk. When this is so, only a `default case` exists. In SAPL there is no way to explicitly force a thunk, these expressions must be removed from the AST:

\[ \text{case } x \text{ of default } \rightarrow e \Rightarrow e \]

By this transformation the evaluation of the given expression is delayed, but, as long as we regard pure functions only, it is certainly safe by the considerations of Section 3.4.

Lambda lifting

SAPL allows lambda expressions as the arguments of a `select` statement only. STG, however, encodes local functions as lambda expressions occurring in `let` bindings. In this step these `let` bindings are lifted to the top-level as new function definitions:
Chapter 3

\[
\begin{align*}
\text{let } b_1, \ldots, f = \{FV_1 \ldots FV_m\} & \lambda\pi \{x_1 \ldots x_n\} = e_1, \ldots, b_n \text{ in } e_2 \\
\Rightarrow \text{let } b_1, \ldots, b_n \text{ in } e_2[f := f'FV_1 \ldots FV_m],
\end{align*}
\]

where \( f' \) is a new, uniquely generated function name, \( FV_i \) is the \( i \)th local free variable (that is not free in the original function) of the expression assigned to \( f \) and \( m \) denotes the number of such free variables. The definition of the \( f' \) function is as follows:

\[
f'FV_1 \ldots FV_m x_1 \ldots x_n = e_1
\]

**Application inlining**

The STG machine requires function and constructor arguments to be atoms (variables or constants). This constraint implies that all sub-expressions are explicitly named and the evaluation order is explicit which is well suited to the pursuit of making the code generator as simple as possible. However, some research indicates that if the bodies of functions or \texttt{let} bindings are mostly small (which is the consequence of flat applications), the interpretation overhead is relatively large [79].

In this step we invert the decomposition of non-flat applications which is done during the translation of GHC Core into STG. That time new \texttt{let} bindings were added for the non-trivial arguments of applications; now \texttt{let}-bound variables are inlined. However, we have to be careful. To preserve the sharing property of thunks, only those \texttt{let} bindings can be inlined which occur (1) only once and (2) only as the argument of an application. Furthermore, a \texttt{let} can be inlined only if it is not mutually recursive. When these conditions are satisfied, the rewrite rule is the following:

\[
\text{let } b_1, \ldots, x = e', \ldots, b_n \text{ in } e \Rightarrow \text{let } b_1, \ldots, b_n \text{ in } e\{x := e'\}
\]

Subsequently, \texttt{let} expressions with an empty binding list must be removed and the transformation step must be repeated until there is no more \texttt{let} binding which satisfies the necessary conditions. After three iterations of this rule, our example looks rather concise (the long line of embedded applications is broken into multiple lines) as can be seen in Figure 3.5.

**Lifting of \texttt{let} bindings**

SAPL does not allow \texttt{let} expressions to nest other \texttt{let} expressions, neither in the bindings nor in the body. In the previous step we eliminated some of them by inlining \texttt{let} bindings when they were applications. The remaining of such \texttt{let} expressions, unfortunately, cannot be managed so easily, some bindings must be lifted as a top-level function. These are the cases when a binding contains another \texttt{let}, or a \texttt{case} expression. The following rewrite rule uses the notations introduced at lambda lifting:
Main.fact ds_sCK =
  case ds_sCK of
    GHC.Types.I# \{ds1_sCN\} →
      case ds1_sCN of
        default →
          GHC.Num.* \{GHC.Num.fNumInt ds_sCK
            (Main.fact
            \{(GHC.Num.- \{GHC.Num.fNumInt ds_sCK (GHC.Types.I# \{1\})\}\})\}
          0 → GHC.Types.I# \{1\}

Figure 3.5: The intermediate code after inlining applications.

\[
\text{let } b_1, \ldots, f = e_1, \ldots, b_n \text{ in } e_2
\Rightarrow \text{let } b_1, \ldots, b_n \text{ in } e_2[f := f' FV_1 \ldots FV_m],
\]
where \(e_1\) is a case or let expression and the definition of \(f'\) is as follows:

\[
f' \ FV_1 \ldots FV_m = e_1
\]

This step introduces new applications. To ensure that these are also inlined if possible, a fixed-point iteration, consisting of this and the previous step, is utilized.

**Fusion of nested let expressions**

So far, we eliminated let expressions occurring in let bindings and inlined most of the nested ones. However, to preserve call-by-need semantics, bindings are allowed to be inlined when they occur only once in the nested body, therefore the intermediate code still can contain nested let expressions. To avoid them, the bindings of the nested expression must be lifted and merged with the bindings of its container. In this case, mutually recursive let bindings can be moved safely, because they will not be broken up.

\[
\text{let } b_1, \ldots, b_n \text{ in let } d_1, \ldots, d_m \text{ in } e
\Rightarrow \text{let } b_1, \ldots, b_n, d_1, \ldots, d_m \text{ in } e
\]

This transformation finally results in an intermediate code which does not contain nested let expressions.

**Transformation of pattern matching logic**

Pattern matching is performed by case expressions in STG. In this step, the two flavors of case expressions, pattern matching of algebraic and primitive
types, are separated and translated into SAPL select and if expressions, respectively. As both the select and if expressions are strict in their first argument, that is force evaluation, this transformation preserves the semantics of the original program. The type of a given case expression can be clearly recognized by the analysis of the left hand side of the case alternatives: if any of them is literal, we have to convert it to an if expression, otherwise we deal with the algebraic type.

- Pattern matching on primitive types is converted to SAPL if expressions. These expressions have three arguments in this order: (1) predicate (2) true-branch and (3) false- or else-branch. The algorithm is the following: the case alternatives are converted to nested if expressions, so we have to determine first the else-branch of the innermost if expression. It will be the default case alternative or the last case alternative if a default case does not exist. Then the remaining alternatives, from top to bottom, are converted to if expressions: every such expression will be the else-branch of the previous one. The predicates are translated into an application of the primitive eq function.

\[
\begin{align*}
\text{case } x \text{ of} \\
l_1 & \rightarrow e_1 \Rightarrow \text{if } (\text{eq } x \ l_1) \ e_1 \\
l_2 & \rightarrow e_2 \quad \text{(if } (\text{eq } x \ l_2) \ e_2 \\
\vdots \\
l_{n-1} & \rightarrow e_{n-1} \quad \text{(if } (\text{eq } x \ l_{n-1}) \ e_{n-1} \ e_n) \ldots \\
l_n & \rightarrow e_n
\end{align*}
\]

- Algebraic data constructors are already discovered by the previous steps, thus we can easily determine the constructor indexes. This information is vital as it defines the order of the arguments of the select statement which is generated from such a case expression. We also need to know the arity of the data constructors; fortunately this information is both available in the definition of data constructors and in the left hand side of the case alternatives.

The GHC compiler does not generate a case alternative for every data constructor of a given type. The case expression generated from a partial function will contain a default case alternative for the non-defined constructors. We will use the primitive nomatch function for the missing case alternatives. With this remark, these functions are neglected from the rewrite rule presented here for the sake of readability.

\[
\begin{align*}
\text{case } x \text{ of} \\
C_1 \ \vec{x}_1 & \rightarrow e_1 \Rightarrow \text{select } x \\
\lambda \vec{x}_{I(1)} &= e_{I(1)}
\end{align*}
\]
The $I(n)$ function yields the index of the constructor in the list of case alternatives by the global constructor index $n$.

The transformed STG factorial function to SAPL:

```haskell
Main.fact ds_sFz = select ds_sFz (\ ds1_sFC =
    if (eq ds1_sFC 0) (GHC.Types.I# 1)
        (GHC.Num.* GHC.Num.fNumInt ds_sFz
            (Main.fact (GHC.Num.- GHC.Num.fNumInt ds_sFz (GHC.Types.I# 1)))))
```

Figure 3.6: The transformed STG factorial function to SAPL.

This final step results in a valid SAPL function (Figure 3.6). In the following section we discuss interoperability issues, that is the differences between SAPL functions generated from Clean and GHC through STG.

### 3.6 Interoperability issues

In addition to the simplicity of the factorial function, there is another reason why we have chosen the factorial function as example in this chapter. It can be compiled by both Haskell and Clean compilers without any modification. This property gives us the opportunity to compare objectively the generated SAPL functions.

Van Groningen et al [126] identified the most salient differences between Clean and Haskell. These are: modules, functions, macros, newtypes, type classes, uniqueness typing, monads, records, arrays, dynamic typing, and generic functions. Studying this list carefully one can recognize that after the elimination of types and syntactic sugar we have to deal with the different low level representation of basic constructs only, e.g. the representation of basic types, arrays, records, tuples and so on.

First of all, it must be defined what we mean by the term interoperability here. Consider the following setting: we have two pieces of programs, one in Haskell and one in Clean. In both pieces of code we define the same type. ¹ We have different functions working on this type written in Clean and Haskell. What happens if we compose them after their translation to SAPL is the primary question this section wants to answer.

The following SAPL code snippet shows the code of the factorial function produced by the Clean compiler:

¹It is not always possible, see [126], but we will study interoperability only for the cases when compatible types exist.
\[ \text{fact } n = \text{if } (\text{eq } n \ 0) \ 1 \ (\text{mult } n \ (\text{fact } (\text{sub } n \ 1))) \]

Comparing it to Figure 3.6, the code fragments are isomorphic except for two parts, one of which is non-essential and one which is essential. We first discuss the non-essential one, and then concentrate on the essential difference. The less significant difference is the usage of type classes in GHC. The \texttt{GHC.Num.\texttt{-}} and \texttt{GHC.Num.*} functions are responsible for subtracting and multiplying numbers. They can work on any kind of number, their first argument is a type class instance, a dictionary (\texttt{GHC.Num.fNumInt} in this example, the \texttt{Int} instance of the \texttt{Num} type class). Substituting them for the wrapper functions \texttt{add’} and \texttt{mult’}, clarifies the similarities and differences:

\[ \text{fact } ds_sFz = \text{select } ds_sFz (\lambda ds1_sFC = \text{if } (\text{eq } ds1_sFC \ 0) (\text{GHC.Types.I#} \ 1) (\text{mult’ } ds_sFz (\text{fact } (\text{add’ } ds_sFz (\text{GHC.Types.I#} \ 1)))) \]

The essential difference is related to the low level representation of primitive types in SAPL generated from Clean, and SAPL generated from STG (SAPL* in the following). In the following subsections these, and also the differences between the representation of other basic types, are discussed.

**Primitive types**

This final form shows one fundamental difference: Haskell primitive values are explicitly boxed by a type-specific data constructor (e.g. \texttt{GHC.Types.I#} for the type \texttt{Int}). These boxing semantics [68] are important properties of Haskell and play an essential role in handling of strictness (see Section 3.7).

These semantics obviously cause interoperability problems. To be able to call the SAPL* factorial function from SAPL and vice versa, the argument and the return value must be boxed and unboxed, respectively. Boxing, however, must be done carefully as the boxed value must be primitive (not a closure). For this purpose some built-in, strict wrapper functions will be used.

In the following snippet, the \texttt{select} expression unboxes the result, while the integer argument is wrapped by the \texttt{boxInt} built-in function.

\[ \text{Clean.fact } x_0 = \text{select } (\text{Haskell.fact } (\text{boxInt } x_0))) (\lambda r = r) \]

**Algebraic data types**

Instances of ADTs can be passed between SAPL and SAPL*, provided that constructor indexes must match.

**Lists and tuples**

Lists and tuples are represented as ADTs in both SAPL and SAPL*. The only difference is in constructor names. SAPL* uses rather special names to refer
to these types. For \texttt{Nil} and \texttt{Cons} the keywords \texttt{[]} and : are used, tuples with different arities are denoted by (,), (,,), ... . In contrast, SAPL uses the following compatible definitions:

\[
:: \text{predefined}\_\text{Cons} \ a1 \ a2 \mid \text{predefined}\_\text{Nil}
:: \text{predefined}\_\text{Tuple1} \ a1
:: \text{predefined}\_\text{Tuple2} \ a1 \ a2
... 
\]

As for interoperability, only constructor indexes matter and this requirement is satisfied by these types.

### Strings

The double-quoted form of a string literal in Haskell is just syntactic sugar for list notation. This is a fundamental difference to Clean where strings are represented as unboxed arrays, that is simple objects. The difference is so essential that it can be solved only by runtime conversion. The two different string representations must be converted to each other by applying special conversion functions to them.

### Arrays

Clean has extensive language support, including syntactic sugar, for the efficient handling of arrays. However, the syntactic sugar is completely eliminated by the Clean compiler frontend, and it appears as a set of primitive functions at the level of SAPL. Haskell has no built-in support, it provides arrays via a standard module. Because there are no special elements for arrays in neither languages, the problem is simplified to using the same implementation through different APIs. In this case, a special SAPL implementation of Haskell \texttt{Data.Array} module must be provided which ensures compatibility by using SAPL primitive functions for array creation and access.

### Type classes

Both GHC and Clean generate ADTs from type classes. By the above-mentioned remarks, these are compatible at this representation level.

### 3.7 Benchmarks

So far we concentrated on only a valid transformation technique. In this section we discuss performance issues and present comparison of run-times
Table 3.1: Run-times of GHC and Clean compiled code running on an Intel Core 2 Duo T5600 PC clocking at 2GHz with 3 GB of memory. The JavaScript benchmarks, compiled from Sapl, were executed in Google Chrome 15.0.874.120.

<table>
<thead>
<tr>
<th>Program</th>
<th>GHC</th>
<th>G</th>
<th>GHC -O2</th>
<th>Stack size</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjoxo</td>
<td>3.99</td>
<td>21.27</td>
<td>0.39</td>
<td>0.28</td>
<td>128</td>
</tr>
<tr>
<td>Braun</td>
<td>3.16</td>
<td>18.47</td>
<td>0.47</td>
<td>0.68</td>
<td>383</td>
</tr>
<tr>
<td>Cichelli</td>
<td>1.83</td>
<td>21.12</td>
<td>0.06</td>
<td>0.13</td>
<td>27</td>
</tr>
<tr>
<td>Clausify</td>
<td>9.93</td>
<td>26.61</td>
<td>14.47</td>
<td>0.39</td>
<td>28</td>
</tr>
<tr>
<td>CountDown</td>
<td>3.16</td>
<td>18.50</td>
<td>0.06</td>
<td>0.09</td>
<td>53</td>
</tr>
<tr>
<td>Fib</td>
<td>22.90</td>
<td>-</td>
<td>-</td>
<td>5.78</td>
<td>1739</td>
</tr>
<tr>
<td>KnuthBendix</td>
<td>11.88</td>
<td>88.23</td>
<td>13.35</td>
<td>1.05</td>
<td>18</td>
</tr>
<tr>
<td>MSS</td>
<td>11.88</td>
<td>88.23</td>
<td>13.35</td>
<td>1.05</td>
<td>18</td>
</tr>
<tr>
<td>OrdList</td>
<td>39.41</td>
<td>24.94</td>
<td>23.84</td>
<td>1.09</td>
<td>573</td>
</tr>
<tr>
<td>PermSort</td>
<td>3.94</td>
<td>140.82</td>
<td>22.17</td>
<td>1.98</td>
<td>510</td>
</tr>
<tr>
<td>Queens</td>
<td>64.91</td>
<td>368.33</td>
<td>109.90</td>
<td>1.80</td>
<td>1070</td>
</tr>
<tr>
<td>Repeat</td>
<td>39.41</td>
<td>24.94</td>
<td>23.84</td>
<td>1.09</td>
<td>573</td>
</tr>
<tr>
<td>Taut</td>
<td>11.81</td>
<td>27.12</td>
<td>14.91</td>
<td>0.30</td>
<td>96</td>
</tr>
<tr>
<td>While</td>
<td>6.36</td>
<td>23.69</td>
<td>0.37</td>
<td>0.27</td>
<td>63</td>
</tr>
<tr>
<td>Queens1</td>
<td>2.22</td>
<td>10.14</td>
<td>3.56</td>
<td>0.12</td>
<td>40</td>
</tr>
<tr>
<td>Queens2</td>
<td>2.42</td>
<td>51.59</td>
<td>26.46</td>
<td>0.57</td>
<td>481</td>
</tr>
</tbody>
</table>

1. Because of high stack utilization Chrome must be run with the –js-flags="--stack-size 2048" switch.

Note: Programs with a 0 run-time were not included in the run-time statistics.
Figure 3.7: The STG code of the factorial function generated by GHC version 7.2.1 using optimized settings.

of JavaScript code generated from SAPL code gained from Clean and GHC through STG using the above described technique.

In Figure 3.7 the optimized STG code (using GHC -O2) of the factorial example is presented. Comparing it to Figure 3.4 one can identify two essential differences. First of all, the factorial function is split into two ones: separate versions are generated for strict and non-strict calls of the function. The entry function (Main.fact) is now responsible only for unboxing the argument and boxing the result. The actual computation logic is moved into a new function which works on an unboxed primitive integer. As for the second difference, the usage of type classes is replaced by direct applications of primitive functions (-#, *#). These modifications result in a completely different SAPL code:

```haskell
Main.wfact = 
{} λ x {ww_sloS} 
  case ww_sloS of 
    default → 
      let sat_slt0 = -# {ww_sloS 1} 
      in case sat_slt0 of 
        default → 
          let ww1_s1oX = Main.wfact sat_slt0 
          in case ww1_s1oX of 
            default → *# {ww_sloS ww1_s1oX} 
      0 → 1

Main.fact = 
{} λ x {w_sloZ} 
  case w_sloZ of 
    GHC.Types.I# {ww_slp2} → 
      let ww1_s1p4 = Main.wfact ww_slp2 
      in case ww1_s1p4 of 
        default → GHC.Types.I# {ww_sloS ww1_s1p4}
```

The new Main.wfact function, apart from the names of variables and primitive functions, is equivalent with the SAPL function generated by the Clean compiler. This optimization has a big impact on execution time and memory consumption, and this is how strictness is handled in Haskell [68].

Table 3.1 presents the results of an extensive range of benchmark programs. It shows the run-times of the binary programs, the run-times of their JavaScript
counterpart, finally, the memory usage and stack utilization of the binary programs. Every such test is evaluated using version 2.3 of the Clean compiler, GHC version 7.2.1 and the same GHC compiler version with the O2 flag. The JavaScript code is generated from SAPL which is produced by the Clean compiler and by compiling the STG output of the GHC compiler using the above presented transformation.

All of the contemporary browsers have call stack size limitations, called recursion limit. As for Chrome, it is currently a bit more than 20,000. Functional programs inherently tend to use a huge amount of stack, so this limitation affects several benchmarks. With the exception of Queens benchmark, when we could overcome the problem by increasing moderately the stack size of Chrome, these ones must have been compiled by optimizing stack use instead of speed. In the table, this special compilation technique, along with the corresponding stack sizes, is indicated by bold faced numbers. Moreover, some JavaScript benchmarks fail because of high memory consumption. In these cases there are no results at the JavaScript run-times and italic font face is used to highlight the memory consumption numbers of the related binary programs. Please note, that the stack/memory consumption of the JavaScript programs was not measured explicitly. The related problems were indicated by Chrome by the means of error messages.

The employed benchmark programs are part of the Reduceron project [106], for details of the programs, including source code, please visit the homepage of the project [107]. The pre-compiled SAPL, STG and JavaScript code can be obtained from the SAPL/STG homepage [37].

Analyzing the results of Table 3.1 we can make some general observations about them:

• The run-times of generated JavaScript programs are on par with the run-times of the corresponding binaries, meaning that the run-times of binaries and JavaScript programs of the certain benchmarks have the same order. There are only three benchmarks breaking this rule (Braun, KnuthBendix, SumPuz), but one of these results (Braun) is related to the unavoidable loss of speed coming from the optimization for stack use instead of speed.

• The high stack utilization of binaries clearly indicates high stack utilization of the related JavaScript programs.

• The high memory consumption of binaries clearly indicates high memory consumption of the related JavaScript programs.

According to these observations, the results indicate that the introduced STG $\rightarrow$ SAPL* $\rightarrow$ JavaScript transformation preserves the properties of the
original programs at a deep level. Notice, however, that the generated JavaScript benchmarks run about 1 to 2 order of magnitude slower than their binary counterparts. Why should we translate Haskell/Clean programs to JavaScript then? The answers are manifold. First of all, there are tasks that inevitable must run in the browser, like validation or management of GUI elements. For a second answer, even if the applications run slower in the browser, most everyday programs they have acceptable run-times. This is even true for relatively large applications, like the SAPL to JavaScript compiler itself.

3.8 Related work

Compilation of traditional programming languages to JavaScript has drawn much attention in the last years as client-side processing for Internet applications has been gaining importance. Virtually every modern language has some kind of technology which allows it to be executed in a browser. A comprehensive overview of JavaScript related technologies of functional languages is given in [39]. Now, we are particularly interested in Haskell based compiler technologies.

There are several implementations of compilers from Haskell to JavaScript; although none of them has reached a full release stage as of the time of writing. Currently there are three important JavaScript backends for the various Haskell implementations. One of the first attempts to compile Haskell to JavaScript took a similar approach as ours, it also used STG as source language, but they implemented an STG machine in JavaScript [120]. Later on this project moved to YHC Core instead of STG, but it is abandoned since the development of York Haskell Compiler is canceled. There is another backend under development for Utrecht Haskell Compiler which is going to be part of the upcoming release [35]. As for GHC, after some initial attempts, the most promising technology is GHCJS [63] which is still in its alpha stage at the time of writing. Finally, a bit different but interesting approach was chosen by JSHC [20] which implements a Haskell2010 compiler in JavaScript. Our approach, to convert the source language to obtain a new target platform is a novel idea which has the definite advantage of reusing well established technologies and that it allows interoperability between different languages.

3.9 Conclusion and future work

In this chapter we presented a technique to translate STG, one of the core languages of GHC, into SAPL, one of the core languages of Clean. To achieve this we used a method called compilation by transformation; the actual transformation consists of a series of rewrite rules. The presented transformation is
restricted to pure functions only. The viability of the technique is proved by a prototype implementation which was used for the compilation of an extensive range of benchmark programs. The translated benchmarks were converted to JavaScript to compare their run-times to the same benchmarks generated by the Clean compiler. Analyzing the results we concluded that the transformation preserves well the run-time characteristics of the original programs.

We have not discussed the question of the reverse transformation, converting SAPL into STG. This task certainly would be harder, because STG, in contrast to SAPL, contains much information for the code generator. However, the necessary information is generated in a later stage by the SAPL to JavaScript compiler, thus the reverse conversation is probably possible.

The current prototype implementation of the compiler also needs further improvement to be applicable for real world tasks. The most important improvement is to investigate the impact of rewrite rules to monadic I/O and how to translate non-pure functions safely. It is also very important to improve the performance of the SAPL* applications. One of the possibilities is to deal better with the strictness optimization in the STG code. In the latest version of the SAPL language, a Clean like strictness annotation was introduced [39] for supporting strictness optimization. Performing strictness analysis on SAPL* code to gain annotated SAPL code could significantly improve the run-time execution characteristics of the translated STG applications.
Part II

iTasks integration
Tasklets: Client-side evaluation for iTask3

iTask3 [113] is the most recent incarnation of the iTASK framework for the construction of distributed systems where users work together on the internet. It offers a domain specific language for defining applications, embedded into the lazy functional language Clean. From the mere declarative specification a complete multi-user web application is generated. Although the generated nature of the user interface (UI) entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom UI building blocks. In this paper, we present an extension to the iTASK3 framework which introduces the concept of tasklets for the development of custom, interactive web components in a single language manner. We further show that the presented tasklet architecture can be generalized in such a way that arbitrary parts of an iTASK application can be executed on the client.

4.1 Introduction

The iTASK framework was originally developed as a dedicated web-based Workflow Management System (WFMS). Its most recent incarnation, iTASK3, however, extends its boundaries beyond classical WFMS and offers a novel programming paradigm for the construction of distributed systems where users work together on the internet.

According to the iTASK paradigm, the unit of application logic is a task. Tasks are abstract descriptions of interactive persistent units of work that have a typed value. When a task is executed, it has an opaque persistent value, which can be observed by other tasks in a controlled way. In iTASK, complex multi-user interactions can be programmed in a declarative style just by defining the tasks that have to be accomplished. The specification of the tasks is given by a domain specific language embedded in the pure, lazy functional language Clean. Furthermore, the specification is given on a very high level of abstraction and does not require the programmer to provide any user interface definition. Merely by defining the workflow of user interaction, a complete multi-user web application is generated, all the details e.g. the generation of web user interface, client-server communication, state management etc. are automatically taken care of by the framework itself.
Chapter 4

Developing web applications such a way is straightforward in the sense that the programmers are liberated from these cumbersome and error-prone jobs, such that they can concentrate on the essence of the application. The iTask system makes it very easy to develop interactive multi-user applications. The down side is that one has only limited control over the customization of the generated user interface, but for this type of applications, this is often acceptable. However, the experiment with real world applications, e.g. the implementation of the Netherlands Coast Guard’s Search and Rescue (SAR) protocol [93, 94], indicated that even if the functional web design is satisfactory, custom building blocks may be required for the purpose of user-friendliness. A good example is the aforementioned SAR workflow, where Google MAPS widgets complemented the otherwise functional web application to visualize the locations of incidents.

To overcome this shortcoming, in this paper we present an extension for the iTask3 system which enables the development of such widgets, the so called tasklets. Tasklets are seamlessly integrated into iTask to preserve the elegance of functional specification by hiding the behavior behind the interface of a task. Tasklets are developed in a single-language, declarative manner and in accordance with the model-view-controller user interface design (MVC) [89]. MVC decouples the application logic (the controller), the application data (the model) and the presentation data (the view) to increase flexibility and re-use. Technically speaking, tasklets are embedded applications which behavior is encoded in Clean written event handler functions. The event handlers are executed in the browser, where, they have unrestricted access to client-side resources. Using browser resources the tasklet can create custom appearance and exploit functionality available only in the browser (e.g. HTML5 GeoLocation API), utilizing the event-driven architecture the tasklet can achieve interactive behavior. With this extension, iTask gains similar characteristics to multi-tier programming languages like Links [28] or Hop [116, 117], in the sense that the same language is used to specify code residing on multiple locations or tiers, such as the client and the server.

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

In this paper we make the following contributions:
The iTask framework is extended to enable the development of client-side, interactive UI components in a single-language, declarative manner. These components can be used to increase the expressiveness of the functional iTask applications, and to provide functionality which is available only in the browser. This facility, called tasklet, is designed in such a way to fit as seamlessly as possible into the iTask formalism, that is to be opaque for the developer of the functional specification and to retain the advantageous generated nature of user interfaces of iTask applications as much as possible;

- Tasklets foster the model-view-controller user interface design to separate the application logic, the application data and the presentation data. The separation of these roles helps with increasing code flexibility, reuse and maintainability;

- We further show that the tasklet architecture is versatile enough to pave the way for the evaluation of almost all tasks at the client-side. Executing tasks in the browser helps with avoiding client-server communication to reduce server load and provide smoother user experience. This feature also creates the preconditions for running iTask applications offline in a browser which is a desired direction of future development;

- Finally, tasklets utilize a special compilation technique to enable the execution of arbitrary expression of an iTask application in the browser without shipping of unnecessary code. This technique is based on runtime deserialization of Clean expressions and involves on the fly compilation to JavaScript. By minimizing the amount of client code, this approach has the definite advantages of reducing communication cost and memory usage in the browser. Moreover it makes it possible to dynamically tune the set of tasks executed in the browser by the current server load or other run-time information.

The remainder of this paper is organized as follows: in Section 4.2 we start with a short overview of the iTask framework and develop a non-trivial, but necessarily simplified example of a flight check-in application to give a taste of iTask. In Section 4.3 we introduce the tasklet architecture and demonstrate its usage by developing a tasklet to enrich the example of the previous section. Some real-world use cases studies are discussed in Section 4.4. In Section 4.5 we briefly discuss the design of the tasklet architecture, then we generalize it in Section 4.6 to enable the execution of legacy tasks; some common restrictions on its applicability is also given in this section. After a discussion of related work in Section 4.7, we conclude in Section 4.8.

The iTask framework has been created in Clean. A concise overview of the syntactical differences with Haskell is in [8]. We assume the reader is familiar with the concept of generic programming and uniqueness typing.
Chapter 4

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

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

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

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

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

Figure 4.1: Combinators and primitive tasks used in the paper

4.2 Introduction to iTask

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

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

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

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

In Figure 4.1, the small set of combinators and primitive tasks of the iTask
DSL is presented which are used throughout this paper (for reasons of present-
ation, the types have been slightly simplified). The full language definition and
its semantics can be found in [113].

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

\[
\begin{align*}
\text{Seat} & = \text{Seat Int Int} \quad \text{// Seat information: row, seat number in the row} \\
\text{Seats} & = [\text{Seat}] \\
\text{Booking} & = \{ \text{bookingRef} :: \text{String} \quad \text{// Unique booking reference number} \\
& \quad \text{, firstName} :: \text{String} \quad \text{// Passenger’s first name} \\
& \quad \text{, lastName} :: \text{String} \quad \text{// Passenger’s last name} \\
& \quad \text{, flightNumber} :: \text{String} \quad \text{// Flight number} \\
& \quad \text{, pid} :: \text{Hidden String} \quad \text{// Unique number of passenger’s ID} \\
& \quad \text{, seat} :: \text{Maybe Seat} \quad \text{// Seat information} \} \\
\text{Flight} & = \{ \text{flightNumber} :: \text{String} \quad \text{// Unique flight number} \\
& \quad \text{, free} :: \text{Seats} \quad \text{// List of free seats} \} \\
\end{align*}
\]

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

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

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

To keep the example as concise as possible, a very simple exception controlled mechanism is used to handle errors; when an exception occurs the application prints the error message and restarts the workflow. Therefore, the main task, checkIn, is responsible for handling exceptions only. The task does not return any meaningful value (Void), its semantics is based on side-effect:

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

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

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

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

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

```haskell
:: BookingInfo = BookingReference String | PassangerLastName String
```
In `lookUpBooking`, if a reference number was provided, the booking record is
looked up. Otherwise the user is asked to choose (using `enterChoice`) one of
the booking records in which the passenger’s last name matches and contains
no seat information. The function returns `Nothing` if a booking record could
not be found:

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

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

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

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

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

---

Figure 4.2: The flight check-in screens

In `lookUpBooking`, if a reference number was provided, the booking record is
looked up. Otherwise the user is asked to choose (using `enterChoice`) one of
the booking records in which the passenger’s last name matches and contains
no seat information. The function returns `Nothing` if a booking record could
not be found:
Figure 4.2 shows the screenshots of the application. As it can be seen, the user interfaces are automatically generated from the type of the tasks only. Nevertheless they commonly look fine and intuitive to use. The only exception in this example is the fourth screen shown; choosing a seat from a list of seat numbers is anything but user friendly. In the next section we develop a more intuitive UI component, a tasklet, for choosing a seat by looking at the layout of the airplane.

4.3 Introduction to tasklets

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

However, despite the common basis, there are many important differences to most multi-tier programming languages. First of all, tasklets are not for the development of complete, customized applications. It is designed to develop independent components to be attached to the generated trunk of an iTask application. As such, we decided not taking the usual lightweight, view-centric web development approach but enforce the model-view-controller user interface design in tasklet development. We believe that the separation of roles suits better the development of components and it is more consistent with the objectives of iTask. This heavyweight approach also fits better for a lazy, purely functional language like Clean, where the expression of side-effects needs special attention.

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

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

```
:: Tasklet st val = { generatorFunc :: (*World → *(TaskletHTML st, st, *World))
, resultFunc :: (st → TaskValue val)
}
:: TaskValue a = NoValue | Value a Stability
```
Tasklets: Client-side evaluation for iTask3

:: Stability := Bool

During initialization, generatorFunc is executed on the server to provide the initial state and user interface of the tasklet. Its only argument, a value of the unique type *World, allows access to the external environment. The current value of the tasklet is calculated when necessary by resultFunc from its internal state. The result type, TaskValue a, an iTasks system type, expresses that the result of a task execution can be an actual value (Value) which is stable or unstable, or can indicate no meaningful value (NoValue).

For the explanation of value stability, please refer to [113], in this paper we always use stable return values, which basically tells the task engine that the computation of the actual task is finished. The user interface (the view) and its behavior (the controller) are defined by the TaskletHTML structure:

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

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

The actual user interface (html field) can be given by any data structure provided that it has an instance of the function class toHtml. In the following, we will use an overly simplified ADT to create HTML definitions which suits well our straightforward example, however may not satisfying for more complicated ones. Core iTask already supports the generation of high-level web forms based on the iData [110] toolkit. In this case full, low-level control over the definition of HTML elements is needed. This can be done in an abstract, monadic way like in Wash [124] or by an XML like domain specific language similar to that of Hop. Furthermore, the MVC concept enables that the three components can be developed separately, and specifically allowing the View to be developed by non-programmers. For this reason, some template mechanism also could be considered to be added similar to e.g. Yesod [118] or Snap [26]. However, providing any particular tool here would beyond the scope of this paper.

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

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

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

\[
\text{mkTask :: (Tasklet st a) \rightarrow Task a}
\]

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

### 4.3.1 Seat choosing by map

To clarify the usage of tasklets, we enrich our example with the aforementioned seat chooser component. So far the passenger was to choose a seat from the list of available seats by their designation. The new idea is to allow the user choosing by looking at a simplified seat map of the airplane as it is shown in Figure 4.3. For this, the Flight record is extended with layout information:

\[
\text{:: Flight} = \{ \ldots
, \text{rows} :: \text{Int} \quad // \text{Number of rows}
, \text{layout} :: [\text{Int}] \quad // \text{Layout of a row}
\}
\]

[Figure 4.3: Choosing a seat]
The rows and layout fields contain the number of rows on the plane and the layout of the rows, respectively. If the layout value is [2,3], rows consist of 5 seats in 2 groups: 2 seats, corridor, 3 seats.

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

\[
\text{chooseSeat (Just } f) = \text{mkTask seatChooserTasklet where}
\]

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

\[
\text{seatChooserTasklet :: Tasklet (Maybe Seat) Seat}
\]

\[
\text{seatChooserTasklet} =
\{ \text{generatorFunc} = (\lambda \text{world} \rightarrow (\text{TaskletHTML gui, Nothing, world}))
, \text{resultFunc} = \text{maybe NoValue (}\lambda v \rightarrow \text{Value v True)}
\}
\]

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

\[
\text{rowLayout } = \text{intercalate [-1] (numbering 1 f.layout)}
\]

\[
\text{numbering i []} = []
\]

\[
\text{numbering i [x:xs]} = [\text{take x [i..]} : \text{numbering (i+x) xs}]
\]

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

\[
\text{genRowUI (Seat _ -1)} = \text{DivTag [corridorStyle] []}
\]

\[
\text{genRowUI seat | elem seat f.free}
\]

\[
\text{= DivTag [TitleAttr (toString seat), IdAttr (genSeatId seat), freeStyle] []}
\]

\[
\text{= DivTag [TitleAttr (toString seat), occupiedStyle] []}
\]

\[
\text{seatMap = DivTag [] (intercalate [DivTag [newRowStyle] []]
}\]

\[
[\text{map (}\lambda s \rightarrow \text{genRowUI (Seat r s)) rowLayout \_ \_ r \leftarrow [1 .. f.rows]]})
\]

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

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

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

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

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

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

setObjectAttr :: JSValue String a *JSWorld → *JSWorld

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

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

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

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

The goal of the second project, called Tonic, is to develop an infrastructure to graphically represent the definition and behavior of tasks. It translates a textual iTASK specification into a graphical one, called a blueprint. The Clean compiler has been adjusted to generate blueprints, and a standalone application, a Tonic viewer, written in iTASK, is developed to visualize them. Such a blueprint is basically a general graph, which consists of special kind of annotated nodes and edges. To be able to draw graphs, a general tasklet is developed. This tasklet is able to create a graphical representation of a graph \( \text{Graph n e} \), provided that a \( \text{GraphletRenderer n e} \) instance for the given node and edge types exist. The graph is given in a standard, functional way, while the renderer must provide a description understandable by the D3 JavaScript library, on which the tasklet is based.
The client-side execution architecture is designed in such a way that the two groups of functions, executed on the client versus executed on the server, are not designated during compilation. Instead of this, two images of the same application are produced by the Clean compiler: the server executable running in native code and an intermediate representation that can be compiled to JavaScript (see Figure 4.4). For the intermediate representation, the so called Simple Application Programming Language (SAPL) [79], a core, lazy functional language is utilized. It is used to execute arbitrary Clean expressions in the browser as follows:

0. There are two images produced by the Clean compiler: a server image (native code, executable) and a SAPL image (intermediate representation);

1. The executable on the server is started;

2. Instead of evaluating an expression on the server, one can decide, at runtime, to evaluate it on the client instead. This can in principle be done for any expression;

3. The expression to evaluate on the client is at run-time converted to an equivalent SAPL expression;

4. This SAPL expression is passed to the run-time linker specially developed for this purpose. The linker collects the dependencies of the expression recursively using the SAPL image of the application;

5. The result is run through a caching mechanism to filter out SAPL code already processed in a previous session;
6. The remaining SAPL code is on the fly compiled to efficient JavaScript code by a newly developed SAPL-to-Javascript compiler [39];

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

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

4.6 Task evaluation on the client

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

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

1. The state transition function is executed on the server to create the user interface and the result value;

2. The result value can be observed by other tasks; they can decide to continue with this current value. In that case the observed task is terminated;

3. The user interface information is sent to the browser to display;

4. If any event occurs on the client, it is passed to the state transition function on the server and the procedure continues with step 2.

The standard way tasks are evaluated closely fits the architecture of tasklets: (1) there is a distinct state to work on (2) the state transition function generates a new state and user interface just as we need in generatorFunc (3) the user interface generates events (4) event handlers modify the state and the user interface (see step 4). The consequence of this perfect fit is that it is possible to define one general tasklet creator to run any task exclusively in the browser:

\[ \text{runOnClient :: (Task a) } \rightarrow \text{ Task a} \]

The result of the runOnClient task is a tasklet in which the state transition function of the enclosed task is utilized in generatorFunc and in the event
handlers. Neglecting any details, at this point the tasklet API was slightly generalized to enable these functions to create and interact with TUI elements in addition to HTML. When the value of `runOnClient anyTask` is requested, the state transition function of `anyTask` is called on the server to create the initial user interface and state of `anyTask`. These, and the JavaScript counterpart of the event handlers (implicitly containing the state transition function) are sent to the browser. Figure 4.5 summarizes the client part of the generalized architecture:

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

### 4.6.1 Limitations

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

4.7 Related work

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

Several other languages address multi-tier programming. In the imperative world the most modern approach is the Google Web Toolkit (GWT) [66], Google Dart [33] and Node.js [122]. GWT utilizes a Java to JavaScript compilation technique for building complex browser-based applications. GWT fosters classical GUI programming where widgets can be developed using a programming model comparable to that of tasklets.

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

Hop [116, 117] uses a declarative approach. It is a dedicated web programming language with a HTML-like syntax built on the top of Scheme. Hop uses two compilers, one for compiling the server side program and one for compiling the client-side part. The client side part is only used for executing the user interface. Hop uses syntactic constructions for indicating client and server part code. The application essentially runs on the client and may call services on the server. In contrast, an iTask application essentially runs on the server and may execute services, tasklets, on the client.
Links [28] and its extension Formlets is also a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. In a Links program, the keywords client and server force a top-level function to be executed at the client or server respectively.

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

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

As for iTask, there are earlier implementations of similar features utilizing a Java written SAPL interpreter [79] as a browser plug-in. The iEditors [80] enables the development of interactive web UI elements as tasklets do, however, it does not allow direct access to browser resources, therefore its applicability is restricted to functionality provided by the plug-in. As a consequence, it does not have the single-language property either, because for some functionality the plug-in has to be extended using Java. There also had been client-side task evaluation attempts for an early version of iTask using the same plug-in based interpretation technology [112]. However, our approach, to give one general solution for both of the problems is a novel strategy.

### 4.8 Conclusion

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

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

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

Finally, we showed that the presented tasklet facility can be generalized to enable the execution of ordinary tasks in the browser instead of the server by turning an arbitrary task into a tasklet. This, amongst other things, can be used to improve the responsiveness of an iTask application by avoiding the latency of communication.
5 Editlets: type-based, client-side editors for iTasks

The iTask framework enables the construction of distributed systems where users work together on the internet. It offers a domain specific language for defining applications, embedded in the lazy functional language Clean. From the mere declarative specification of the tasks to do and their interconnection, a multi-user web application is generated which can coordinate the work thus described. User interfaces are generated automatically which is realized by using type-driven generic functions. Although this way of generating user interfaces entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom user interface building blocks. In a precursory work we proposed tasklets for the development of custom, interactive web components. However, experimenting with real-world applications indicated that they lack some fundamental properties limiting their usability; these are the tight integration with the type-driven user interface generation, and the capability of working with shared data. In this paper, we introduce editlets to overcome these limitations. In addition, editlets also provide a general way to communicate changes instead of exchanging the whole data to reduce communication overhead.

5.1 Introduction

Task Oriented Programming [113, 92] (TOP) is a paradigm for designing multi-user, distributed web-applications. The iTask system [113], or iTasks, is a TOP framework that offers a domain specific language which is shallowly embedded in the strongly typed, lazy, purely functional programming language Clean [114, 126].

In the TOP paradigm, the unit of application logic is a task. Tasks are descriptions of interactive persistent units of work that maintain a typed value. When a task is executed, it has a persistent value, which may change over time reflecting the current state of the work taken place. This task value can be observed by other tasks. In iTasks, complex multi-user interactions can be programmed in a declarative style just by defining the work that has to be accomplished. The definition is given on a very high level of abstraction and does not require the programmer to provide any user interface definition.
Merely by defining the workflow of user interaction, a complete multi-user web application is generated, all the details e.g. the generation of web user interface, client-server communication, state management etc. are automatically taken care of by the framework itself.

The iTask system uses generic programming [73, 12] and a hybrid static-dynamic type system [129, 128] to generate the user interface. From the programmer’s perspective, it is achieved in two levels. At the most basic level, the iTask engine can be asked to generate a graphical user interface (GUI) for any conceivable first order model type. iTasks uses a predefined set of primitive user interface elements to generate the GUI, a client-side editor, for the given type, then dynamically creates an associated primitive task. On the higher level, additional user interface elements are generated automatically as tasks are combined together. These elements reflect the actual combinators in use and express the “flow” of the application.

Developing web applications in such a way is straightforward in the sense that programmers are liberated from these cumbersome and error-prone jobs, such that they can concentrate on the essence of the application. The iTask system makes it very easy to develop interactive multi-user applications. The down-side is that one has only limited control over the customization of the generated user interface. In real-world applications, it is often necessary to develop custom user interface elements to achieve special functionality.

To overcome this limitation, in previous work we introduced tasklets [45], a special primitive task type, for the development of custom, interactive web components. Tasklets are written in Clean and executed in a web browser using a Clean to JavaScript compilation technique [39]. In the browser, they have unlimited access to browser resources through some library functions while on the server they behave like ordinary iTasks tasks.

Using tasklets, we have successfully developed many interactive components for a wide range of applications, but we also experienced certain limitations of the technology:

1. Tasklets cannot work with shared data. As an example, it is not possible to create an interactive map, and enable multiple users to make concurrent modifications to that (e.g. add marks).

2. Tasklets are not compatible with the generic, type-driven user interface generation. User interfaces can be generated automatically for any first order type, in a compositional manner. For example, the system knows, given a user interface for type \( t \), how to construct a user interface for a container type \( t' \) in which type \( t \) includes, such as a list of \( t \) or a tree of \( t \). However, a user interface created for a tasklet belongs to a particular task, not to a type, and can therefore not be applied in the generic user interface generation.
3. Tasklets have a simple communication interface based on the exchange of the whole underlying data, which has a high associated communication cost.

Given the experience we obtained with developing real-world applications, we revised our first approach of defining custom interactive components. The new component type is called editlets. Editlets solve all the aforementioned limitations while preserving compatibility: in the most basic use cases they give back the functionality of tasklets.

Editlets also have the property that the client-server communication is done in edits, which means that the value of the editlet is communicated through changes, that is incremental updates, instead of exchanging the whole value at every update. In certain cases this drastically reduces the associated communication cost (consider a source code editor component as an example).

In this paper we show how editlets can be defined, how they work and interact with the other part of the iTasks system. This is done in a number of steps:

1. We extend iTasks with editlets. An editlet has an associated value type and consists of a description of the behavior of the component on the client-side, and the logic of creating and applying edits from and to its current value.

2. We develop a simple, but still realistic example of a drawing application, where multiple people can work on the same shared image, to give a taste of editlets.

3. We explain the technical background of editlets along with additional remarks on how they fit in the iTasks architecture.

The remainder of this paper is structured as follows: to set the context, we start with a short introduction of the iTask framework in Section 5.2. In Section 5.3 an overview of our old approach, tasklets, is given, where we also identify some general shortcomings of the architecture. Based on these shortcomings, we define new requirements in Section 5.4. The new editlet architecture is introduced in Section 5.5, then a small, but illustrative example is developed in Section 5.6. In Section 5.7 we briefly discuss the design of the architecture of the client-side execution used by editlets. After a discussion of related work in Section 5.8, we conclude in Section 5.9.

The iTask framework has been created in Clean. A concise overview of the syntactical differences with Haskell is in [8]. We assume the reader is familiar with the concept of type-driven generic programming and uniqueness typing [19].
Chapter 5

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

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

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

// Assigning a task to a user
:: User:=String
(@:) infix 3 :: User (Task a) → Task a | iTask a

// Shared Data Sources
:: ReadWriteShared r w
withShared :: b ((ReadWriteShared b b) → Task a) → Task a | iTask a & iTask b

// User interaction
enterInformation :: String → Task m | iTask m
updateInformation :: String m → Task m | iTask m
viewInformation :: String m → Task m | iTask m

// User interaction using shared data
updateSharedInformation :: String (ReadWriteShared r w) → Task w | iTask r & iTask w
viewSharedInformation :: String (ReadWriteShared r w) → Task r | iTask r & iTask w

Figure 5.1: Combinators and primitive tasks used in the paper

5.2 Introduction to iTasks

Task Oriented Programming (TOP) is a paradigm that is designed to construct multi-user, distributed web-applications. The iTask system is a TOP framework that offers four core concepts for software developers:

1. Tasks which are abstractions of the work that needs to be performed by (teams of) human(s) and software components. A task is a value of parameterized type (Task a). The type parameter a models the task value the task is currently processing. The task value may change over time while the task is being worked on (see [113]). The current value can be inspected by other tasks.

2. Shared data sources (SDS) which are abstractions of information that is
shared between tasks. An SDS is a value of type `(ReadWriteShared r w)`. The type parameters `r` and `w` model the *read* and *write* values. The shared data sources enable safe concurrent read/write access for some shared data, and offer change notifications (i.e. a task which depends on a particular SDS will be notified when the SDS has been changed by some other task).

3. **Combinator functions** that compose tasks and SDSs into more complex tasks and SDSs.

4. **Generic interaction** with the users. A TOP framework generates user interfaces *generically* for any type of data used by tasks. This means that it is not necessary to design a user interface and program event handling just to enter or view some information.

For programming convenience, a large set of primitive tasks, task combinators, and types are predefined in the iTask library on top of these core concepts. Figure 5.1 displays a fragment of these tasks, types and combinators, that we use in this paper.

Most type definitions of the iTasks combinators contain a context restriction at the end of their type signature, e.g. `| iTask m`, defining a restriction on type variable `m`. It is similar to context restrictions on overloading as can be found e.g. in Haskell. In Clean, context restrictions not only may refer to overloaded functions, it may also refer to generic functions. The context restriction `| iTask m` means, that `m` can be of arbitrary type, provided that a class of generic functions, necessary for the iTasks run-time system, have instances for type `m`. In contrast to the overloaded use, the programmer does not need to define these instances. The Clean compiler can automatically derive instances for generic functions for any conceivable first order model type.

In this paper we only use a few simple combinator functions for sequential and parallel compositions. Task `f >>= s`, created by the monadic `>>=` combinator, first performs task `f`, then the value produced by `f` can be used by task `s` to compute any new task expression. The `-||-` parallel combinator groups two tasks of the same type in parallel and returns the result of the task that is completed first. With the assign operator `& if`: a task to work on can be assigned to a specific user. It can be seen as a special case of a parallel task with the property that the task has to be performed by the indicated user.

Many functions are defined on Shared Data Sources, e.g. reading, writing, and observing. In this paper we only make use of the `withShared` function. It creates a shared data source in memory with a given initial value and a limited scope. The shared data can only be accessed by the task(s) defined in the continuation function.

In iTasks there are a family of primitive tasks for generating user interface for first order types. The system uses generic programming and a hybrid
static-dynamic type system to be able to achieve this functionality. Some of these tasks are `viewInformation`, to display, and `enterInformation`, `updateInformation`, to allow the user to enter or edit some data. The first argument of these functions is a brief description of what the end-user is expected to do. Commonly, these primitive tasks also have a counterpart that work with shared data instead of private, local data. In this paper we will use `updateSharedInformation` and `viewSharedInformation`.

To give a quick glimpse of iTasks, we present two very simple examples. In the first example a user is asked to enter two numbers in a sequence, after which their sum is displayed. One can see in the code how information produced by one task is passed to the next one in a monadic style. The corresponding generated user interfaces for doing the interaction are shown in Figure 5.2. In this example, GUIs are generated for type `Int`.

```
calculateSum :: Task Int
calculateSum
    = enterInformation "Enter a number"
    >>= \num1 -> enterInformation "Enter another number"
    >>= \num2 -> viewInformation "The sum of these number is:" (num1 + num2)
```

In the following example we define the task `updateAndView` delivering a value of arbitrary type `a`. It starts two tasks in parallel. One is assigned to `user1`, the other one to `user2`. Both tasks communicate via a newly created shared data source with initial value `initialValue`. For `user1` a user interface is created which enables her to update the data source, while `user2` is offered a view to follow the changes made. When a shared value is changed, all tasks that rely on it are automatically informed and refreshed.

```
updateAndView :: User User a -> Task a | iTask a
updateAndView user1 user2 initialValue
    = withShared initialValue
        (\shared -> update user1 shared
            -||-
            view user2 shared)
```

Figure 5.2: GUIs generated for values of type `Int`.
Figure 5.3: Generated GUI for entering values of type [Person].

```haskell
update u sh = u @: updateSharedInformation "enter information" sh
view u sh = u @: viewSharedInformation "view information" sh

A task like `updateAndView` is very generally applicable, it can be used for any first order type for which the `iTask` class of generic functions have been generated. The user interfaces that will be generated by the generic functions depend on the concrete type the task is applied with. In the task `twoWorkers` below, `updateAndView` is applied to let Alice and Bob work together, producing a value of type [Person]. Using a derive statement, an instance for all generic `iTasks` functions for this type is created by the compiler. Alice is offered an interface to create a list of Persons (see Figure 5.3), and the changes she made can “real-time” be observed by Bob.

```haskell
:: Person =
  { name :: String,
    placeOfBirth :: String,
    dateOfBirth :: Maybe Date
  }

derive class iTask Person

twoWorkers :: Task [Person]
twoWorkers = updateAndView "Alice" "Bob" []
```

Although it is nice that user interfaces can be generated for any first order type, one would like to have an easy way to do something different than the default behaviour and be able to let arbitrary work be done on the client using the latest facilities offered in modern browsers.

Furthermore, in general, an arbitrary number of tasks, varying over time, may view or update shared data. This can of course lead to conflicts when multiple people want to update the same data at the same time. The default behaviour of the system is to ignore a conflicting update. The update is lost and the corresponding task is refreshed showing the most recent information.
known. Although it is possible to redefine this default behaviour for a specific data source, when rich clients are being used one has to be able to define how to recover from a failing update on the client.

5.3 The first approach: tasklets

In iTasks, because the user interface is generated and handled automatically, one has only limited control over its customization. Although, for most of the iTasks applications, this is acceptable, our experiment with real-world applications, e.g. the implementation of the Netherlands Coast Guard’s Search and Rescue (SAR) protocol [93, 94], indicated that even if the functional web design is satisfactory, custom building blocks may be required for the purpose of user-friendliness.

To overcome this shortcoming, we presented an extension for the iTasks system which enables the development of such components, the so called tasklets [45]. Tasklets are seamlessly integrated into iTasks to preserve the elegance of functional specification by hiding the behavior behind the interface of a task.

Tasklets are developed in a single-language (i.e. Clean), declarative manner and in accordance with the model-view-controller user interface design (MVC) [89]. MVC decouples the application logic (the controller), the application data (the model) and the presentation data (the view) to increase flexibility and reuse. Technically speaking, tasklets are embedded applications whose behavior is encoded by means of event handler functions. The event handlers are compiled to JavaScript so that they can be executed in the browser. They have unrestricted access to client-side resources. Using browser resources the tasklet can create custom appearance and exploit functionality available only in the browser (e.g. HTML5 GeoLocation API). It utilizes the event-driven architecture to achieve interactive behavior. With this extension, iTasks gains similar characteristics to multi-tier programming languages like Links [28] or Hop [116, 117], in the sense that the same language is used to specify code residing on multiple locations or tiers, such as the client and the server.

A tasklet consists of the definition of a client-side application which has a state of type state, and a function which tells how to extract the result of the tasklet of type value from the state:

```
:: Tasklet value = ∃ state:
    { genUI :: *World → *(TaskletHTML state, *World)
      , resultFunc :: state → value
    }
```

The first part, the definition of the client-side application is generated by the genUI non-pure function (the uniquely attributed type *World allows access to the external environment), while the transition from state to value is
given by resultFunc. Finally, a tasklet must be turned into a primitive task to make it executable by iTasks:

\[
\text{mkTask :: } (\text{Tasklet value}) \rightarrow \text{Task value}
\]

The life cycle of a tasklet is the following: it starts when the value of the wrapper task is requested. First, \text{genUI} is executed on the server to provide the user interface of the tasklet. Then, the definition of the user interface is on the fly compiled to JavaScript and shipped to the browser. In the browser, the application is executed in a tasklet container. As it runs, when its state is changed, resultFunc is called to create a new task value that is sent to the server immediately. The life cycle of the tasklet is terminated by the framework when the task value is finally taken by another task.

Tasklets are backed by a JavaScript compilation technique integrated with the Clean compiler. During the compilation of an iTasks application, besides the server executable running in native code, an intermediate representation of the same application in the SAPL [79] language, is also created. This intermediate language is designed to contain only the essential minimum of language features of a lazy, functional language like Clean or Haskell, while preserving the semantics. Furthermore, its syntax is carefully constructed to be easy to handle at source code level. These features enable us to perform fast source code level linking based on any initial expression. The actual JavaScript compilation is done during run-time in a demand driven way: given an expression the depending SAPL code is collected and then compiled to JavaScript on the fly. This technique has the advantage of reducing the size of the generated code to the essential minimum. The compilation technique is explained in more detail in Section 5.7.

The tasklet architecture enables us not only to create custom interactive components, but it also allows us to execute arbitrary tasks on the client. This means that we can dynamically choose where a task is to be executed, on the server or on the client [45]. Tasklets are just perfect for the latter purpose, but further experiments revealed that they also have certain limitations concerning the creation of interactive components:

- In iTasks the user interfaces are generated in a generic, type-driven way. The iTask system can be asked to generate a user interface for a value of type \([a]\) for example, where \(a\) is any first order type. However, tasklets are not integrated with this type-driven approach. The main problem is that the user interface generated for the type \((\text{Tasklet } a)\), should produce a value of type \(a\) instead of \((\text{Tasklet } a)\). Thus, one should rather be able to tell the system that for a given type \(a\), instead of creating the generic user interface, use a specific tasklet instance of type \((\text{Tasklet } a)\) for customization.
- Tasklets cannot work with shared data. As an example, it is not possible to create an interactive map, and enable multiple users to make concurrent modifications to that (e.g. add marks). This is a serious limitation as working with shared data sources to achieve collaborative work is one of the main principles of iTasks.

- In certain applications tasklets have a huge associated communication overhead. An example of this is a tasklet which implements a syntax highlighted source code editor component. Exchanging the whole source code between the client and server every time when a part of it is changed has a big negative impact on the performance of the whole application.

We concluded that the first two of these limitations are bound to the way tasklets are integrated into iTasks, while the last one is a general limitation of the tasklet architecture. In the light of these limitations, we decided to refine the requirements and create a new component type to implement them.

5.4 Requirements

This is the extended list of requirements for our new component type, called editlet, derived from our previous experiments with tasklets:

1. It should be general enough that one can develop with it arbitrary browser applications. That basically means that editlets should have unlimited access to browser resources.

2. It should be developed in the single language Clean, no matter the platform it will run on.

3. It should be integrated with the type-driven approach used in iTasks to generate the user interface. An editlet should be registered in iTasks to be associated with a given type, such that it can be used by the system to replace the generic user interface. In this way, the user interfaces of the container types could be generated generically, while their elements may be customized with an editlet.

4. It should be able to work with shared data.

5. It should be able to detect and resolve conflict situations when working on shared data.

6. One should be able to develop such an editlet in such a way that one can minimize the communication overhead of the given editlet. We also require that the communication overhead related to the architecture of editlets should be as optimal as possible (e.g. the amount of the JavaScript code sent to the browser).
Editlets: type-based, client-side editors for iTasks

:: Editlet value = ∃ state edit:
{ genUI :: (*World → *(ComponentHTML edit state, *World))
  , appEditClt :: edit state *JSWorld → *(state, *JSWorld)
  , genEditSrv :: value value → Maybe edit
  , appEditSrv :: edit value → value
}

Figure 5.4: Editlet Interface Definition

7. Finally, as these requirements indicate additional steps in the building process, e.g. compiling to JavaScript, linking, etc., we want these steps to be fully integrated with the Clean toolchain, and therefore, to be completely transparent (c.q. invisible) for the developer and for the end users.

These requirements fall into three different groups. Requirements 1,2,7 put constraints on the definition of the client application and are basically already satisfied by tasklets. Requirements 3,4 impose restrictions on the way editlets are integrated into iTasks. Finally, requirements 5 and 6 affect the design of the communication interface.

In the next section we introduce our revised approach, and present how they meet these requirements.

5.5 Introduction to editlets

On an abstract level, editlets consist of three parts: (1) the type of the value the editlet produces, (2) a client-side application which is a stateful, event-driven application written in a single-language manner in Clean, and (3) a data synchronization interface which describes how to convert the state of the client application to the value of the edilet via edits and vice-versa.

On a more technical level, editlets basically consist of a set of functions which manipulate values of three types. These are the type of the value that the edilet is supposed to process on the server, the type of the state which is maintained by the client application, and, what is new compared to tasklets, the type of the data which is used for the client-server communication, dubbed edit.

5.5.1 The Edilet Interface

Figure 5.4 shows the type definition of the edilet interface. Editlets are defined by the means of the (Editlet value) record type, where the type parameter is the aforementioned value type. The other two types, the state of the client
application and the edit type, are of no concern for a programmer applying an editlet, and therefore hidden via existential quantification.

In the editlet interface one has to define four Clean functions: genUI which generates the application to run on the client, and the remaining to handle edits: appEditClt is executed on the client, while genEditSrv and appEditSrv run on the server. Client functions potentially alter the *JSWorld environment. The server function has access to the ‘regular’ *World environment of any side-effectfull program. The access to the *World enables the genUI function to do some IO during the initialization of the client application, e.g. for the purpose of templating. The *JSWorld environment is also used by the event handler functions to interface with the JavaScript foreign function interface explained in Section 5.6.2.

5.5.2 The Client Side Application

The genUI function produces the definition of the application to run on the client which is of type (ComponentHTML edit state) (see Figure 5.5). It is an event driven application which has an internal state of type state. The application basically consists of some HTML code that will be generated by Clean functions and a list of event handlers to handle the interaction with the end-user.

The actual user interface (html field) can be given by any data structure provided that it has an instance of the function class toHtml. In this paper we will use an overly simplified ADT to provide HTML definitions. In reality, one probably would like a more sophisticated way to have full, low-level control over the definition of HTML elements. This can be done in an abstract, monadic way like in Wash [124] or by an XML like domain specific language similar to that of Hop [116]. Furthermore, as the genUI function of the editlets is non-pure, it enables us to utilize some template mechanism similar to e.g. Yesod [118] or Snap [26]. However, providing any particular tool here is beyond the scope of this paper.

The run-time behavior of an editlet is encoded in a list of event handler functions given in the eventHandlers field. It is also possible to create and attach event handlers dynamically, but this facility is not important for the explanation. Event handlers are defined using the (ComponentEvent edit state) type. Its only data constructor has three arguments: the identifier of an HTML element, the name of the event and the event handler function itself. During the instantiation of the editlet on the client, the event handler function is attached to the given HTML element to catch events of the given name.

The event handler functions work on the JavaScript event object (a JSObj typed value in Clean) and the current internal state of the editlet. They can change the state by returning a new one, and they can also change the value associated with the editlet by returning an edit, wrapped in a value of type
:: ComponentHTML edit state =
{ html :: HtmlDef,
  eventHandlers :: [ComponentEvent edit state]
}

:: HtmlDef = ∃ a: HtmlDef a & toHtml a

:: ComponentEvent edit state
  = ComponentEvent HtmlElementId HtmlEventName
    (ComponentEventHandlerFunc edit state)

:: HtmlElementId == String
:: HtmlEventName == String

:: ComponentEventHandlerFunc edit state
  == (JSObj state *JSWorld →
      * (state, ComponentEdit edit state, *JSWorld))

:: ComponentEdit edit state
  == Edit edit (Conflict state *JSWorld →
       * (state, ComponentEdit edit state, *JSWorld))
    | NoEdit

:: Conflict == Bool

Figure 5.5: The ComponentHTML type

(_ComponentEdit edit state) (see Section 5.5.3). The event handlers also have access to the browser resources, e.g. HTML Document Object Model (DOM), to maintain their appearance for example, which is done through a foreign function interface (FFI).

From the point of event handlers, manipulating browser resources is a non-pure behavior. Therefore, FFI functions can be used as IO access is done in Clean, through uniquely attributed types. That is what the unique *JSWorld type is used for, in a similar way as the unique *World type is used on the server. Introducing a new type to have IO on the client has the advantage that it reflects for different purposes of client and server side code. The server code can access all resources of the server computer, like the file system, not available on the client; at the same time, the client code has external access to a resource accessible only on the client.
5.5.3 Synchronizing Clients and Server

The synchronization is based on the exchange of edit typed values, edits, between the clients and the server. The architecture is very similar to master/slave replication, where the server side shared data takes the role of the master, while the editlet instances in the browsers take the role of slave replicas. In such an architecture, updates are directly committed to the master to avoid distributed synchronization issues.

When a client wants to inform the server that something is changed which might have consequences for the value maintained by the server, it generates an edit which is sent to the server. The server applies the edit to its value, and if this value is shared, the same edit is distributed to the other tasks that share this value, in particular the editlet clients who depend on it. It sounds straightforward, but there are two serious issues we have to deal with in such a situation:

- First of all, the edit generated by the client, may no longer be valid and therefore cannot be used to calculate the new server value. This can happen if the value stored at the server is shared and the edit is created against a different (old) version of the underlying data. In the meantime the shared data might be changed by some other application, or some other editlet instance. Hence, one can have an update conflict, and editlets must be able to resolve this.

- Because update conflicts can happen on the server, the clients should commit updates in a synchronous manner to be sure that the update went through (and to try to resolve the conflict otherwise). However, there can be a significant network latency between the clients and the server and the lag could make the component highly unresponsive.

The first issue is solved by a standard technique of using a version attribute. The server side shared data has a version number which is increased every time the data is changed. When the server sends an edit to a client, it goes along with the current version of the shared data. Later on as the client generates edits, they are sent to the server along with this locally stored version number, so it can be compared with the actual version of the shared data on the server. If the numbers differ, a conflict is detected.

To solve the second issue we decided to use optimistic concurrency control [90]. Instead of sending the edits to the server in a synchronous manner, it is sent asynchronously and it is assumed that there will be no conflict. If, however, a conflict situation happens, the client is informed by a callback mechanism. In this way one can either roll back the change, or try again by sending a new edit to resolve the conflict.
In the editlet architecture, client-side edits are generated by the event handlers by means of the \((\text{ComponentEdit edit state})\) type (see Figure 5.5). An edit is provided along with a continuation function which is executed when this edit is accepted or rejected. In the latter case the first argument of the continuation function is set to \texttt{True} to indicate conflict.

The edits sent by the client to the server are applied to the server side value by the \texttt{appEditSrv} function that is used to calculate the new value. This function is, in contrast to its client-side counterpart, pure, as the only responsibility of the server is to hold pure data of type \texttt{value}.

In certain cases the server makes use of the \texttt{genEditSrv} function which is also defined in the editlet interface. It is used to find out what the difference is between an old and new value that is just calculated. This function is only used when the shared data is not changed by an editlet, but some other type of task or application. Assume e.g. that the shared data is stored in a shared file which is overwritten by someone else. In such a case there is no edit available to be distributed to the clients, it must be generated.

When the server has deduced that clients need to be informed about its changed value, it sends the corresponding edit to the client which calculates the consequences for its local state by applying the \texttt{appEditClt} function as defined in the editlet interface. It may not only cause a change to the state of the editlet. As it is non-pure, it can adjust its appearance according the received data as well.

To illustrate the role of these functions better, consider the following use case:

1. User interaction on the client triggers the execution of one of the event handlers of the editlet. New data is generated and an edit is created, a value of type \((\text{ComponentEdit edit state})\). The edit is sent to the server to synchronize with the server value. The logic of saving the new data in the client state (or roll it back) is encoded in the continuation function which is attached to the edit.

2. The edit, along with the local version number of the shared data, is sent to the server.

3. The server compares the version numbers. If they are the same, the edit is applied by the \texttt{appEditSrv} function, the version number is increased, a notification is sent to the client and the edit is distributed to the other instances of the editlet. If the version numbers differ, only a notification is sent to the client to be informed about the conflict.

4. At the client who transmitted the edit, the continuation function is executed to permanently commit or roll back the changes. As a reaction to
a conflict, it can decide to send a new edit and the workflow continues with Step 2.

5. At all the other clients, the edit sent by the server is applied by the appEditClt function.

The architecture guarantees that the edits and the conflict/success notifications are delivered to the clients in that order as they appear on the server. Based on this guarantee, it is assumed that the appEditClt and appEditSrv functions never fail. Working with version numbers and edits furthermore has an advantage that the synchronization between clients and server can be achieved with a minimal amount of communication.

5.5.4 Integrate with iTasks

Finally, iTasks must be made aware of the editlet. For this, one has to overwrite the default UI rendering logic of iTasks by providing an explicit instance of a generic function for the value type of the editlet. Section 5.6.1 demonstrates how to do that in practice.

5.6 Editlets by example

In the previous section we sketched out the big picture of the editlet architecture; in this section we show how it works in detail by developing a small, but illustrative example, where users can work together to draw an image.

\[
\text{workTogether :: User User a → Task a | iTask a}
\]

\[
\text{workTogether user1 user2 initialValue = withShared initialValue}
\]

\[
\lambda \text{shared} → \text{update user1 shared} -||- \text{update user2 shared}
\]

In this example we also want to explain how update conflicts are being handled. The general applicable task workTogether we use here is a variant of the updateAndView task as introduced in Section 5.2, where now both workers work on and update the same shared data such that update conflicts may arise.

\[
:: \text{Drawing} = \text{Drawing } [\text{Shape}]
\]

\[
:: \text{Shape} = \text{Line } \text{Color } \text{Int} \text{Int} \text{Int} \text{Int} \\
| \text{Rect } \text{Color Filled} \text{Int} \text{Int} \text{Int} \text{Int} \\
| \text{Circle } \text{Color Filled} \text{Int} \text{Int} \text{Int} \text{Int}
\]

\[
:: \text{Color} = \text{Yellow } | \text{Red } | \text{Green } | \text{Blue } | \text{Black}
\]
Now we are going to apply this general applicable task to a concrete type in a `drawingExample` where two workers, Alice and Bob, need to work together to produce a drawing of type `Drawing`. A drawing is simple and just consists of a list of shapes of lines, rectangles, and circles.

The `drawingExample` task can be executed as is, but would offer an interface to Alice and Bob for editing a list of shapes, similar to the list of persons interface we showed in Section 2. It would work and ensure that Alice and Bob can only create type correct drawing values, but our friends would certainly not be very happy with the look of the generated interface. We may assume that instead they would prefer to make drawings using some dedicated drawing application to produce drawings in a natural way. The behaviour should be such that whenever Alice draws something, Bob would see what has been drawn, and the other way around. When they are both drawing at the same time, we need to be able to handle both their contributions without upsetting them.

Changing the visualisation furthermore does not have any other consequences for the tasks being defined. The underlying technology remains completely hidden when type `Drawing` is used. Yet we want to obtain a nice drawing tool in the browser for free whenever a value of type `Drawing` is being used by some task. What does the iTask system developer have to do to make this possible?

### 5.6.1 Specialization

First, one has to overwrite the default GUI rendering for type `Drawing` of the iTask system and tell it to use an editlet for dealing with `Drawings` instead.

```haskell
:: Filled := Bool
drawingExample :: Task Drawing
drawingExample = workTogether "Alice" "Bob" []

5.6.1 Specialization

First, one has to overwrite the default GUI rendering for type `Drawing` of the iTask system and tell it to use an editlet for dealing with `Drawings` instead.

```
:: JSVal a // Pointer to a JavaScript value of some type
:: JSObject
:: JSObj:= JSVal (JSObject) // Unknown JavaScript value

// Convert foreign value to Clean dynamics
fromJSVal :: (JSVal a) *JSWorld → *(Dynamic, *JSWorld)

// Unsafe functions converting foreign values
jsValToString :: (JSVal a) → String
jsValToInt    :: (JSVal a) → Int

// Handling JavaScript objects:
class JSObjAttr a
instance JSObjAttr String
instance JSObjAttr Int

:: JSObjSelector

// Select an attribute of object
class (.#) infixl 3 s :: s t → JSObjSelector | JSObjAttr t
instance .# (JSVal o)
instance .# JSObjSelector
getElementById :: String → JSObjSelector

// Read an attribute, assign a value, call a function
.? :: JSObjSelector *JSWorld → *(JSVal r, *JSWorld)

(= ) infixl 2 :: JSObjSelector v → *(v, *JSWorld)
(.? ) infixl 1 :: JSObjSelector a
       → *(v, *JSWorld) | ToArgs a

// Drop return value of the application function
( .) infixl 1 :: o a → *(a, *JSWorld) | .? o λ& ToArgs a

Figure 5.6: The subset of JavaScript FFI used in the paper

This can be realized by defining an explicit instance of the generic gEditor function for type Drawing. Next we have to define all the components of the edilet record: the generator of the client-side painter application painterGUI (see Section 5.6.3) and the three functions to handle the synchronization between the clients and the server (updateClient, calculateEditsServer, and updateServer, see Section 5.6.4).
5.6.2 The Foreign Function Interface to JavaScript

Before we can explain how the painting application can be defined, we need to explain how the FFI (Foreign Function Interface) to JavaScript and to the DOM of the browser is offered.

Figure 5.6 shows a subset of types and functions of the iTasks JavaScript FFI which is used in this example. The FFI can be split into two groups. The first group deals with JavaScript values. For this purpose we use the \((\mathit{JSVal\, a})\) type. This type is a pointer to a JavaScript value, which real type is \textit{unknown}. Its type parameter is just a phantom type used in higher order APIs. However, we use a special technique to be able to deal with JavaScript values in a type safe manner. This technique is based on overloading the built-in dynamic type system of Clean to generate type information for a given JavaScript value [41]. This type information can be used to pattern match on the actual type of the value. The type information is generated by the \texttt{fromJSVal} function, but, based on this function, unsafe instances are also available to be able to write more concise code in case the type of the value is known reliably.

In the second group there are functions to deal with attributes and methods of objects. These are accessing members of objects (\(\mathit{\.\#}\)), accessing elements of the DOM (\texttt{getElementById}), reading an attribute of an object (\(\mathit{\.\?}\)), assigning value to an attribute of an object (\(\mathit{\.\=}\)) and calling a method of an object (\(\mathit{\.\$\?, \., \$.}\)).

To illustrate the usage of the interface, we can already define some utility functions to deal with a HTML5 canvas element which is used by our component to draw graphics. Its \texttt{getContext} method returns a rendering \textit{context}, a built-in HTML5 object, with many properties and methods for drawing paths, boxes, circles, text, images, and more. The definition of some of the utility functions are neglected as they are straightforward to implement based on the others:

```plaintext
// Context provides methods and properties for drawing on the canvas
:: JSCanvasContext
:: Context := JSVal JSCanvasContext

// Return the context of a canvas given by HTML id
getContext :: String *JSWorld → *(Context, *JSWorld)
getContext canvasId world
  = (getElementById canvasId .λ "getContext" .? ("2d")) world

// Clear a canvas for redrawing
clearCanvas :: Context *JSWorld → *JSWorld

drawRect :: Context Color Filled Int Int Int Int *JSWorld → *JSWorld
drawCircle :: Context Color Filled Int Int Int Int *JSWorld → *JSWorld
```

95
Chapter 5

// Draw a line using the given color, coordinates and context
drawLine :: Context Color Int Int Int Int *JSWorld → *JSWorld
drawLine context color x1 y1 x2 y2 world
λ world = (context .λ "beginPath" . () ) world
λ world = (context .λ "strokeStyle" .= color) world
λ world = (context .λ "moveTo" . (x1, y1)) world
λ world = (context .λ "lineTo" . (x2, y2) ) world
λ world = (context .λ "stroke" . () ) world
= world

// Draw an arbitrary shape using the specialized functions
draw :: Context Shape *JSWorld → *JSWorld

These functions are intuitive to read and write as their structure mimics the
structure of their imperative counterparts. The names of the operators are
also chosen carefully that the individual calls resemble to the corresponding
JavaScript commands. Still, the functions are written in the single-language
manner in Clean enabling access to all the features of a pure, lazy functional
language.

5.6.3 The Painting Application

The application we want to create on the client is shown in Figure 5.7. Below
the actual canvas, the user can choose the tool from a drop-down list, and the
current color can be chosen by clicking on one of the small boxes next to the
canvas.

To keep the example illustrative, it is not allowed to delete or modify an
already drawn shape. That would add much complexity to both the user inter-
face and the synchronization parts, thus hampering the comprehension of the
main idea. Handling conflict situations, depending on the actual task, can be
arbitrarily hard, and there is no “proper” logic to do it. Editlets offer a general
mechanism to build any customized conflict resolution logic.

Since in the editor on the client one can only add, but cannot remove or
modify shapes, the incremental updates can be given by the list of newly added
shapes. Therefore, [Shape] is used as the edit type for this application.

The following record type is used as the state of the client-application:

:: PainterState = { selectedTool :: Tool
, selectedColor :: Color
, currentOrigin :: Maybe (Int, Int)
, currentShape :: Maybe Shape
}

:: Tool = TLine | TRect | TRectF | TCircle | TCircleF
:: Color = Yellow | Red | Green | Blue | Black

96
The selectedTool and selectedColor fields contain the currently selected tool and color, respectively. For understanding the remaining two fields, currentOrigin and currentShape, we need to explain how the drawing of a given shape takes place.

A very simple approach could be to ask the user to select two points on the canvas and the shape is drawn between these points. It is very tempting to follow this approach, mainly because of its simplicity. However, in the same time, it would render our component completely unusable, as it is hard to draw images for a human without constant visual feedback. Thus, the usual approach is taken: drawing of a shape starts when the user presses the left mouse button, and ends when it is released. As long as the left button is pressed, while the user moves the mouse, the current appearance of the shape is continuously updated on the screen.

Therefore, the currentOrigin field contains the start coordinates of the shape being drawn, while the currentShape contains the drawn shape itself. They only contain an actual value while the left mouse button is pressed.

Finally, some explanation on the implementation of this mechanism. In JavaScript, it can most easily be done by using a temporary drawing canvas to overlay a permanent canvas. The permanent canvas contains all the shapes which are already “committed”, while the shape being drawn is put on the
temporary canvas, which can be cleared and redrawn as it contains only that one. When the user releases the mouse button, the final shape is copied to the permanent canvas.

The actual user interface is created by the painterGUI function. It generates the two canvases discussed before, small boxes as HTML DIV elements for the color selectors to the right of the canvases, and a drop down list just under the canvases for choosing the current tool (please note that the code below is slightly simplified for presentation purposes):

\[
\text{painterGUI :: ComponentHTML [Shape] PainterState} \\
\text{painterGUI = \{ html = DivTag [] [canvases:editor]} \\
\text{, eventHandlers = eventHandlers} \\
\text{\}} \\
\text{where} \\
\text{canvases =} \\
\text{DivTag [StyleAttr "position: relative; float: left;"] [} \\
\text{CanvasTag [IdAttr "pcanvas"] []}, \\
\text{CanvasTag [IdAttr "tcanvas",} \\
\text{StyleAttr "position: absolute;"] []} \\
\text{]} \\
\text{editor = [} \\
\text{DivTag [StyleAttr "float: right;"] (map selector colors),} \\
\text{DivTag [StyleAttr "clear: both;"] []}, \\
\text{DivTag [] [Text "Tool:"}, \\
\text{SelectTag [IdAttr "tool"] (map tag tools)} \\
\text{]} \\
\text{tools = [TLine, TRect, TRectF, TCircle, TCircleF]} \\
\text{colors = [Yellow, Red, Green, Blue, Black]} \\
\text{tag tool = let text = toString tool} \\
\text{in OptionTag [ValueAttr text] [Text text]} \\
\text{selector color = DivTag [IdAttr (mkId color),} \\
\text{StyleAttr ("background-color:" ++ toString color ++ ";") []} \\
\text{colorEvent color} \\
\text{= ComponentEvent (mkId color) "click" (onSelectColor color)} \\
\text{eventHandlers = map colorEvent colors ++ [} \\
\text{ComponentEvent "tcanvas" "mousedown" onMouseDown,} \\
\text{ComponentEvent "tcanvas" "mouseup" onMouseUp,} \\
\text{ComponentEvent "tcanvas" "mousemove" onMouseMove,} \\
\text{ComponentEvent "tool" "change" onChangeTool} \\
\text{]} \\
\text{98}
The list of event handlers is associated with the elements of the user interface: one for each of the color selectors, one for the drop down list of the available tools, and one for each of the mouse events we use to handle drawing.

As the user clicks on one of the color selectors, the border of the selected color box is highlighted and the state is changed according to the new selection, but no edit is generated. The selected color is in the first argument of the event handler as partially applied functions are used for this purpose:

```
:: PaintEventHnd :== JSObj PainterState *JSWorld → *(PainterState, ComponentDiff [Shape] PainterState, *JSWorld)
```

```
onSelectColor :: Color → PaintEventHnd
onSelectColor color e state world
  # world = foldr (setBorder "white") world allBoxes
  # world = setBorder "pink" (e .# "target") world
  = (\{state & selectedColor = color\}, NoEdit, world)
```

When the tool is changed, we just read the new identifier and set it in the state:

```
onChangeTool :: PaintEventHnd
onChangeTool e state world
  # (idx, world) = .? (e .# "target" .# "selectedIndex") world
  # (os, world) = .? (e .# "target" .# "options") world
  # (tool, world) = .? (os .# jsValToInt idx .# "value") world
  = (\{state & selectedTool = fromString (jsValToString tool)\}, NoEdit, world)
```

The drawing begins when the user pushes the mouse button. Then we read the current position of the mouse and set it in the state to indicate that drawing is in progress:

```
getCoordinates :: JSObj *JSWorld → *((Int, Int), *JSWorld)
getCoordinates e world
  # (x, world) = .? (e .# "layerX") world
  # (y, world) = .? (e .# "layerY") world
  = ((jsValToInt x, jsValToInt y), world)
```

The actual drawing happens when the mouse is moved while the mouse button is pressed. Thus, in the onMouseMove event handler function, first the presence of the currentOrigin coordinates must be checked:
onMouseMove :: PaintEventHnd
onMouseMove e state world = case state.currentOrigin of
  Just coordinates = onDrawing coordinates e state world
  Nothing = (state, NoEdit, world)

If they are set, then using these coordinates and the current coordinates of the mouse, along with the current color and tool, we can create a shape to draw it to the temporary canvas (which is cleared before that). Finally, the shape is saved in the state as it will be needed to finalize the drawing:

onDrawing :: (Int, Int) → PaintEventHnd
onDrawing (ox, oy) e state world
  # ((x, y), world) = getCoordinates e world
  # shape = case state.selectedTool of
    TLine  = Line state.selectedColor ox oy x y
    TRect  = Rect state.selectedColor False ox oy x y
    TRectF = Rect state.selectedColor True  ox oy x y
    TCircle = Circle state.selectedColor False ox oy x y
    TCircleF = Circle state.selectedColor True  ox oy x y

  # (tempcontext, world) = getContext "tcanvas" world
  # world = clearCanvas tempcontext world
  # world = draw tempcontext currentShape world
  = ({state & currentShape = Just shape}, NoEdit, world)

The drawing is finalized when the mouse button is released: the temporary canvas is cleared and the shape saved by onDrawing is copied to the permanent canvas:

onMouseUp :: PaintEventHnd
onMouseUp e state world
  # (tempcontext, world) = getContext "tcanvas" world
  # world = clearCanvas tempcontext world
  # (edit, world) = case state.currentShape of
    Just shape
      # (context, world) = getContext "pcanvas" world
      = (addShape shape, draw context shape world)
    Nothing
      = (NoEdit, world)
  = ({state & currentOrigin = Nothing, currentShape = Nothing}, edit, world)

This is the only point of user interaction when the value associated with the editlet can change: when the user releases the mouse button, and the mouse moved since the button was pressed, that is the currentState field of the state contains a shape. In this case the shape is copied to the permanent canvas and an edit is generated:
Editlets: type-based, client-side editors for iTasks

addShape :: Shape \rightarrow \text{ComponentEdit} [\text{Shape}] \text{PainterState}
addShape shape = Edit [shape] callback

where
  callback True state world
    # (context, world) = getContext "pcanvas" world
    = (state, addShape shape, draw context shape world)
  callback False state world
    = (state, NoEdit, world)

As it is explained in Section 5.5, the client part of the editlet applies edits to the shared server value in an asynchronous manner. There is a continuation function associated with the edits which is executed when the edit is finally applied or rejected. In this particular example, we take a highly optimistic approach: the new shape is drawn to the canvas in the same time when the edit is created. Later on, if the edit is rejected, the shape is drawn to the canvas again (to be the top most shape on the canvas again) and the same edit is tried again. If it is accepted we are fine, the shape is already the top most on the canvas.

When the notification of the rejection of a previous edit is triggered on the client, the client state had already been synchronized with the server value (the edit(s) causing the conflict on the server are applied to the client state). Further considering that an edit, in this particular case, can describe the addition of new shapes only (thus does not depend on the current value to be applied to), it is safe to reapply the rejected edit at that point.

5.6.4 Synchronization Functions

So far, we have a function to generate the user interface (the painterGUI function), which also describes when and how to generate edits on the client. To finish our editlet, we need to provide the rest of the synchronization interface. These are the functions for the genEditSrv (the function updateClient), appEditSrv (the function calculateEditsServer) and appEditClt (the function updateServer) fields of the (Editlet value) record type.

updateClient :: [Shape] PainterState \times \text{JSWorld} \rightarrow \times (\text{PainterState}, \times \text{JSWorld})
updateClient edit state world
  # (context, world) = getContext "pcanvas" world
  = (state, foldl (\(\text{world } s = \text{draw context } s \text{ world}) \text{ world edit})

calculateEditServer :: \text{Drawing} \times \text{Drawing} \rightarrow \text{Maybe} [\text{Shape}]
calculateEditServer (\text{Drawing oldss}) (\text{Drawing newss})
  = \text{case drop} (\text{length oldss}) \text{ newss of}
    [[] \times \text{Nothing}
     edit = \text{Just edit}
updateServer :: [Shape] Drawing → Drawing
updateServer ns (Drawing ds) = Drawing (ds ++ ns)

In this particular case the state of the client-application, the PainterState does not actually need to store a client-side counterpart of the server value. We do not need to traverse that data any time on the client, only have to draw the shapes to the canvas in the correct order. It is guaranteed that edits are delivered in the proper order to the clients. Hence the updateClient function just updates the user interface by drawing the new shapes to the canvas, and does not modify the state value.

On the server, an edit sent by one of the clients, a list of new shapes being added, is handled by the function updateServer. It just appends the list of new shapes to the shapes that already have been drawn and collected on the server in the value of type Drawing.

Finally, the calculateEditServer function also exploits the fact that the original drawing cannot be modified: it calculates the difference between an old and a new drawing by determining the number of shapes that might have been added.

5.7 The architecture of client-side execution

A crucial point of a single-language solution is the way the JavaScript code is produced and handled. In a single-language setting, client and server code is mingled, and, unless there is special syntactic construction introduced in the language for indicating which code is intended for client or server, the code cannot be separated during compilation. This can have the consequence that the whole application, including the code which is only relevant to the server, is compiled to JavaScript and shipped to the browser. This results in an explosion of code that not only causes huge communication overhead, but also a waste of browser resources. From security perspective, shipping unnecessary cross-compiled server code to the client, would also expose the structure of the server, which can help to reveal any potential weaknesses.

To overcome this issue in iTasks, we developed a special JavaScript compilation technique integrated with the Clean language. The compilation technique has four key components: (1) the SAPL [79] language, (2) a compiler extension, (3) run-time support, and (4) the SAPL compiler infrastructure [38], which is a library to handle SAPL source code. This library supports low level functions e.g. parsing SAPL source code, program transformations, and it also provides high level functionality, e.g. linking. A full-featured SAPL to JavaScript compiler [39] is also implemented.

The first component, the SAPL language, is an intermediate language designed to contain only the essential minimum of language features of a lazy,
Editlets: type-based, client-side editors for iTasks

functional language like Clean or Haskell, while preserving the semantics. Furthermore, its syntax is carefully constructed such that it can be easily handled at source code level. These properties makes it perfect for efficient source code level linking, and for fast cross compilation as a source language.

The second component is a Clean compiler extension. During the compilation of an iTasks application, besides the server executable running in native code, an intermediate representation of the same application in the SAPL language is also created. This extension is seamlessly integrated with the Clean compiler, it just transparently creates a directory, along with the native binary, which contains all the SAPL source code.

The third component is the run-time support. During the execution of an application, the Clean run-time can be asked to provide the SAPL source code needed for the evaluation of an arbitrary Clean expression.

Using the SAPL expression, the last component, the SAPL library, is utilized to recursively collect the SAPL code the expression depends on, using the SAPL source code of the application. Then, the collected SAPL source is ran through an iTasks specific, per client caching mechanism. Its task is to filter out the functions which have already been sent to a given client in the corresponding session. The SAPL functions which are not yet on the client, are on the fly compiled to JavaScript and shipped to the browser.

This overall architecture enables to reduce the communication cost to the potential minimum and to preserve as much browser resources as possible.

5.8 Related work

There are three main groups of works that are closely related to editlets. These are the multi-tier programming languages, including our previous approach with tasklets, JavaScript cross-compilers, and the theory of change based bidirectional transformations.

Multi-tier programming languages

Several other languages address multi-tier programming. In the imperative world the most modern approach is the Google Web Toolkit (GWT) [66], Google Dart [33] and Node.js [122]. GWT utilizes a Java to JavaScript compilation technique for building complex browser-based applications. GWT fosters classical GUI programming where widgets can be developed using a programming model comparable to that of editlets.

The Dart language and the Node.js framework take a different approach. They enable multi-tier programming by providing a run-time environment of their languages for both client and server side. The language of Node.js is JavaScript, which is native in the web browsers; the framework also provides
a run-time environment, including IO libraries, for the server side. Dart is a programming language developed by Google specially designed for web application engineering. On the client, it compiles to JavaScript, on the server it is executed by a Dart virtual machine.

These aforementioned systems have a more general approach than iTasks and editlets, but they still share the idea of using the same language on both client and server side and implicitly bridging the communication between them.

Hop [116, 117] uses a declarative approach. It is a dedicated web programming language with a HTML-like syntax built on top of Scheme. Hop uses two compilers, one for compiling the server side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. Hop uses syntactic constructions for indicating client and server part code. The application essentially runs on the client and may call services on the server. In contrast, an iTasks application essentially runs on the server and may execute services, editlets, on the client.

Links [28] and its extension Formlets is also a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. In a Links program, the keywords client and server force a top-level function to be executed at the client or server respectively.

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

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

As for iTasks, we already compared our previous approach, tasklets, to editlets in Section 5.3. However, there are also earlier implementations of similar features utilizing a Java written SAPL interpreter [79] as a browser plug-in. The iEditors [80] enables the development of interactive web UI elements as editlets do, however, it does not allow direct access to browser resources, therefore its applicability is restricted to functionality provided by the plug-in. As a consequence, it does not have the single-language property either, because for some functionality the plug-in has to be extended using Java. There also had been client-side task evaluation attempts for an early version of iTasks using the same plug-in based interpretation technology [112]. However, our
approach, to give one general solution for both of the problems is a novel strategy.

**JavaScript cross-compilers**

JavaScript cross-compilation is a subject that has drawn much attention in the last years as web applications getting richer and richer to improve the web experience. Virtually every modern programming language has at least one JavaScript cross-compiler, thus we limit ourselves to the comparison of some very closely related technologies and concentrate on the high level architecture only (an explanation of the compiler can be found in [39]).

The most relevant technologies are the aforementioned Links and Hop languages. Both languages are functional just like Clean, however unlike these languages, Clean is a **lazy** functional language. Although this property has a big impact on the actual compilation technique, it only slightly affects the high level architecture. The main difference between them is rather that these languages are specially designed as multi-tier. This has a consequence that the client and server side code is distinguished by special syntactic constructs at source code level. Thus, the client and server side code can be separated during compilation time. However, this does not take into consideration the dynamic behavior of the application. In our architecture, only those functions are shipped to the browser which are actually requested at **run-time**; the difference can be significant in some executions.

Another relevant technology is GHCJS [63], the most advanced JavaScript cross-compiler for the Glasgow Haskell Compiler (GHC). As Haskell and Clean are from the same family of the functional languages, and they are actually very similar in many viewpoints, it is worthwhile to compare the architecture of its flagship JavaScript cross-compiler and the architecture of our cross-compiler.

In contrast to our solution, GHCJS takes a module based approach. During compilation, along with the object files, the JavaScript version of the modules are also generated. The final client-side “executable” is produced by simply combining the JavaScript versions of all the referenced modules together. Although this approach definitely has the advantage that no runtime component is necessary in the architecture, it also suffers from producing an explosion of code: Haskell applications tend to use many libraries and many generically created functions which are blindly merged resulting in an enormous amount of JavaScript code.

Although most of the cross-compilers use a kind of simplified core language as the source of the compilation, the idea to use this intermediate core language for linking as well, thus reducing the size of the output, is a novel idea as far as we know.
Change based bidirectional transformations

In general, bidirectional transformations are a mechanism for maintaining the consistency of two related sources of information [31]. In the field of bidirectional transformations, the most closely related work is the so called bidirectional lenses [58]. Within lenses, edit lenses [75, 131] bear the most resemblance to our data synchronization interface.

In a nutshell, edit lenses define bidirectional transformations between pairs of connected structures, where each of the two structures may contain information that is not present in the other (also known as symmetric lenses [74]). Moreover, edit lenses work with descriptions of changes to structures, rather than with the structures themselves. In practice that means that with each of the connected structures there is an additional associated data structure, an edit language, to define the changes of the original structure. The actual changes, the edits, of the structures are converted to each other by a bidirectional transformation, the edit lens, in a stateful manner.

It has two main differences comparing with editlets: (1) our case is not symmetric in the sense that the value held by the server does not contain information that is not synchronized with the clients; (2) there is only one edit language.

However, from another perspective, these are not important differences. If the lens is applied on the client (in the appEditClt function), which is stateful, then for the external observer the state of the edit lens and one of edit languages is hidden. This means that the necessary synchronization functions of an editlet can be easily defined based on an existing edit lens implementation.

Finally, another group of works addresses the generation of edits [132, 36, 64]. Although these are interesting for our case, adapting such a framework is beyond the scope of this paper.

5.9 Conclusion and future work

In this paper we have presented an extension to iTasks, for the development of interactive web components in a single-language manner. This extension is based on our previous experiment in the same topic, called tasklet. We have identified several shortcomings of this previous approach based on experiments with real-world applications, what we used to set new requirements. Based on this new set of requirements, we have designed a new component type, called editlet, which is superior to tasklets from many perspectives. The main properties of the editlet architecture are the following:

- Editlets enable the development of arbitrary browser applications in the single language Clean.
Editlets are integrated with the type-driven approach used in iTasks to generate the user interface. The default behavior of the user interface generation can be overwritten by registering an editlet to be associated with a given type.

Editlets can work on shared data, moreover, it is done in a way which enables the developer to deal with conflicting updates efficiently in an asynchronous manner to keep the user interface responsive.

The client-server communication is done in changes, called edits, instead of exchanging the whole shared value. In certain applications, it dramatically reduces the communication cost.

Editlets are based on an advanced client-side execution architecture to reduce the amount of generated JavaScript code to the minimal possible. The JavaScript cross-compiler is integrated transparently in the Clean tool chain.

Although editlets is an iTasks extension, these properties make it interesting in a global perspective. The client-side execution architecture, the edit based communication interface and the type-based approach are all interesting properties in their own right, and can be integrated with other languages and frameworks.

As for the future work, we are planning two major tasks. First we would like to upgrade iTasks SDSs to be based on edits as well. The edit type would be attached to the value type by functional dependencies to be global for the whole system, and the SDSs would be updated by edits even on the server. In this way we could get rid of the genEditSrv function from the editlet definition.

The other task affects the JavaScript cross-compiler. Currently it is not safe in the sense that during code generation some sensible data may be shipped to untrustworthy clients. We would like to overcome this issue by investing in techniques similar to those developed for the Links language [16].
Part III

Applications
Clean up the web! – Rapid client-side web development with Clean

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

6.1 Introduction

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

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

The main problem is the limitation of the available resources in the browser: the run-time system imposes severe constraints on heap and stack usage. As iteration in functional languages is accomplished via recursion, stack limitation seems to be the most serious issue. A standard technique to overcome this is
trampolining [123], but, as it increases the memory footprint and the running
time of the application, usually it does not perform effectively enough in the
case of lazy functional languages. The reason for the higher memory footprint
in these languages is the need to maintain thunks, i.e. delayed computations.

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

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

The rest of the paper is structured as follows. Section 2 gives a brief intro-
duction to the iTask system and Tasklets. Section 3 presents what contribu-
tions the present paper makes. To that end, it illustrates the approach and some
of the technical issues through three carefully selected examples. Section 4
discusses type correspondence between the JavaScript and Clean side of the
code. Related work is described in section 5. Finally, section 6 concludes. The
system as well the examples presented here can be downloaded from the web.\footnote{http://people.inf.elte.hu/dlacko/papers/rapmix/}

\section{Preliminaries}

The iTask system [95] is a framework for programming workflow supporting
applications in Clean using a new programming paradigm built around the
concept of \textit{tasks} [113]. A task is an abstract description of an interactive
persistent unit of work which delivers a value when it is executed. From a
practical point of view, a task can be anything from a system call to some
interaction to be performed in a web browser by a user.

iTask provides a combinator-based embedded domain specific language to
specify compositions of such interdependent tasks. A complete multi-user web
application can be generated from the specification of the workflow and of the
Clean up the web! – Rapid client-side web development with Clean
different data types involved – all the details (including the web user interface,
client-server communication, state management etc.) are automatically taken
care of by the framework itself.

Developing web applications such a way is straightforward in the sense
that the programmers are liberated from these cumbersome and error-prone
jobs, such that they can concentrate on the essence of the application. The
iTask system makes it very easy to develop interactive multi-user applications.
The down side is that one has only limited control over the customization of
the generated user interface. Sometimes, even if the functional web design is
satisfactory, custom building blocks may be required for the purpose of user-
friendliness.

Tasklets, a recent extension to iTask, are introduced to overcome this short-
coming [45]. Tasklets enable the development of interactive web components
directly in Clean. A tasklet consists of an inner state, user interface, and be-
behavior provided by non-pure event handler functions. The user interface can
be defined in any abstract or concrete way that enables HTML code gener-
ation. The event handlers are written in Clean, but compiled to JavaScript
and executed in the browser where they have unrestricted access to client-side
resources. Using browser resources, the tasklet can create custom appearance
and exploit functionality available only in the browser; utilizing the event-
driven architecture the tasklet can achieve interactive behavior.

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

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

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

:: TaskletGUI st = TaskletHTML (TaskletHTML st) | ...
:: TaskletHTML st = { html :: HtmlDef
, eventHandlers :: [HtmlEvent st]
}
:: HtmlDef = ∃ₐ: HtmlDef a & toHtml a

:: HtmlEvent st = HtmlEvent HtmlElementId EventType (EventHandlerFunc st)
:: EventType = OnClick | OnMouseOver | OnMouseOut | ...
The actual user interface (html field) can be given by any data structure provided that it has an instance of the function class toHtml.

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

The event handler functions work on the JavaScript event object (a value of type HtmlObject in Clean) and on the current internal state of the tasklet. They also have access to the HTML Document Object Model (DOM) to maintain their appearance. The DOM is a shared object from the point of event handlers, therefore it can be manipulated only the way as IO is done in Clean, through unique types. That is, accessing the DOM is possible only using library functions controlled by the unique *HtmlDocument type.

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

\[
\text{mkTask} :: (\text{Tasklet st a}) \rightarrow \text{Task a}
\]

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

### 6.3 Rapid development with iTask

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

\[
\text{Start} :: \ast\text{World} \rightarrow \ast\text{World}
\]

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

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

```javascript
Start world = startEngine |
| url = "/test"
, task = TaskWrapper (const testTasklet)
, defaultFormat = JSONGui|
world
```

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

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

```html
<html>
<head>
    <script type="text/javascript" src="tasklet−runtime.js"/>
</head>
<body>
    <div id="out"/>
</body>
</html>
```

The JavaScript library `tasklet-runtime.js` is less than 10 kB compressed. It contains the logic for loading and instantiating tasklets, as well as the run-time environment of the Clean to JavaScript compiler. Function `loadTasklet` tries to load a tasklet from the URL given in its first argument. Since the loading mechanism is implemented with an asynchronous AJAX request, a call-back function must also be provided as a second argument; this will be called when the tasklet is loaded and created in the browser.
An instantiated tasklet is represented as a JavaScript object with the pre-defined prototype Tasklet. It encapsulates and hides all the properties of a tasklet (the user interface, the state and the behavior), and exposes only the display method which injects the UI of the tasklet into a given point in the HTML DOM.

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

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

\[
\text{initially :: st} \rightarrow \text{Tasklet st Void}
\]
\[
\text{initially st =}\{\text{generatorFunc} = \lambda \text{world} = (\text{NoGUI, st, world})
\]
\[
, \text{resultFunc} = \text{const Nothing}\}
\]
6.3.1 Writing the logic of a web application in Clean

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

The Clean to JavaScript compiler generated 136 kB of JavaScript from the 33 kB of Clean code, and the source code viewer was up and running. Now we take a closer look at the code.

```clean
:: Color ::= String

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

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

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

world
```

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

In the case of this simple tasklet, not only the GUI and the result value, but also the internal state is absent, i.e. Void. The only way to interact with the highlighter tasklet is to call its single interface function, annotateI, from the JavaScript code. When a tasklet is created with mkInterfaceTask (defined in the tasklet library), a list of interface functions can be passed. In this case, this list has a single entry: whenever the JavaScript code calls the annotate method of the tasklet, the code generated from the annotateI function is executed. This annotateI takes the current (Void) state of the tasklet and an event queue (explained in section 6.3.3), and returns them unmodified. Information from JavaScript to Clean is received through the second parameter,
which is of type `Maybe Dynamic`. Dynamics provide dynamic typing facilities in a statically typed language [7, 109].) We expect here that a list of strings is stored in the `Dynamic`, namely the lines of some Clean source code. If the dynamic pattern matching fails, the run-time engine triggers an exception to inform the caller. The `highlight` function will be called with the lines found in the dynamic: it splits each line into tokens, and annotates each token with its colour. The token and its colour is represented as a pair of strings, a list of pairs corresponds to a line, and the list of lists is the whole program text syntax highlighted. This list of lists of pairs of strings is sent back to the JavaScript side as a component of the triple returned by `annotateI`.

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

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

    for(var i=0; i<tokens.length; i++){
        for(var j=0; j<tokens[i].length; j++){
            var token = tokens[i][j][0];
            var color = tokens[i][j][1];
            appendToken(token, color);
        }
        appendNewLine();
    }
}
loadTasklet("http://localhost/highlighter", onLoadTasklet);
```

When the page is loaded, the function `loadTasklet` is executed by the browser. The tasklet is loaded from the specified URL, instantiated, and `onLoadTasklet` is called with it. This latter function first creates an array of strings (i.e. `lines`), which is passed to the `annotate` interface function of the tasklet (the interface functions are created under the `intf` namespace to avoid possible name collisions with the original properties of the Tasklet prototype). Note that this array of strings corresponds to a `Dynamic` containing a list of strings in the Clean side (the details of the type correspondence algorithm are explained in section 6.4). Function `annotate` returns an array of arrays of strings, `tokens`, which is processed in a straightforward way in the `for`-loop.

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

### 6.3.2 Adding state and interaction

Suppose you must write some interactive presentation logic to be executed in a browser. For example, you want to display the bibliographic data of your publications in a searchable, filterable way on the web (Fig. 6.1). The application should receive a BibTeX file as input, and parse, filter and pretty-print the entries found in this file. To write a client-server application for this, and implement parsing and filtering on the server would be too much hassle. It is more reasonable to send over the data to the browser all at once, parse it, and then let an interactive client side application filter the data and display the selected items. Coding all these activities in JavaScript is not what you would like to do on a rainy Friday afternoon. Contrarily, much of the functionality is fairly straightforward to develop in Clean, using higher order functions. To implement parsing, for instance, the Parser Combinator library of Clean may prove useful. It turns out that tasklets are a valuable tool for building this application.

The main difference between this and the syntax highlighter application is that interaction with the user is required, and that there is some state that

![Figure 6.1: Web application for filtering bibliographic data](image)
Chapter 6

should be preserved between user interactions. We suggest that the state should be stored in the JavaScript side of the code, and state-to-state functions should be written in Clean. The following fragments present the interesting parts from the JavaScript side of the code.

```javascript
var entries;
var tasklet;

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

    var refs = prepareReferences();

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

    display_bibitems(entries);
}
```

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

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

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

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

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

    display_bibitems(selected);
}
```

Similarly to the syntax highlighter, this tasklet is also stateless and provides no GUI. It does not make use of `eventqueue` either.
The interface functions of the tasklet have a similar structure to that of `annotateI` in the previous example. The only argument they use is the one of type `Maybe Dynamic`, on which they pattern match. The `filter` method calls in the JavaScript code, for instance, has three actual arguments, therefore the dynamic in the corresponding Clean function, `filterI`, should be a triple.

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

Section 6.4 will explain why `res`, the result from filtering is wrapped in a `dynamic`.

### 6.3.3 Even more state and even more interaction

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

The level of interactivity is much higher in this example than in the previous one. The Rubik cube is controlled by moving the mouse and by pressing some
keys; the cube is rendered (Fig. 6.2) on-the-fly by the Clean side of the code when its state is changed. Another interesting issue in this example is how information flows between the pure Clean side of the code and the impure JavaScript side. Although the tasklet depends on information from the outer environment (the browser), and has an impact on its environment as well, referential transparency is not violated.

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

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

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

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

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

\[
\text{standard} \ (x, y, z) \\
| \text{abs } x > \text{abs } y \land \text{abs } x > \text{abs } z = \text{if } (x < 0.0) \ "green" \ "blue" \\
| \text{abs } y > \text{abs } x \land \text{abs } y > \text{abs } z = \text{if } (y < 0.0) \ "yellow" \ "white" \\
| \text{otherwise} = \text{if } (z < 0.0) \ "orange" \ "red"
\]

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

```clean
tasklet.addListener("draw", function(event){
  var v = event.value;
  var color = v[1];
  var p1 = v[0][0];
  ...
  drawPolygon(p1,p2,p3,p4,color);
})
```

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

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

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

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

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

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

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

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

³ http://people.inf.elte.hu/dlacko/papers/rapmix/rubiksource.html
6.4 Type correspondence in parameter passing

The communication between the JavaScript side and the Clean side of the code is bidirectional. The JavaScript side calls the interface functions of tasklets, passing arguments and expecting results. Moreover, the Clean side fires events, with parameters attached, and the JavaScript side may observe these events and receives their attached parameters. In both cases information exchange between the two sides is achieved through pass-by-value parameters and, in the first case, through pass-by-value return values. The proper transmission of data requires a consequent correspondence between Clean types and JavaScript types. Certain types carry over between the two languages quite straightforwardly, others need special encoding.

It must be emphasized, however, that when we talk about the Clean side of the code, we actually mean some JavaScript code that was generated from Clean code by our cross-compiler. For clarity, we will refer here to the JavaScript side of the application as JS code, and to the code generated from the Clean side as JS* code. JS* uses a special run-time encoding of Clean types. For details on this encoding, the Reader is referred to [39].

To facilitate information exchange between Clean and JavaScript, a conversion from the JS* encoded values to JS is provided. The programmer could use the JS* encoded values in the JS code directly, but the structure of the encoded values is quite unnatural. Therefore, our runtime environment converts JS* values to JS values that are easier to use. As the examples in section 6.3 revealed, (1) during conversion primitive types are preserved; (2) the encoding of lists and tuples of Clean are converted to arrays; (3) algebraic types are also represented by arrays, where the name of a data constructor is stored in the first element of such an array. The conversion of functions in JS* to JS is not supported in the current version of the system. Handling partially applied functions and lazy arguments would demand special care of these values on the JS side, which, in our opinion, is not worth the effort.

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

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

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

1. JS booleans can be unified with Clean `Bools`.

2. Although JS has no special character type, strings of one length can be unified with `Char` in Clean.

3. There are no separate integer and floating point types in JS, a JS integer value can be unified with both `Int` and `Real` in Clean.

4. Non-integer numbers can be unified with Clean `Reals` only.

5. JS strings can be unified with the `String` type of Clean.

6. An array of JS values can be unified with a Clean list type, if
   
   (a) all of its elements can be determined by the preceding rules,
   
   (b) they have the same type, and

   (c) this type is equivalent with the type parameter of the Clean list.

7. In all other cases type unification fails.

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

### 6.5 Related work

Compilation of traditional programming languages to JavaScript has drawn much attention in the last few years as client-side processing for Internet applications has been gaining importance. Virtually every modern language has some kind of technology which allows its client-side execution – see [14] for an overview.

An interesting approach to avoid the usage of JavaScript, is the so called single-language compilation technique. Single-language systems allow the development of all tiers of a whole client-server application in the same language.
Those parts of the application which are needed on the client are automatically transformed to JavaScript, while the other parts are compiled to some server-side binary. Communication between the client and the server can be transparent. The most mainstream example is GWT [66] for Java. As for functional languages, the prominent representatives of this approach are Links [28] and Hop [96]. A notable advantage of single-language systems is that the whole application can be type checked. However, mixed-languages solutions, like ours, are also advantageous: one can use the best of all languages. GWT, for instance, also makes it possible to export libraries as well [66].

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

UHC-JS is the JavaScript backend of the Utrecht Haskell Compiler [35]. Although it is still in beta stage, it can already successfully compile a fair amount of Haskell programs. Its main advantage is that the generated JavaScript code is acceptably small, albeit relatively slow. Compilation can either proceed in a per-module basis or the modules can be linked together using source code level linking. Unfortunately, in the second case the whole application has to be compiled, and the start expression cannot be specified. Its abilities to interact with JavaScript are very limited. In fact, they are restricted to a standard foreign function interface (FFI) and some DOM manipulation libraries implemented above it.

The Fay language [46] has a unique approach – namely, it does not utilize a Haskell compiler for preprocessing, but directly parses Haskell source code using third party libraries, and generates JavaScript code from the abstract syntax tree. As a consequence, Fay supports only a limited subset of the Haskell language, which makes it less appealing for us. JavaScript interoperability is enabled through a trivial foreign function interface.

GHCJS [63] is the most promising compiler technology among those discussed here. However, it has a rather heavyweight approach compared to our solution. It compiles most Haskell libraries without a problem, but suffers from a relatively slow engine (an advanced engine is under development) and huge code footprint. It uses GHC as a front end, and JavaScript code is generated from the resulting STG. Complete interactive applications can be developed using GHCJS through non-standard support libraries, such as WebKit, bindings for WebKitGTK+, which provide a low level DOM interface, and different low and high level interfaces for JavaScriptCore. Unfortunately, due to the use of these libraries, even the most trivial application will consist of several hundred kB (or even MB) of JavaScript. On the other hand, these libraries enable the most advanced JavaScript interoperability among the compilers of study. Besides the ubiquitous FFI support, GHCJS enables callbacks to the Haskell code as well. Type safety of these calls are ensured, but limited to
primitive types, like Numbers, Booleans and Strings. Furthermore, GHCJS utilizes an algebraic data type to deal with JavaScript values – this is highly limited compared to our Dynamic-based approach. The deployment process is overcomplicated, several JavaScript files are generated, and have to be included in the final application along with numerous pre-compiled libraries.

Finally, the Haste compiler [47] is a relatively new approach aiming at small code footprint and a fast engine. Currently it compiles only full applications, which sets a limit on its applicability. Haste supports calling JavaScript functions from Haskell through a standard foreign function interface.

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

6.6 Conclusions

In this paper an extension to iTask and Tasklets has been presented, which enables rapid client-side web development with Clean. The solution is basically an unorthodox application of the iTask system, which in this way becomes an application server for client-side web applications. The presented method, in terms of deployment and integration, makes web development in Clean a competitive alternative to development directly in JavaScript. In terms of productivity, the balance is clearly tilted towards programming in Clean.

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

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

The technology described here can be generalized in at least two ways. First, languages other than Clean can be used for writing the main body of applications. Our Clean to JavaScript compiler uses Sapl [39] (one of the core
languages of Clean) as an intermediate language. A Haskell to Sapl compiler is currently under development. Besides writing a small server-side application for run-time source code level linking of Sapl and the compilation of the result to JavaScript, one technical problem must be solved: to obtain dynamically the Sapl source code of an arbitrary expression. This would make Haskell a proper replacement for Clean here.

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

iTasks enables the rapid creation of multi-user web-applications by automatically generating form-based graphical user interfaces (GUIs) for any first-order type. In some situations, however, form-based GUIs are not sufficient or do not even make sense. We introduce a purely compositional library for creating interactive user interface components, based on Scalable Vector Graphics (SVG). Not only are all images purely compositional, interaction on them is specified by pure functions. The graphics library is integrated with iTasks in such a way that one can easily switch between the generic form-like GUIs and graphics-based user interfaces. Still, a large part of the library is fully iTasks-agnostic and can therefore be used in other contexts as well. We demonstrate the capabilities of this library by implementing the multi-player Ligretto card game in iTasks. This is an interesting case study because it requires a good answer to the challenges of defining multi-user, distributed applications with appealing graphics.

7.1 Introduction

The iTask system [113, 92] (iTasks) is an implementation of the Task Oriented Programming (TOP) paradigm in the strongly typed, lazy, purely functional programming language Clean [114]. The TOP paradigm has been designed to support the development of distributed, multi-user web applications in which humans and software systems collaborate. iTasks offers a client-server infrastructure for the coordination of the tasks being defined, where typically multiple people work closely together on the Internet, making use of standard browsers. Types play a central role in iTasks: from any first-order type, a form-like graphical user interface (GUI) is generated automatically. To do this successfully, it is vital that these interfaces are purely compositional, i.e.: the meaning of an interface is determined exclusively by its sub-components and their composition. This design principle can be traced back to Henderson’s Functional Geometry [71], and indeed, the form-like GUIs generated by iTasks adhere to this property.

For many application domains, such as status displays or games, commu-
communicating information via form-like GUIs is not informative enough, or simply not appropriate. In these cases, it is better to use dynamically adjustable interactive graphics. Several libraries already exist that allow a programmer to create interactive graphics using JavaScript and HTML 5. However, all libraries that we have encountered impose a hidden state model on their API, e.g., by using some kind of single-canvas-abstraction, having attribute-setting operations, using canvas-wide transformations, and so on. Put in other words, they are not purely compositional. Lack of compositionality places the burden on the programmer to find out in which order the graphics operations need to be performed to create the desired images. A compositional image library would shift this problem from the programmer to the library author.

For example, for the communication with domain experts, we are currently developing Tonic [121]. It automatically generates a kind of task flow-chart at compile-time, called a blueprint, that displays an iTasks program’s static task structure. Blueprints are augmented with concrete information at run-time to show which concrete tasks have been created, who is working on what, what progress has been made, how tasks are related to each other, etc. Generating images requires compositionality, since their sizes are generally not known beforehand. The lack of a compositional graphics library has hampered the development of this tool in such a way that we decided to design a new graphics library which is compositional. In the implementation we have to compensate for the lack of compositionality in the underlying libraries.

There are many real-world use-cases that can profit from compositional images. One such use-case is found in the naval domain. Modern ships include interactive plotting-boards that schematically display the ship’s layout. These boards are dynamically updated when, e.g., calamities arise, such as fire or leaks. These same boards can then be used interactively to coordinate calamity mitigation efforts. At the same time, graphs and dials may indicate the fire’s heat developments or a leak’s water levels. We anticipate that using a compositional graphics library reduces the development time of these plotting-boards, and similar systems, significantly.

Being able to draw images in a compositional way solves the drawing problem, but we also need to be able to deal with interaction. Fortunately, this is what iTasks is designed for. In this paper, we introduce the Graphics.Scalable library, with which one can create custom vector-based images in a purely compositional way. We integrate this library seamlessly in the TOP concept of interactive editor tasks in order to make images interactive, using only pure functions.

The integration with iTasks turns out to be mutually beneficial. The image library profits because interaction can be specified as pure functions on model data types within editor tasks, and it can rely on the existence of task combinators to specify application behaviour. This greatly simplifies the API of the
image library. Vice versa, iTasks profits because the appearance and behaviour of editor tasks can be customized to meet the needs of applications that require custom user interfaces.

A real-world use-case, which we address specifically in this paper, concerns multi-player, distributed games such as Trax [9] and Ligretto. We demonstrate how the latter card game can be created with the Graphics.Scalable image library and iTasks.

In this paper we make the following contributions:

- We present the Graphics.Scalable library, a purely compositional image API.
- We define interaction on images using pure functions.
- We integrate the library in iTasks, reusing the existing editor infrastructure.
- We demonstrate its usage by a case study: the Ligretto game.
- We map Graphics.Scalable images to the Scalable Vector Graphics (SVG) standard [32].
- We overcome the technical challenges imposed by the client/server architecture of the Internet, using editlets.

We start our explanation by first concentrating on static, purely compositional images as provided in Graphics.Scalable in Section 7.2. We show how to render the state of the card game Ligretto. This is a non-trivial show-case of compositional rendering (you are invited to browse ahead to Figure 7.1(I)). We show how static images are made interactive in Section 7.3 and turn the example into a full-fledged, multi-user application. The underlying technology of the Graphics.Scalable library is SVG. Mapping to SVG has proven to be challenging mostly because SVG adopts a single-canvas rendering model which conflicts with the purely compositional nature of Graphics.Scalable. The implementation is presented in Section 7.4.

Functional programming and creating images, whether they are interactive or compositional or both, share a long research history. The Graphics.Scalable API is greatly influenced by old and recent research. In Section 7.5 we discuss this in more detail. The combination of the Graphics.Scalable image library and TOP is a novel contribution to the field of programming interactive applications in a functional style. We conclude in Section 7.6.
7.2 Compositional Static Images

In this section we describe the compositional image library (Sections 7.2.1–7.2.6). The concepts are illustrated step by step by rendering the entire state of the Ligretto card game (Section 7.2.7).

7.2.1 Image concepts

Conceptually, an image is an infinitely large, perfectly transparent ‘slide’ that renders a value of some model type \( m \). This is captured with the opaque type \( \text{Image} \ m \). The ‘slide’ can be scaled, rotated, and skewed. There is no global coordinate system. When defining an image we impose a local coordinate system, the span box. The span box consists of two dimensions: the \( x \)-span increases from ‘left’ to ‘right’ (perfectly horizontal) and the \( y \)-span increases from ‘above’ to ‘below’ (perfectly vertical). The unit of measure is pixel, expressed with real values. Pixels get a physical interpretation only when the image is actually rendered on a device. This is natural in the context of scalable vector graphics.

It is important to note right away that the span box is not the same as the common bounding box concept. The bounding box of an image is identified by the minimum and maximum coordinates of its visual content. In contrast, the span box of an image defines its conceptual size that is used for layout. We deliberately allow visual content to exist outside of the span box or within a ‘tighter’ bounding box. These design decisions seem to be minor, but they are not: what an image looks like should be unconnected with where it happens to be and what its size is.

Stacking ‘slides’ is the only way to compose new images from simpler ones. Conceptually, stacking creates a \( z \)-axis that is oriented perfectly towards the viewer. ‘Higher’ images can obscure ‘lower’ images, depending on their opacity or masking attribute (Section 7.2.3). We literally create a collage. The span boxes of the images are used to specify their relative positions. For that purpose layout combinators are used (Section 7.2.4). Note that in the presence of infinitely large images, a translation transformation does not change the image, hence our library does not support image translation. All we need to care about are the relative positions of images.

7.2.2 Basic images

The image library supports common shapes as basic images:

\[
:: \text{Span} \quad // \text{an opaque data type} \\
\text{px} :: \text{Real} \rightarrow \text{Span} \quad // (\text{px} \ x) \text{ represents } x \text{ pixels}
\]
Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics

empty :: Span Span → Image m
circle :: Span → Image m // Circle by diameter
ellipse :: Span Span → Image m
rect :: Span Span → Image m
xline :: Span → Image m // Lines are 1-dimensional
yline :: Span → Image m // Lines are 1-dimensional
line :: Slash Span Span → Image m
text :: FontDef String → Image m
normalFontDef :: String Real → FontDef

:: Slash = Slash | Backslash

A number of aspects are worth noting. The empty image has no visual content and only an \(x\)-span and a \(y\)-span. What a piece of text looks like is determined by the used font as well as the content, both must be part of its specification. The FontDef structure collects all SVG font properties, such as font-size, font-weight, font-style, etcetera. The convenience function (normalFontDef name h) captures the frequently occurring situation that it suffices to specify the font family name and font height in pixels (also the \(y\)-span), setting all other font properties to "normal". The \(x\)-span of the text image depends on the used font and text. The default renderings of the circle, ellipse, and rect shapes is the same as the default rendering of text. These can be changed with the image attributes (Section 7.2.3). Finally, lines are also drawn with a default stroke of one pixel and use the color black. In the presence of rotation a single line primitive is sufficient, but for convenience we provide primitives for horizontal, vertical, and ‘tilted’ lines (xline, yline, line). The Slash parameter identifies the imaginary rectangle corner points that are ‘connected’ by the line (Slash, /, left-bottom to right-top corner and Backslash, \, left-top to right-bottom corner).

### 7.2.3 Image attributes

Image attributes alter the appearance of visual elements without altering the span box. In this way, the purpose of the span box does not get mixed with the appearance of an image. In SVG, attributes are defined with name-value pairs. We adopt the SVG names:

:: StrokeAttr m = { stroke :: SVGColor }
:: StrokeWidthAttr m = { strokewidth :: Span }
:: XRadiusAttr m = { xradius :: Span }
:: YRadiusAttr m = { yradius :: Span }
:: FillAttr m = { fill :: SVGColor }
:: OpacityAttr m = { opacity :: Real }
:: DashAttr m = { dash :: [Int] }
:: MaskAttr m = { mask :: Image m }
Each type constructor is made an instance of a type constructor class `tuneImage`, having trivially derived operators and function.

```haskell
class tuneImage attr :: (Image m) (attr m) -> Image m
(>▷) infixr 2 :: (attr m) (Image m) -> Image m | tuneImage attr
(<@<) infixl 2 :: (Image m) (attr m) -> Image m | tuneImage attr

tuneIf :: Bool (Image m) (attr m) -> Image m | tuneImage attr
```

For the specification of colors we adopt the extensive set of SVG color names and the common RGB-triplets:

```haskell
class toSVGColor a :: a -> SVGColor
instance toSVGColor String, RGB
:: RGB = { r :: Int, g :: Int, b :: Int }
```

### 7.2.4 Image composition

Images are composed by stacking. The images that are to be stacked are given in a *finite* list. Elements with lower list-index positions can be obscured by elements with higher list-index positions. This leaves only the relative layout along the $x$-axis and $y$-axis unspecified. This relative layout can be defined with or without a *host image*. A host image serves two purposes: its span box is the local coordinate system in which the positions of the stacked images are specified, and it is the background image on top of which these images are stacked. If no host image is used, then the span box equals the bounding box of the span boxes of the stacked images. Offsets are defined as a pair of an $x$-span and $y$-span value. The initial layout of images is *always* computed without the offsets. The final layout is obtained by adding the $i$-th offset to the initial position of the $i$-th image.

```haskell
:: Layout m := ImageOffset -> [Image m] -> (Host m) -> Image m
:: Host m := Maybe (Image m)
:: ImageOffset := (Span, Span)
```

The image list must be finite. In the image layout functions, any other list argument need not have the same length. If they are too short, then padding values are defined for them (for offsets, this is zero). If they are too long, then the surplus is not evaluated. In this way we can keep the specification of the image list separate from other concerns such as offsets and alignments in the other image layout functions. It also avoids cluttering of the image list specifications.

Conceptually, the image library has only one *core* image layout function:\(^1\)

\(^1\)Although internally, other layout combinators are modeled explicitly as well for reasons of efficiency.
collage :: Layout m

In a collage, the images are initially stacked with their left-top span box corners aligned. The final position of the \( i \)-th image is obtained by adding the \( i \)-th offset to that initial position.

*Derived* image layout functions are overlay, grid, above, below, and margin. The first of them, overlay, adds horizontal and vertical alignment options to the layout specification:

\[
\text{overlay} \quad :: \quad [\text{ImageAlign}] \rightarrow \text{Layout m}
\]

\[
:: \text{ImageAlign} := (\text{XAlign}, \text{YAlign})
:: \text{XAlign} = \text{AtLeft} | \text{AtMiddleX} | \text{AtRight}
:: \text{YAlign} = \text{AtTop} | \text{AtMiddleY} | \text{AtBottom}
\]

In an overlay, the initial position of the images is determined using the list of alignments: the position of the \( i \)-th image is determined by the \( i \)-th alignment value. The final position of the \( i \)-th image is obtained by adding the \( i \)-th offset value to the \( i \)-th initial position.

Images often need to be placed in a grid-like structure:

\[
:: \text{GridDimension} = \text{Rows Int} \quad | \quad \text{Columns Int}
:: \text{GridMajor} = \text{ColumnMajor} \quad | \quad \text{RowMajor}
:: \text{GridXLayout} = \text{LeftToRight} \quad | \quad \text{RightToLeft}
:: \text{GridYLayout} = \text{TopToBottom} \quad | \quad \text{BottomToTop}
:: \text{GridLayout} := (\text{GridMajor}, \text{GridXLayout}, \text{GridYLayout})
\]

\[
\text{grid} :: \text{GridDimension} \quad \text{GridLayout} \quad [\text{ImageAlign}] \rightarrow \text{Layout m}
\]

A grid’s dimensions are specified by providing either a number of rows or a number of columns. The number of images then determines the corresponding number of columns or rows. The grid can be populated in eight different ways, determined by the grid layout: column-by-column or row-by-row (GridMajor), in combination with left-to-right or right-to-left (GridXLayout), in combination with top-to-bottom or bottom-to-top (GridYLayout). The span boxes and alignments of the images are used to compute the images’ initial positions, which are then fine-tuned with the corresponding offsets to obtain all final positions.

Images are often placed beside or above each other:

\[
\text{beside} :: [\text{YAlign}] \rightarrow \text{Layout m}
\]

\[
\text{above} :: [\text{XAlign}] \rightarrow \text{Layout m}
\]

These are immediately derived from the grid image layout function: beside is one row of left-aligned images and above is one column of top-aligned images.

Finally, it is useful to add margins around an image. This merely increments the span box but does not alter the image. We follow the convention of SVG to specify margins in several ways:
Chapter 7

class margin a :: a (Image m) \rightarrow Image m
instance margin Span,
    (Span, Span),
    (Span, Span, Span),
    (Span, Span, Span, Span)

The ‘one-span’ instance $a$ imposes a uniform margin $a$ around the image, the
‘two-span’ instance $(a, b)$ imposes margin $a$ above/below and $b$ left/right of the
image, the ‘three-span’ instance $(a, b, c)$ imposes margin $a$ above, $b$ left/right, $c$
below the image, and the ‘four-span’ instance $(a, b, c, d)$ imposes margin $a$
above, $b$ right, $c$ below, and $d$ left of the image.

7.2.5 Symbolic span expressions

The image layout functions need to manipulate span values symbolically in
order to compute the desired image positions. Examples of symbolic span
values are text width, image width and height, column width, and row height.
Examples of symbolic span computations are the usual arithmetical operations
as well as negating the value and taking the absolute value and determining
the minimum and maximum span value. These are covered by the following
span-definitions and instances of arithmetic operations:

:: ImageTag

// Symbolic span values:
textxspan :: FontDef String \rightarrow Span // text width
imagexspan :: ImageTag \rightarrow Span // image width
imageyspan :: ImageTag \rightarrow Span // image height
columnspan :: ImageTag Int \rightarrow Span // column width
rowspan :: ImageTag Int \rightarrow Span // row height

// Symbolic span arithmetic:
instance zero Span
instance + Span
instance - Span
instance \sim Span
instance abs Span

class (\*) infixl 7 a :: a n \rightarrow a | toReal n
class (/.) infixl 7 a :: a n \rightarrow a | toReal n
instance \times. Span, Real, Int
instance /. Span, Real, Int

minSpan :: [Span] \rightarrow Span
maxSpan :: [Span] \rightarrow Span
The opaque type `ImageTag` refers to an image. In case of `imagexspan` and `imageyspan`, this can be any image; in case of `columnspan` and `rowspan`, the image tag needs to be associated to a grid image. The number argument of the latter two functions identifies the column or row number, starting at index zero. If the image tag does not happen to refer to an image, then the symbolic span value is zero.

Image tags must identify an image uniquely. This is guaranteed by taking advantage of Clean’s *uniqueness type system*. The image author has no means to define `ImageTag` values herself. Instead, the top-level image rendering function is provided with an infinite list of fresh image tag values. These image tag values come in pairs: the first is a *non-uniquely attributed* image tag (of type `ImageTag`) and the second is a *uniquely attributed* image tag (of type `*ImageTag`). To identify an image, the image author is forced to use the uniquely attributed image tag:

\[
tag :: \*ImageTag \ (Image \ m) \to Image \ m\]

In this way, it is statically guaranteed that an image tag is associated with an image at most once. Even if the tagged image is used several times, it is guaranteed that the tag identifies the very same image. Hence, the corresponding symbolic span values have the same size.

The types of the arithmetic operations should reflect the ‘physical’ dimension. Span values can be added and subtracted, and their absolute and negated value can be computed. These operators do not alter the dimension, so they can be defined using ordinary operator overloading (\(+\), \(-\), \(\text{abs}\), and \(\sim\)). For other operators this is not true: multiplication of span results in square span, division of span results in a scalar value, and comparison of span values evaluates to a boolean. For this reason the image library supports slightly different overloaded operators for these purposes: \(*\) and \(\div\) for multiplication and division with a scalar value, and \(\text{minSpan}\) and \(\text{maxSpan}\) for determining the smallest and largest span from a list of span values. The experiments that we have conducted so far indicate that the lack of comparison operators on span values does not limit the expressiveness of symbolic span expressions.

Finally, the symbolic span expression language in combination with the `collage` image layout function is sufficiently expressive to derive all other image layout functions (shown in Figure 7.6). This expressive power is also available for the image author who can use the same language to define new image layout patterns herself.

### 7.2.6 Image transformations

Any (composite) image can be subject to transformation:
Chapter 7

rotate :: Angle (Image m) → Image m
skewx :: Angle (Image m) → Image m
skewy :: Angle (Image m) → Image m
fit :: Span Span (Image m) → Image m
fitx :: Span (Image m) → Image m
fity :: Span (Image m) → Image m
flipx :: (Image m) → Image m
flipy :: (Image m) → Image m

:: Angle
rad :: Real → Angle
deg :: Real → Angle

Angles are expressed as radians or as degrees. In general, the span box of a rotated or skewed image differs from the span box of the original image. Non-proportional scaling is done with fit which ensures that the resulting image has exactly the specified x-span and y-span. Proportional scaling is done with fitx and fity: they ensure exact x-span and y-span, respectively and scale the other span proportionally. Flipping, or mirroring, an image around its x- or y-axis is done with flipx and flipy.

7.2.7 Case study: rendering the Ligretto state

In this section we demonstrate how to exploit the compositional features of the image library to render the state of a game of Ligretto. We first present the data types that model the game state (Section 7.2.7) and then show how it is rendered (Section 7.2.7).

Ligretto model types

Ligretto is a card game for two, up to twelve players. In this paper we restrict ourselves to a maximum of four players. Each player has forty cards that come in four front colors: red, green, blue, and yellow. The ten cards of one color are numbered on the front side from one through ten. For identification purposes, the back sides of the cards have a unique color for each player. These facts can be modeled in a straightforward way:

:: Card = { back :: Color, front :: Color, no :: Int }
:: SideUp = Front | Back
:: Color = Red | Green | Blue | Yellow

At the start of the game, each player shuffles her cards, and places them as follows on the table from right to left (Figure 7.1(k)):

- The row cards, which lie beside each other, faced up. The number depends on the number of players (five cards in case of two players, and up to three cards in case of four players).
• The ligretto pile, which is a pile of ten cards, faced up.

• The hand cards, which is divided in two sub piles: the concealed pile which at start are all remaining cards, facing down, and the discard pile which come from the concealed pile, facing up.

Finally, there is a shared area for all players, called the middle (Figure 7.1(j)). In the middle, piles of cards of the same front color are created by all players at the same time. A new pile must always start with number 1, face up. Cards with a number $n + 1$ are allowed to be placed only on a middle pile of the same front color and top-most card having number $n$. Although players are uniquely identified via their color, we also keep track of their name and render it in the game. These facts are modeled as follows:

```
:: NoOfPlayers ::= Int
:: Middle ::= [Pile]
:: Pile ::= [Card]
:: Player = { color ::= Color,
                  name ::= String,
                  row ::= RowPlayer,
                  ligretto ::= Pile,
                  hand ::= Hand,
                  seed ::= Int }
:: RowPlayer ::= [Card]
:: Hand = { conceal ::= Pile, discard ::= Pile }
```

no_of_cards_in_row ::= NoOfPlayers → Int

colors ::= NoOfPlayers → [Color]

The complete Ligretto game state consists of the middle card piles and the participating players:

```
:: GameSt = { middle ::= Middle, players ::= [Player] }
```

We can now turn our attention to rendering this game state.

**Ligretto rendering**

The Ligretto game state is rendered step by step in a compositional way. The individual images are shown in Figure 7.1.

We start with defining images for cards and attempt to make them look similar to commercially available Ligretto cards. The physical size of these cards is 58.5mm by 90.0mm, so we adopt these values for the rendered cards as well:

```
card_width = px 58.5
card_height = px 90.0
```
Figure 7.1: Compositional rendering of the Ligretto game state
The shape of a Ligretto card is that of a rectangle with rounded corners (Figure 7.1(a)):

\[
\text{card\_shape} = \text{rect \ card\_width \ card\_height} \\
\quad <@ \{ \text{xradius} = \text{card\_height} / 18 \} \\
\quad <@ \{ \text{yradius} = \text{card\_height} / 18 \}
\]

For rendering the text on cards we use the font family *Verdana* in several sizes:

\[
\text{cardfont \ size} = \text{normalFontDef "Verdana" \ size}
\]

The model colors need to be mapped to SVG colors that best match the physical cards. We select the following SVG colors:

\[
\text{instance \ toSVGColor \ Color \ where} \\
\quad \text{toSVGColor Red} = \text{toSVGColor "darkred"} \\
\quad \text{toSVGColor Green} = \text{toSVGColor "darkgreen"} \\
\quad \text{toSVGColor Blue} = \text{toSVGColor "midnightblue"} \\
\quad \text{toSVGColor Yellow} = \text{toSVGColor "gold"}
\]

We abbreviate *white* and *black*:

\[
\text{white} = \text{toSVGColor "white"} \\
\text{black} = \text{toSVGColor "black"}
\]

The number on the front side of a card is displayed in a large font (Figure 7.1(b) shows big_no 7 Red):

\[
\text{big\_no \ no \ color} = \text{text (cardfont 20.0) (toString no)} \\
\quad <@ \{ \text{fill} = \text{white} \} \\
\quad <@ \{ \text{stroke} = \text{toSVGColor color} \}
\]

At the back side of the card, the text *Ligretto* is displayed (Figure 7.1(c) shows ligretto Red):

\[
\text{ligretto \ color} = \text{text (cardfont 12.0) "Ligretto"} \\
\quad <@ \{ \text{fill} = \text{toSVGColor "none"} \} \\
\quad <@ \{ \text{stroke} = \text{toSVGColor color} \}
\]

With these image functions, we can render the front side (Figure 7.1(d) or back side (Figure 7.1(e)) of a card:

\[
\text{card\_image :: SideUp \ Card \ → \ Image \ m} \\
\text{card\_image \ side \ card} \\
\quad | \ \text{side} = \text{Front} \\
\quad \quad = \text{let} \ \text{no} = \text{margin (px 5.0) (big\_no \ card\_no \ (no\_stroke\_color \ card\_front))} \\
\quad \quad \quad \text{in} \ \text{overlay} [(\text{AtMiddleX, AtTop}), (\text{AtMiddleX, AtBottom})] [] \\
\quad \quad \quad [\text{no, rotate (deg 180.0) no} \ \text{host}] \\
\quad | \ \text{otherwise} \\
\quad \quad = \text{overlay} [(\text{AtMiddleX, AtBottom})] [] \\
\quad \quad \quad [\text{skewy (deg -20.0) (ligretto \ card\_back)}] \ \text{host}
\]

where
Chapter 7

```haskell
host = Just (card_shape <@< \{ fill = if (side== Front) (toSVGColor card.front) white\})
```

The stroke color of the card number depends on the card color:

```haskell
no_stroke_color :: Color -> Color
no_stroke_color Red = Blue
no_stroke_color Green = Red
no_stroke_color Blue = Yellow
no_stroke_color Yellow = Green
```

We introduce an ‘empty card’ that serves as a visual placeholder for an empty pile (Figure 7.1(f)).

```haskell
no_card_image :: Image m
no_card_image = overlay [(AtMiddleX, AtMiddleY)] []
  [text (pilefont 12.0) "empty"] host
where
  host = Just (card_shape <@< \{ fill = toSVGColor "lightgrey"\})
```

The simplest way of rendering a pile of cards is to render only the top-most card. However, in this way, players have no visual clue how many cards the pile has. Instead, we display the cards as being stacked on top of the ‘empty card’ in reversed order and each card having a slightly increased vertical offset (Figure 7.1(g)):

```haskell
pile_of_cards :: SideUp Pile -> Image m
pile_of_cards side pile
  = overlay [] [(zero,card_height / . 18 *. dy) \\ dy<- [0 .. ]] (map (card_image side) (reverse pile)) host
where
  host = Just no_card_image
```

For large piles it does not make a lot of sense to show all cards, so instead we show the top-most ten cards (if present) of a pile. For larger piles we include the total number of cards as a small number above the rendered pile (Figure 7.1(h)).

```haskell
pile_image :: SideUp Pile -> Image m
pile_image side pile
  | no_of_cards > 10 = above [ AtMiddleX ] []
    [ text (pilefont 10.0)
      (toString no_of_cards)
      , top_cards_image ]
    Nothing
  | otherwise = top_cards_image
where
  no_of_cards = length pile
  top_cards_image = pile_of_cards side (take 10 pile)
```
We choose to render the player names as a bold faced text on top of a rectangle that is filled with the player’s card color. Instead of scaling long or short names, we use masking to prevent long names from running outside of the host image (Figure 7.1(i) shows the result for a player named *alice* playing the red cards).

```haskell
name_image :: Player → Image m
name_image {name,color} = overlay [(AtMiddleX,AtMiddleY)] []
    [ text {cardfont 16.0 & fontweight = "bold"} name
      @@ {fill = if (color== Yellow) black white}
      ] host
    @@ {mask = rect width height @@ {fill = white}
        @@ {stroke = white}}
where
    width = card_height * 1.8
    height = card_width * 0.4
    host = Just (rect width height @@ {fill = toSVGColor color})
```

With the above ingredients we are able to render a complete Ligretto game state. The players are ‘sitting’ at a round table. We arrange the elements as three concentric circular tiers. The innermost tier contains the middle cards, the middle tier shows the player names, and the outermost tier shows the player cards. For this purpose we first create a general function that moves and rotates an arbitrary list of images *imgs* along a circle segment of *a* radians, and the circle having radius *r*:

```haskell
circular :: Span Real [Image m] → Image m
circular r a imgs = overlay (repeat (AtMiddleX,AtMiddleY))
    [ (~* *. cos angle, ~* *. sin angle)
      \ i ← [0.0, sign_a ..]
      , angle ← [i * alpha - 0.5 * pi]]
    [ rotate (rad (i * alpha)) img
      \ i ← [0.0, sign_a ..]
      & img ← imgs]
    (Just (empty (r * 2) (r * 2)))
where
    sign_a = toReal (sign a)
    alpha = toRad (normalize (rad a)) / toReal (length imgs)
```

The circular image is created by stacking all images with their centers (according to their span boxes) aligned. Each image gets placed along the circle segment using the proper offset and gets oriented along that circle segment by rotating the image with the same angle.

The innermost tier, *middle_image*, simply distributes all middle piles along a full circle:

```haskell
middle_image :: Span Middle → Image m
```
middle_image \( r \) middle
\[ = \text{circular} \ (2.0 \times \pi) \ (\text{map} \ (\text{pile_image} \ Front) \ \text{middle}) \]

Figure 7.1(j) shows the result of the initial middle for three players, which consists of twelve empty piles, as each player has the potential to start four piles.

The middle tier, \( \text{names_image} \), distributes all player names along a full circle:

\[
\text{names_image} :: \text{Span} \ [\text{Player}] \rightarrow \text{Image} \ m \\
\text{names_image} \ r \ \text{players} \\
= \text{circular} \ (2.0 \times \pi) \ (\text{map} \ \text{name_image} \ \text{players})
\]

Before we construct the outermost tier of all players, we first render the cards of a single player. These are either in a pile (the hand and Ligretto piles), or are single cards (the row cards).

\[
\text{hand_images} :: \text{Hand} \rightarrow [\text{Image} \ m] \\
\text{hand_images} \ \{\text{conceal, discard}\} \\
= [ \text{pile_image} \ \text{Back} \ \text{conceal}, \ \text{pile_image} \ \text{Front} \ \text{discard} ]
\]

\[
\text{row_images} :: \text{RowPlayer} \rightarrow \text{Image} \ m \\
\text{row_images} \ \text{row} = \text{map} \ (\text{card_image} \ Front) \ \text{row}
\]

The player cards are placed along a circle segment that is slightly less than a quarter of a circle (Figure 7.1(k)):

\[
\text{player_arc} = 0.45 \times \pi
\]

\[
\text{player_image} :: \text{Span} \ \text{Player} \rightarrow \text{Image} \ m \\
\text{player_image} \ r \ \{\text{row, ligretto, hand}\} \\
= \text{circular} \ r \ \text{player_arc} \ ( \text{row_images} \ \text{row} \\
+ + [\text{pile_image} \ \text{Front} \ \text{ligretto}] \\
+ + \text{hand_images} \ \text{hand} )
\]

The outermost tier, \( \text{players_image} \), distributes all player cards along a full circle:

\[
\text{players_image} :: \text{Span} \ [\text{Player}] \rightarrow \text{Image} \ \text{GameSt} \\
\text{players_image} \ r \ \text{players} \\
= \text{rotate} \ (\text{rad} \ \text{angle}) \\
\ (\text{circular} \ \text{zero} \ (2.0 \times \pi) \ (\text{map} \ (\text{player_image} \ r) \ \text{players}))
\]

where

\[
\text{angle} = \text{player_arc} / (\toReal \ (2 \times \text{no})) - \text{player_arc} / 2.0 \\
\text{no} = 3 + \text{no_of_cards_in_row} \ (\text{length} \ \text{players})
\]

Without the additional rotation, the first player’s cards are displayed as shown in Figure 7.1(k). We prefer the layout of Figure 7.1(l) and therefore rotate the entire image by half the \( \text{player_arc} \), decreased with half the angle required for one card.
Finally, the entire image overlays the three tiers (Figure 7.1(l) gives the result of a typical initial Ligretto game state for three players):

```haskell
game_image :: GameSt → Image m
game_image {players,middle}
  = overlay (repeat (AtMiddleX,AtMiddleY)) []
    ([(middle_image (card_height *. 2 ) middle,
       names_image (card_height *. 3.2) players,
       players_image (card_height *. 4 ) players])
    ) host

where
  host = Just (empty (card_height *. 12) (card_height *. 12))
```

7.2.8 Discussion

When thinking of an image-under-construction, we map each individual layer to an image. What an image looks like, and how we would like to use it in layout, are two distinct concepts that we have separated by replacing bounding box with span box, and thinking of images as if they are infinitely large. When thinking of the layout, we first and foremost decide on the overall layout (e.g. collage or grid, relying on span boxes), and pinpoint the exact location (alignment and offsets) later. Finally, when design choices are in a sense arbitrary, we have adopted SVG’s design choices.

7.3 Compositional Interactive Images

In this section, we describe how to turn static images into interactive ones by integrating them in iTasks. We start with a brief description of iTasks (Section 7.3.1). In iTasks, user-interaction is delegated to specialized tasks; the editor tasks. Hence, these are the tasks that need to be enriched with images (Section 7.3.2). Finally, we show how to turn the static Ligretto images interactive, and create a complete TOP specification of a game of Ligretto (Section 7.3.3).

7.3.1 iTasks essentials

The TOP paradigm, as embodied in iTasks, builds on a few core concepts: tasks, which define the work that needs to be done; combinators, to compose tasks from simpler ones; editors, which are tasks that facilitate user interaction; and shared data sources (SDSs), to handle shared information in a uniform way.

Tasks are represented by the monad-like\(^2\) type \(\text{Task } a\), which has an associated task value of type \(a\). By inspecting the current task value, other task

\(^2\)We say monad-like, because the right-identity law does not hold for Task
functions) can get informed about the state of the task (in progress or finished).
Tasks can be composed sequentially, using the step combinator \((\Rightarrow\Rightarrow)\), or in
parallel, using the parallel combinator. Examples of their use are given
when we continue with the case study in Section 7.3.3.

Editors are a means to view data or to interact with it. They are tasks
that use type-driven generic programming to generate a user interface for any
first-order type. Examples of editors are viewInformation, used to provide
a read-only editor for a given type, and updateInformation, which allows
the user to modify a value. The types of these editors are given here:

\[
\begin{align*}
\forall v : \text{ViewWith} (a \rightarrow v) & \land \text{iTask } v \\
\forall v : \text{UpdateWith} (a \rightarrow v) (a \rightarrow v \rightarrow b) & \land \text{iTask } v
\end{align*}
\]

\[
\begin{align*}
\text{viewInformation} :: \text{Title } [\text{ViewOption } m] m \rightarrow \text{Task } m \mid \text{iTask } m \\
\text{updateInformation} :: \text{Title } [\text{UpdateOption } m m] m \rightarrow \text{Task } m \mid \text{iTask } m
\end{align*}
\]

In both cases, the third parameter is the type of the initial value that is dis-
played or updated. Instead of providing an initial value, an editor can also
be ‘connected’ to an SDS. In that case, the current value of the SDS serves
as source for rendering, and any update coming from the editor is written to
the SDS. In this way, one can define a set of parallel communicating tasks.
For every above-mentioned editor, there is a share-enabled counterpart that
automatically reacts to changes in the SDS they are connected with:

\[
\begin{align*}
\forall r : \text{ReadWriteShared } r w & \land \text{iTask } r \\
\forall r : \text{ReadWriteShared } r w & \land \text{iTask } r
\end{align*}
\]

\[
\begin{align*}
\text{viewSharedInformation} :: \text{Title } [\text{ViewOption } r] (\text{ReadWriteShared } r w) \rightarrow \text{Task } r \mid \text{iTask } r \\
\text{updateSharedInformation} :: \text{Title } [\text{UpdateOption } r w] (\text{ReadWriteShared } r w) \rightarrow \text{Task } w \mid \text{iTask } r \land \text{iTask } w
\end{align*}
\]

Figure 7.2 shows the result of applying these editors to a value of type Card
(Section 7.2.7).

![Figure 7.2: Generic Card view- and updateInformation tasks.](image)

Clearly, neither resulting interface is the one that is required for the case
study (Figure 7.1(d)). It should be noted that without special support from

\[
\begin{align*}
\forall v : \text{introduces an existentially quantified type variable } v, \text{ while } \land \text{iTask } v \text{ places a type-class constraint for class iTask on } v.
\end{align*}
\]
iTasks, the View- and UpdateOption types are of no help either: with these options the programmer can control the domain of the values that are viewed or updated but not the generic rendering. In the next section we show how to integrate the static images into these editors.

### 7.3.2 Enhancing editors with images

We first integrate static images with editors by introducing a new option for view(Shared)Information editors:

\[
\text{imageView} :: (r \to \star[\{\text{ImageTag}, \star\text{ImageTag}\}] \to \text{Image } r) \to \text{ViewOption } r \mid \text{iTask } r
\]

With (imageView \(render\)), the rendering function \(render\) is used to visualize the model value of type \(r\). Hence, with the same Card value that was used in Figure 7.2, the following editor:

\[
\text{viewInformation } \text{"A Ligretto card"} \\
\text{[ViewWith (imageView (\lambda \text{card } \_ \to \text{card_image Front card}))]} \text{red_green_7_card_model}
\]

displays the card graphically, as in Figure 7.1(d).

Interactive images require more effort. First, we introduce a new option for update(Shared)Information editors:

\[
\text{imageUpdate} :: (r \to v) (v \to \star[\{\text{ImageTag}, \star\text{ImageTag}\}] \to \text{Image } v) (r \to v \to w) \to \text{UpdateOption } r \mid \text{iTask } v
\]

With (imageUpdate \(f render g\)), a source value of type \(r\) is transformed to a view model with function \(f\), to which the \(render\) function is applied to create the image. Whenever the viewed value is changed by an interaction, a destination value of type \(w\) is constructed out of the original source value and changed view value with function \(g\).

Second, we need to make the images themselves interactive. In Section 7.2.3 we have omitted one image attribute:

\[
\text{:: OnClickAttr } m \{ \text{onclick } :: m \to m \}
\]

If \(img\) has type \((\text{Image } m)\) then \((img <@< \{\text{onclick } = f\})\) is the same image enhanced with \textit{mouse hit-detection}. Whenever the user clicks on a part of \(img\), then the function \(f\) is applied to the current model value that is associated with the image and computes a new model value, updating the model value. In turn, this triggers the functions on the update(Shared)Information editors to re-render the model value, if necessary. For example, when a change is made to a shared model value by applying some onclick function after an interaction, all tasks looking at this shared value will automatically be notified and updated such that they can show the new view corresponding with the new model value. Moreover, depending tasks can inspect this new task value, not knowing whether it originated from an interactive image or a generic interactive
task. Compositionality is preserved because the onClick function is unaware of any final position, rotation, skewing, masking, or duplication of the image with which it is associated.

### 7.3.3 Case study continued: interactive Ligretto

In this section we continue with the Ligretto case study in two steps: we turn the static image of Section 7.2.7 into an interactive image (Section 7.3.3) and then proceed with the final iTask specification of the entire game (Section 7.3.3). In this section we assume the presence of the following pure functions:

```haskell
play_row_card :: Color Int GameSt → GameSt
play_concealed_pile :: Color GameSt → GameSt
play_hand_card :: Color GameSt → GameSt
```

(play_row_card player no game) moves the card of `player` found at row number `no` (counting from 1) to an available middle pile and, if such a middle pile exists, moves the top card of the player’s ligretto pile to the row. (play_concealed_pile player game) moves the top three cards of the concealed pile to the discard pile of `player`, if these exist, and shuffles the discard pile back to the concealed pile, if not. Finally, (play_hand_card player game) moves the top card on the discard pile to an available middle pile, if such a pile exists. These functions are only concerned with the model types defined in Section 7.2.7. They ensure that only legal moves can be made.

#### Interactive Ligretto images

The `game_image` function defined at the very end of Section 7.2.7 shows the entire state of the game as seen from the perspective of the ‘first’ player. To show the game from the perspective of any player, we need to rotate the image according to that player’s position in the list of participants. This is the purpose of the `player_perspective` function which is parameterized with the color of the player. This color parameter is also used to make certain that this player can only play her own cards.

```haskell
player_perspective :: Color GameSt * \((\text{ImageTag}, \text{ImageTag})\) → Image GameSt
player_perspective color gameSt _
  = rotate (rad (~{toReal my_no * angle})) (game_image color gameSt)
where
  angle = 2.0 * pi / (toReal (length gameSt.players))
  my_no = hd [i \ \ player ← gameSt.players & i ← [0 ..] | player.color == color]
```

(Note that this function ignores the image tag source because they are not required by any of the image rendering functions.)
The new `game_image` function merely passes the player color to the outermost image tier that renders all playable and non-playable cards. The other two image tiers remain static.

```haskell
game_image :: Color GameSt → Image GameSt
game_image color {players, middle} = overlay (repeat (AtMiddleX, AtMiddleY)) []
  ([middle_image (card_height * 2) middle,
    , names_image (card_height * 3.2) players,
    , players_image (card_height * 4) color players]) host

where
  host = Just (empty (card_height * 12) (card_height * 12))
```

The only change to the `players_image` function is that for each player-rendering it is determined whether this rendering is going to be interactive or not.

```haskell
players_image :: Span Color [Player] → Image GameSt
players_image r color players = rotate (rad angle) (circular zero (2.0 * pi))
  [player_image r (player.color == color) player \ player ← players]

where
  angle = player_arc / (toReal (2 * no)) - player_arc / 2.0
  no = 3 + no_of_cards_in_row (length players)
```

Consequently, `player_image` has an additional Boolean parameter that tells whether the image is interactive. The interactive elements of a player are the row-cards and the hand-cards.

```haskell
player_image :: Span Bool Player → Image GameSt
player_image r interactive player = circular r player_arc
  (row_images interactive player.row
   ++ [pile_image Front player.ligretto]
   ++ hand_images interactive player.hand player.color)
```

Playing a row card is defined by the pure function `play_row_card`. Only if the image is interactive is it added as an `onclick` attribute:

```haskell
row_images :: Bool RowPlayer → [Image GameSt]
row_images interactive row = [tuneIf interactive (card_image Front row_card)
  {onclick = play_row_card row_card.back no}
  \row_card ← row & no ← [1 .. ]]
```

Similarly, the two sub-piles of the hand cards behave as specified by the pure functions `play_concealed_pile` and `play_hand_card`, but only if the images are interactive:
Chapter 7

hand_images :: Bool Hand Color → [Image GameSt]
hand_images interactive {conceal, discard} color
 = [{tuneIf interactive (pile_image Back conceal)
                      {onclick = play_concealed_pile color}
               ,tuneIf interactive (pile_image Front discard)
                      {onclick = play_hand_card color} }]

These extensions are sufficient to turn the static Ligretto rendering into an interactive image that can be used by editor tasks. It should be noted that the compositional style is not compromised by making these images interactive: none of these functions are aware of the ultimate position, angle or size in the fully rendered Ligretto game. The next section shows how to integrate these editor tasks into a complete distributed TOP application.

The Ligretto game

One of the Ligretto players takes the initiative and invites one through three friends to join in. Each player is assigned one of the Ligretto colors. In addition, we need to extract initial random values for the shuffling activities by all players. Once this is done, we can set up the shared game state and start to play:

play_Ligretto :: Task (Color, String)
play_Ligretto
 = get currentUser
 >>> λme → invite_friends
 >>> λthem → let us = zip2 (colors (1 + length them)) [me : them]
             num_us = length us
             in allTasks (repeatn num_us (get randomInt))
 >>> λrs → let gameSt = {middle = repeatn (4 * num_us) []
                      ,players = [ initial_player num_us c
                                    (toString u) (abs r)
                      |\ (c, u)←us & r←rs]
                      in withShared gameSt (play_game us)

currentUser is an SDS that contains a User value describing which user is currently performing the task. randomInt is another SDS that holds random numbers. (withShared v t) creates an SDS with initial value v, and passes it to t. The invite_friends task terminates only with the correct number of friends.

invite_friends :: Task [User]
invite_friends
 = enterSharedMultipleChoice "Select friends to play with" [] users
 >>> λyou → if (not (isMember (length you) [1 .. 3]))
           (viewInformation "Oops" [] "number of friends must be 1, 2 or 3"
           >>| invite_friends)
           (return you)
users is an SDS that contains all known users of the system. A selection of this list can be made with enter(Shared)MultipleChoice.

All players receive a new task to play a game of Ligretto:

```haskell
play_game :: [(Color,User)] (Shared GameSt) → Task (Color,String)
play_game users game_st = anyTask [u @: play (c, toString u) game_st \ (c, u) ← users]
```

anyTask is a parallel task combinator that terminates as soon as one of its sub-tasks terminates. Here, each sub-task, play, is assigned to one of the players, using the task assignment combinator @:

For each player, the game proceeds in two phases. In the first phase, cards are played until one of the participants obtains an empty ligretto pile. In the second phase, the winner receives her accolades.4

```haskell
play :: (Color,String) (Shared GameSt) → Task (Color,String)
play (color,name) game_st = updateSharedInformation name
    [imageUpdate id (player_perspective color) (const st)] game_st
>>> [OnValue (game_over color game_st)]
where
    game_over me game_st (Value gameSt _) = case and_the_winner_is gameSt of
        Just {color, name} = let won = (color, name)
            in Just (accolades won me game_st >>| return won)
        _ = Nothing
```

The play task is an editor enhanced with the player perspective function that has been developed in Section 7.3.3. This task edits an SDS because all players manipulate the same middle cards and want to see the progress of their opponents at the same time. Players play simultaneously, but only their own cards are click-able and can be played in any order. The model functions presented in Section 7.3.3 guarantee that only legal moves can be made. Race conditions may occur, e.g. when two players want to play their card on top of the same middle pile. This is automatically solved by the shared system on a first-come-first-serve basis. The move of the second player is ignored. The step combinator >>> continuously checks the current value of the game state (that is manipulated by all players in parallel) to determine whether one of the players has obtained an empty ligretto pile, and if that is the case, proceeds with the accolades task. This terminates the entire play task (and therefore also the anyTask application in play_game).

Finally, to convince all other players that the winner has won fair and square, not only her name is announced, but also the entire game state. To

---

4This is a simplification of the rules of the game in which the remaining points need to be calculated. For brevity we omit this.
disallow further editing of the game state, it is merely rendered as a view.

accolades :: (Color, String) Color (Shared GameSt) → Task GameSt
accolades won me game_st
    = viewSharedInformation ("The winner is " ++ won)
        [imageView (player_perspective me)] game_st

7.3.4 Discussion

Due to the expressive power of the iTasks editors and combinators, the definition of an interactive graphical oriented game such as Ligretto can be given in a concise declarative style. Static images can be turned into interactive ones by adding pure functions to (sub)images. No complicated mouse detections algorithms are needed to find out what has been clicked, it does not matter how the (sub)images are being transformed or used. It is clear that being compositional is a desirable property for an image library. However, it is commonly not so easy to realize this. The implementer needs strong support from the underlying graphical library.

7.4 Implementation

In this section, we explain how images are incorporated in iTasks’ architecture (Section 7.4.1). We give an introduction to SVG and briefly evaluate its strengths and weaknesses (Section 7.4.2). Finally, we discuss how we generate SVG from images (Section 7.4.3).

7.4.1 Customizable interactive tasks

iTasks has a client-server architecture. Commonly, interactive tasks run as client in the browser while the coordination and communication between the tasks is handled by the server. Type driven generic functions are used with which form based editors can be generated for any first order type. As we have seen in Section 7.3.1, one can also specialize such an editor for a specific concrete type. One can even define rich client tasks, by using editlets [42], which can be thought of as an embedded client-side JavaScript application.

An editlet consists of two parts: one part of the editlet runs on the server (in native code) while the other part runs on the client (just-in-time compiled to JavaScript). Each part maintains its own state. A diff-based synchronization mechanism keeps the two states synchronized. Whenever the client receives a new diff, it has the ability to execute arbitrary JavaScript code. Editlet programmers do not write JavaScript code directly, but use a foreign function interface and a sophisticated cross-compilation mechanism from Clean to JavaScript [39]. This mechanism allows us to execute any Clean function in the
Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics

As a consequence, it is possible to write almost all code in one single language. We can decide at run-time which tasks and functions to execute on the server, and which to execute on the client.

In order to integrate interactive images in iTasks, we have created an SVG editlet which synchronizes an image’s model value on the server with the client, after which the client renders the image and enables it to respond to on-click events.

7.4.2 SVG: Introduction, strengths, and weaknesses

SVG is a plain-text, XML-based markup language that describes vector graphics. It has been explicitly designed to work well with existing browser technologies, such as JavaScript, CSS, and the DOM. At the moment of writing this paper, SVG 1.1 Second Edition is the most recent published version of the specification. This version is largely supported by all modern mainstream browsers.

SVG has facilities for drawing both arbitrary shapes and text. For the former, it features one primitive shape: the path. A path is a sequence of individual path segments, which can either be straight or curved. All other shapes can be defined in terms of a path, although that would be cumbersome in practice. For that reason, SVG defines several basic shapes: rectangle, circle, ellipse, line, polygon, and polyline. Each of these basic shapes is represented by an SVG XML element. A shape’s dimensions are specified with attributes on the shape element itself.

SVG also has facilities to render text, which is different from path-based shapes in that text is a sequence of font glyphs, specified in plain-text, rather than a sequence of paths. Font properties, such as the font family and font weight, are specified textually as SVG attributes on the text element. As a consequence of the way SVG implements text, one cannot determine the exact width of a piece of text until it is inserted into the browser’s DOM and is rendered, even if all font properties have been specified. This is due to the fact that rendering text relies on the font definition being available on the client. If the client does not have the specified font, it chooses a fall-back font. The fall-back font may have different glyph-widths than the specified font, resulting in a different text-width. This makes images containing text harder to render with predictable results.

A collection of shapes can be grouped using the group element <g/>. These shapes can then collectively be identified, transformed, interacted with, or attributed with certain properties.

All shapes can be styled by specifying properties on the individual elements. All shapes, except path, can be positioned relative to the current coordinate system by specifying x and y properties.
Chapter 7

Shapes can be transformed using a transformation matrix. For convenience, however, SVG provides specific transformations: translation, scaling, rotation and skewing.

SVG is largely compositional by itself. Individual shapes can be drawn and positioned independently from others. However, this compositionality is lost when rotation transformations are applied; when rotating an image, its axes rotate along with it. Any subsequent transformations, such as translations, then act relative to these rotated axes. As a consequence, first rotating an image around its center and then translating it yields a different result than first translating the image and then rotating it around its center. Figure 7.3 shows the problem graphically.

![Figure 7.3: SVG rotation and translation in different orders](image)

Square A is the original square. Square B is our desired result and is what we get after first translating square A along the x-axis and then rotating it 45 degrees around its center. However, when we first perform the rotation and then the translation, we end up with square C. We compensate for this behavior by wrapping an image in a group element immediately after it is rotated. Any subsequent transformations are then applied to the group, rather than the original shape. This effectively resets the image’s axes, allowing us to obtain result B, regardless of the order in which the transformations have been applied.

Transformations also pose specific challenges for text, because rotation and translation are always performed relative to an image’s origin. In all other SVG elements, the origin is situated in the element’s top-left corner. For text elements, however, the origin is situated on the left of the text’s baseline, as is illustrated in Figure 7.4.

As a consequence of the different origin, we need to compensate when translating or rotating a piece of text. To do so accurately, we require at least the font’s ascent and descent heights. However, the current SVG specification does not provide an API to obtain these metrics. A common workaround to this problem is to count pixels of a text glyph on a raster-based canvas. We choose a simpler approximation: we assume that the ascent and descent heights are 75% and 25% of the text height, respectively. While this heuristic has worked reasonably well in practice so far, it is far from a general solution.
Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics

7.4.3 Generating SVG

Since a text’s width cannot be known until it is inserted into the DOM, we are forced to interact with the browser during SVG generation. Because of this, we choose to execute all parts of the rendering process on the client. We have created an SVG editlet which synchronizes the model value between the server and client, turns that model value into an image on the client, then calculates the text widths, and finally renders that image as SVG. This process is illustrated in Figure 7.5.

Even with known text-widths, images can still contain lookup-spans which we need to resolve and reduce to pixel values, before we can generate SVG. Several iterations may be needed until we arrive at a fix-point and have resolved all lookup-spans. In the worst case, this process can diverge. When we have converged on a fix-point, SVG is generated and inserted in the DOM.

Generating SVG code is simplified by desugaring the internal image structure. All grids and overlays are desugared to collages, as shown in Figure 7.6. We then only have to concern ourselves with rendering SVG for collages.

To translate overlays to collages, we first calculate the spans of all sub-images, after which we determine the spans for the largest image in the overlay, or the span of the host image, if present. We then calculate the offsets required to align all images relative to these spans, and add them to the offsets manually provided by the image programmer.
toSVG (Overlay aligns offsets images host) =
  let allSpans = getAllSpans images
       (maxX, maxY) = getMaxSpans allSpans host
  in toSVG (Collage positionOffsets images host)
toSVG (Grid offsets aligns images host) =
  let spanss = getAllGridSpans imagess
         offsets = calculateGridOffsets
                   (getColumnXSpans spanss)
                   (getRowYSpans spanss)(aligns imagess offsets)
  in toSVG (Collage flatten offsets flatten imagess host)

Figure 7.6: Outline of the SVG conversion algorithm.
These offsets are then used to express the overlay as collage. Translating a grid to a collage is a bit more involved. First, we obtain a list of lists of the spans of the individual images in the grid layout. Each list in the outer list represents one row, while each index in the inner lists represents one column. To calculate the offsets of each cell, we first obtain the $x$- and $y$-spans of each row and column. These spans are determined by the widest and highest cell in each row and column. Each cell’s offset is calculated by adding the dimensions of previous cells together, keeping into account the alignment and manual offsets that each cell has. We end up with a list of lists of offsets, which we then flatten to obtain the list of offsets required to form a collage. Figure 7.6 omits the implementation details of basic images, since they have a one-to-one correspondence to basic SVG shapes.

7.4.4 Discussion

Choosing SVG as rendering mechanism has the advantage that images are inherently scalable and are viewable in any modern browser. However, it also poses new problems.

The plain-text nature of SVG introduces problems with rendering fonts, because not all font metrics required for positioning text are available in the SVG API. Future SVG standards will likely address these problems. Additionally, we wish to add support for embedded fonts. Currently, we cannot guarantee a particular font is available on the client. With embedded fonts, we can. Both SVG 1.1 and CSS 3 support embedding fonts. An additional benefit is that we can always calculate the width of text snippets server-side if an embedded font is used, thereby eliminating the need to calculate text widths on the client.

Another problem is due to the fact that we are currently computing images completely on the client. This is significant slower than doing so on the server, because JavaScript is an interpreted, garbage-collected language, which has to work with limited heap space. We frequently trigger JavaScript’s garbage collector while evaluating Clean expressions. This is due to the fact that the representation of our client-side runtime system heavily uses arrays, which it frequently creates and destroys, creating garbage on the JavaScript heap. In practice, these slowdowns make it infeasible to play a game of Ligretto on slower machines, because the computational lag can be as much as one full second. We reduce this problem by firstly reducing the size of the span-expressions as much as possible during their construction. This is not always possible, however, due to the presence of lookup-spans. Secondly, we make the client-side computations as strict as possible, eliminating unnecessary thunk evaluation. Still, these are only optimizations, rather than actual solutions. We want to pursue three solutions to this problem. Firstly, we want to generate all SVG on the server, so that we only need to send a string of SVG to the client. This requires first calculating all text widths on the client, requiring us to
implement a rendering protocol. Currently, however, the editlet infrastructure does not allow for implementing protocols, so the infrastructure will need to be extended. Secondly, we want to completely eliminate the standard JavaScript garbage collector from the editlet runtime and replace it with our own. This approach is advocated by the asm.js [15] initiative, which is a highly optimizable subset of JavaScript. Pursuing this solution, we also want to generate low-level, asm.js-style JavaScript instead of the high-level, human-readable JavaScript we are currently generating. Thirdly, we want to do partial updates to the images, so that only the parts that have changed need to be recalculated and redrawn.

7.5 Related work

Peter Henderson’s Functional Geometry (FG) is a seminal approach to purely compositional images [71]. Henderson states [72] that the design principle “...was based on contemporary views of what was good practice in declarative systems”. Similar to FG, we always specify the layout of sub-images relative to each other. Unlike FG, we do not abstract from ‘size’ (or rather, span boxes, in our terminology, because we regard images to be infinitely large). For instance, in FG, the span boxes of \( \text{beside}(p, p) \) and \( p \) are equal. In Graphics.Scalable (and most other approaches), the span box of \( \text{beside}([], [], [p, p], \text{Nothing}) \) has twice the width of the span box of \( p \). In FG, overlaying images consists of taking the union of graphic elements ([72] Section 5) which is a sensible choice because the primitive elements are (curved) lines only. Any approach that supports (partially) filled shapes must make the order of rendering of graphic elements explicit, either via ordering the graphics operations (typically on a canvas-model) or via a stacking concept. We have chosen the latter route and separate stacking images (z-axis) from specifying their relative layout (x- and y-axes). This idea can be traced back, although in a very different way, to Haggis [55, 56], in which piles of widgets (i.e. common user-interface elements, such as text fields) are created monadically and put in containers separately to control their layout along the x- and y-axes. At the risk of diverging, it should be mentioned here that this solution has been adopted in other GUI approaches, viz. Object IO [10], TkGofer [25], and wxHaskell [91]. More recently, the Diagrams approach by Brent Yorgey [135], very explicitly deals with stacking using lists and monoids as organizational principle of structuring the library. Diagrams features an elegant way of placing images besides each other using their outlines instead of bounding boxes. However, Diagrams is restricted to non-interactive images only, and the other approaches do not offer the usual graphical transformations such as rotation, scaling, and skewing on widget-like components. One of the advantages of using SVG as graphics back-end is that it extends to both graphics and widgets. Arbitrary HTML can be embedded in SVG document using the <foreignObject> element, after which it can
be arbitrarily transformed like all other SVG elements.

The layout combinators of Graphics.Scalable were inspired by the Racket image API [115, 53], which has a mature, but rather baroque, API for the compositional specification of images. For instance, for the specification of layout, it features 22 functions. In contrast, Graphics.Scalable has 1 core layout function, collage, and 5 derived combinators (Section 7.2.4). These are sufficient to model all Racket image layout combinators, and more, as the Racket API does not support the grid-combinator. In addition, we profit from the orthogonality of the SVG back-end in that we can support flipping transformations, which is restricted to images without text in Racket. The Racket image API is bitmap-oriented and offers features such as manipulating bitmaps directly, extracting color-lists and bitmaps from images, ‘freezing’ images, and defining a pragmatic equality relation that is based on the current bitmap pixels. Except for the ability to embed bitmaps in SVG, the other features do not match naturally with the vector graphics philosophy. Both Racket and SVG offer elements that have not yet been transferred to Graphics.Scalable (both: Bézier curves; Racket: pinholes; SVG: paths, gradients, and filtering). We conjecture that they can be added to Graphics.Scalable without compromising its design principles.

An entirely different view on images is taken by Conal Elliot et al in their work on Pan [48], enhancing it with interaction, resulting in Fran [49] which gave birth to the paradigm of functional reactive programming (FRP) and, amongst others, Yampa [29, 77]. Characteristic to these approaches is to consider images as functions from coordinates to a well-defined range (Pan and Fran), animations as functions from continuous time to images, and interactive applications as functions from discrete events to animations (Yampa). A recurring theme in their work is that specifications are functions from a continuous domain to a discrete domain. The implementation ‘samples’ these functions. This differs greatly from our approach that advocates a ‘structurally-analytic’ view on image specifications and embedding in TOP to define behavior.

Another different path has been taken by Magnus Carlsson and Thomas Hallgren in their work on the Fudgets system [24]. Just like FRP and TOP, it features combinators to structure the top-level behavior of the interactive application. The basic elements are the fudgets which conceptually behave as typed value-transformers at their API-side, abstracting from the concrete way they work. This is also the key difference with iTasks and TOP that features task abstraction that processes a value. Images can be programmed in Fudgets using an approach that is similar to the Pictures abstraction that is used in the above mentioned Haggis system [56].
7.6 Conclusions and future work

We have presented an image library and have integrated it with iTasks to allow the creation of distributed, multi-user, web applications with custom-built interactive, graphical user interfaces. The image library is implemented on top of SVG, produces interactive scalable vector graphics, and can be used in any modern browser. An important property of the image library is that it is purely compositional, both for static and interactive images.

The Ligretto case study demonstrates how graphically based multi-user tasks can be defined in a concise way, offering a good separation of concerns to the programmer. This involves three separate stages: first, one concentrates on modeling the game's domain, using pure data structures and pure functions; second, one defines the graphic visualization as functions from this domain to image values; third, one defines the application behavior as an iTask and integrates visualization within editors. We have observed this same pattern of working in an earlier experiment [9] that, at that time, did not have the refined SVG support as Graphics.Scalable. We are going to investigate the generality of this application design pattern.

The current implementation suffers from severe performance issues of the generated client-side JavaScript code. We want to address this problem by generating asm.js-style code, replacing the garbage collection by our own, and moving calculations from client to server where possible. Early experiments that perform a round-trip to the client to measure text widths, but render the SVG on the server show promise of greatly improved performance.

Our event model is currently limited: interaction is restricted to the single model type of the entire image, and the event model is restricted to on-click events only. We want to investigate how to define and combine interactions on sub-images. We need additional ways of interacting with images such as drag-and-drop, double-click, and right-click, but also keyboard input. We want to explore more complex forms of interaction, such as touch gestures. The challenge in incorporating these interactions is that they must not compromise the way of working and thinking of the Graphics.Scalable library.

As mentioned in the introduction, we are using the library to draw Tonic diagrams. In these diagrams, individual nodes are connected with edges. Tonic's diagrams are simple enough that we can compute these edges in a straight-forward manner. However, this is not the case in general. Therefore, we want to introduce the concepts of connector points (which can be attached to an image), and include automatic edge routing between these connector points.
Part IV

On the side
Parametric lenses: change notification for bidirectional lenses

Most complex applications inevitably need to maintain dependencies between subsystems based on some shared data. The dependent parts must be informed that the shared information is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible. To achieve this, one must be able to define, with arbitrary precision, which party is depending on which data. In this paper we offer a general solution to this general problem. The solution is based on an extension to bidirectional lenses, called parametric lenses. With the help of parametric lenses one can define compositional parametric views in a declarative way to access some shared data. Parametric views, besides providing read/write access to the shared data, also enable to observe changes of some parts, given by an explicit parameter, the focus domain. The focus domain can be specified as a type-based query language defined over one or more resources using predefined combinators of parametric views.

8.1 Introduction

Complex applications commonly have to deal with shared data. It is often confined to the use of a relational database coupled with a simple concurrency control method, e.g., optimistic concurrency control [90]. In other cases, when a more proactive behavior is required, polling or some ad hoc notification mechanism can be invoked. At the farther end of the range there are some very involved applications (multi-user applications, workflow management systems, etc.), which are based on interdependent tasks connected by shared data. In the most general case, one has to deal with complex task dependencies defined by shared data coming from diverse sources, e.g. different databases, shared memory, shared files, sensors, etc.

As an example, consider the following case which is based on a prototype we have developed for the Dutch Coastguard [93]; it is used throughout the paper to introduce the problem, and the concepts of the proposed solution. We have
a small database which acts as a source of data of ships: name, cargo capacity, last known position, etc. The positions of the ships are updated repeatedly as the ships move; ships have a transponder on board which sends their latest position on a regular basis. As a basic task, we simply want to show the positions of the ships on a map, of which users are allowed to select a region to view, the focus of their interest. In this setting we can think of map instances and update processes as interdependent tasks that are connected by the data of ships they share. When the position of a ship is updated in the database, the map instances, of which focus covers the old or the new coordinates, must be refreshed.

From a theoretical perspective, it would be correct behavior to notify every map instance on every ship movement. However, this leads to huge efficiency issues in practice. There are many thousands of ships in the North Sea constantly moving around. Only those map instances need to be refreshed in which region the position of a ship is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible. Thus, we need a notification system which, for efficiency reasons, can be as accurate as needed for optimal efficiency.

As the problem described above is a very common computational pattern,
we would like to offer a general, reusable solution.

From the computational perspective, focusing on a specific domain of the underlying data can be achieved by creating and working with one of its abstract views. Lenses [58, 21, 74, 18, 60, 132, 75, 59] are commonly used for creating abstract views. They can be used to support partial reading and writing, for access restriction or to provide a specific view of the data. Lenses enable to define bidirectional transformations. In a nutshell, a lens describes two functions to map the input to an output and backwards.

In our example two kind of abstract views are needed for serving different processes: one to show the ships located in a given region of the map, and another one for the update process, which periodically updates the coordinates of a ship in the database.

The general notification problem is depicted in Figure 8.2. Given is a set of shared data sources of any type (A and B in the picture) holding a set of data ($D_A$, $D_B$). There are also given some lenses defined on top of the data sources and on each other. These are $L_1$, $L_2$, $L_3$ and $L_4$ in the picture. The additional subscripts of the original data sets, $D_A$ and $D_B$, denotes the sets of data we gain after applying a series of lenses to the original data sets (e.g. $D_{A,L_2}$ denotes the set of data that can be seen from $A$ through $L_2$). One typical question can be, e.g., whether a given update through $L_4$ affects the $D_{A,L_1}$ or not? What about the other way around?

Unfortunately, classical lens theory does not provide any tool to discover whether a given update through some lens affects the data that can be seen through another lens. In this paper we present a general extension to lenses as a solution for this general problem. In this extension, called parametric lenses, lenses are partially defunctionalized to extract a first-order parameter, the focus domain, that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain
is mapped to the output domain by the lens. This additional focus information will enable to read, update, and observe specific parts of the underlying data.

Parametric lenses are pure, thus cannot be applied to some shared data directly, they must be lifted into an impure context. Therefore, they are attached to the shared data through a non-pure abstract interface called parametric view. The parametric views are allowed to be composed using predefined combinators. Using these combinators, one is able to specify the focus domain as a type-based query language defined over one or more resources. With the query language, one can focus on a specific part of the underlying shared data during reading, writing, or it can be used for notification purposes.

We use two examples throughout the paper to present our solution. The first example is based on the simplest form of parametric views and it is compact enough to give a nice insights in the main idea; it shows how to find a node, by some property, in an arbitrary tree structure. The selected node can be used then not only for reading or updating, but also for observing its changes.

The second example, our motivating one, is slightly more complex, and requires the introduction of additional combinators. For its development we parametrize some relational lenses developed in [21] for solving the so called view-update problem.

We offer the following contributions in the paper:

1. We introduce parametric lenses as a general extension to bidirectional lenses. Parametric lenses enable the development of efficient notification systems based on them. Parametric lenses are embedded into compositional parametric views which are defined over shared data;

2. We implement the executable semantics, using Haskell [108], of the combinators and an underlying notification engine. The complete Haskell implementation, along with the examples developed in the paper, can be found at https://wiki.clean.cs.ru.nl/File:PViewIFL.zip;

3. We develop two examples in the paper to demonstrate the usage of parametric lenses. An introductory example based on a simple recursive data structure, and a simplified real world example based on the iTasks coastguard prototype described above.

The remainder of this paper is structured as follows: in Section 8.2, after a brief overview of classical lenses, the parametric extension is introduced in Section 8.3. In Section 8.4, we introduce parametric views. In Section 8.5, the realization of the parametric and the classical, non-parametric variants of lenses, in the setting of parametric views, are contrasted. The first, introductory example is developed in Section 8.6. Then, before we proceed with more advanced cases, a new combinator is introduced in Section 8.7 to be able to join views of different data sources together. Using this combinator, our second,
motivating example is developed in Section 8.8. In Section 8.9, an alternative implementation is provided of the second example to increase the accuracy of the notifications. It is followed by a discussion of related work in Section 8.10 and concluding remarks in Section 8.11.

The executable semantics and the examples are written in Haskell [108], and they are also dependent on some extensions of the Glasgow Haskell Compiler (GHC) [100] and its libraries. The given implementation and example code uses the following language extensions and libraries: generalized algebraic datatypes (GADT) [62], the Data.Typeable package [34], monads [130] (and in particular the State [130, 119] and Writer [82, 134] ones), monad transformers [23] and applicative functors [101]. Their basic knowledge is necessary for the comprehension of the paper.

8.2 Introduction to Lenses

The starting point for this work is the class of bidirectional transformations known as lenses. Thus, in this section a brief overview of lenses is given to explain what they are, and how they work.

Lenses enable the definition of bidirectional transformations. In a nutshell, a lens describes two functions to map the input (or source: \(X\)) to an output (or view: \(Y\)) and backwards. The \textit{get} function maps the input to some output, while the \textit{put} function maps the modified output, together with the original input, to a modified input:

\[
\begin{align*}
\text{get} & \in X \rightarrow Y \\
\text{put} & \in Y \times X \rightarrow X
\end{align*}
\]

Lenses are expected to obey the following “round-tripping” laws for every \(x \in X\) and \(y \in Y\):

\[
\begin{align*}
\text{put (get x)} x & = x & \text{(GETPUT)} \\
\text{get (put y x)} & = y & \text{(PUTGET)}
\end{align*}
\]

These laws express fundamental expectations about how the components of a lens should work together. The \texttt{GETPUT} law (also known as \textit{consistency} [132]) ensures that all updates on a view are captured by the updated source, while the \texttt{PUTGET} law (also known as \textit{acceptability}) prohibits changes to the source if no update has been made on the view. Lenses obeying these laws are called \textit{well-behaved} [58].

Sometimes a third law, called \texttt{PUTPUT}, is also considered. For every \(x \in X\) and \(y, y' \in Y\):
\[ \text{put } y \left( \text{put } y^\prime x \right) = \text{put } y x \]  
(PutPut)

This law states that the effect of a sequence of two \textit{puts} is just the effect of the second. Well-behaved lenses which also satisfy the PutPut law, are called \textit{very well-behaved}.

In the next section we go beyond the classical theory and parametrize lenses. With a parametrized lens, we can focus on a specific part of the underlying data for reading, writing and observing.

### 8.3 Introduction to Parametric Lenses

In the parametric lens extension classical lenses are partially defunctionalized to extract a first-order parameter (the \textit{focus domain}: \( \Phi, \Psi \)) that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain is mapped to the output domain by the lens. Parametric lenses additionally return a predicate in the \textit{put} direction. This predicate, called the \textit{invalidation function}, encodes the semantic information associated with the focus domain.

\[
\begin{align*}
get_F &\in \Phi \times X \to Y \\
\text{put}_F &\in \Phi \times Y \times X \to X \times (\Phi \to \text{Bool})
\end{align*}
\]

The invalidation function tells whether the particular update with some focus affects a given other focus from the same domain or not. To illustrate the role of this function, consider the following sequence of operations (\( \phi, \psi \in \Phi, x, x' \in X, y, y', z \in Y \) and \( \text{inv} \in \Phi \to \text{Bool} \)):

\[
\begin{align*}
y &= \get_F \phi x \\
(x', \text{inv}) &= \text{put}_F \psi z x \\
y' &= \get_F \phi x'
\end{align*}
\]

We say that the invalidation function \textit{inv} is \textit{consistent} if \( y \neq y' \Rightarrow \text{inv} \phi = \text{True} \). If \( y \neq y' \Leftrightarrow \text{inv} \phi = \text{True} \), we say that \textit{inv} is \textit{accurate}. Consistency is a fundamental property, all invalidation functions must satisfy it. It expresses the fundamental requirement that all the actual changes can be observed, but also allows false notifications. Accuracy however is not obligatory and, especially when different focuses can overlap, may not be implemented effectively in practice. Nevertheless, during the development of parametric lenses, one should aim for accuracy to effectively reduce the number of triggered notifications (as an example, the constant True function satisfies consistency).
Parametric lenses: change notification for bidirectional lenses

With the extended functions, the round-tripping laws take the following form for every \( \phi \in \Phi \), \( x \in X \) and \( y \in Y \):

\[
\begin{align*}
\text{fst} \left( \text{put}_F \phi \left( \text{get}_F \phi \ x \right) \right) &= x \\
\text{get}_F \phi \left( \text{fst} \left( \text{put}_F \phi \ y \ x \right) \right) &= y
\end{align*}
\]

(getput) (putget)

The \text{PutPut} law is the following for every \( \phi \in \Phi \), \( x \in X \) and \( y, y' \in Y \):

\[
\text{put}_F \phi \ y \left( \text{fst} \left( \text{put}_F \phi \ y' \ x \right) \right) = \text{put}_F \phi \ y \ x
\]

(pupput)

The elements of a given focus domain, e.g. \( \phi \in \Phi \), are called the focuses. The semantics of the focuses are encoded by the invalidation function, and there is no general restriction on them; they can denote an arbitrary part of the underlying data, and, for instance, they can also overlap in an arbitrary way.

In our example, the focus domain can be the set of possible contiguous regions of the map. The elements of this domain, the focuses, are just individual regions of the map, which can overlap. It is the task of the invalidation function to predicate whether two values of the focus domain, two regions, actually overlap or not.

In the next section we introduce parametric views to implement this idea. In this implementation focus domains take the form of types, and the actual focuses of the domain are values of this type.

### 8.4 Introduction to Parametric Views

A parametric view is a compositional abstract interface to provide access to some shared data in a general way. The interface enables the underlying data to be read, written and observed according to a domain of focus. In this section we present the basic structure of parametric views and extend it later in the paper.

A parametric view is represented by the \((\text{PView} \phi \ m \ a)\) generalized algebraic data type (GADT), see Figure 8.3. The type parameter \( \phi \) is the focus domain, the type of the focus parameter, which must be Typeable (this requirement is explained later on in this section). The type \( m \) is any monad which provides access to the underlying shared data. Finally, \( a \) is the type of the values that can be read from and written into the view.

The \text{PView} data constructors play the role of combinators. With the combinators, one can create a view based on some shared data (the \text{Source} combinator, introduced later on in this section), change the focus domain or the associated read/write type of the view (\text{Focus} and \text{Project} combinators, see
Section 8.5), or one can combine views to synthesize a compound one (Product combinator, see Section 8.7).

The interface of a parametric view consists of three functions: the read and update functions for reading and writing the view, and the observe function to ask for change notifications:

```haskell
type ObserveId = String

read :: (Monad m, Typeable φ) => PView φ m a -> φ -> PViewT m a
update :: (Monad m, Typeable φ) => PView φ m a -> a -> PViewT m ()
observe :: (Monad m, Typeable φ) => PView φ m a -> φ -> ObserveId -> ObserveHnd m -> PViewT m ()
```

The first two arguments of these functions are the parametric view of type \(PView φ m a\) and the actual focus value of type \(φ\). The observe function further takes a unique identifier of the subscription (ObserveId, a string value for the sake of simplicity), and an event handler function (an action in the monad \(m\)). The ObserveId argument is used e.g. to remove the subscription later.

The interface functions are monadic, they are based on the \(PViewT\) monad transformer, which basically introduces a state. The definition of this \(PViewT\) type, along with the explanation of its purpose is given later on in this section.

Type \(m\) is constrained to be a Monad, but also used as an applicative functor to provide a more applicative style implementation. It is assumed that Applicative => Monad.

Figure 8.3: The PView type
The Source type describes how to interface with the actual data source in the associated monad. A parametric view is created from a Source description by the Source data constructor of the PView GADT (see Figure 8.3). A Source is already parametric, thus the provided access functions require a focus value:

```haskell
data Source φ m a where
  MkSource :: (Monad m, Typeable φ) => (φ -> m a) sread (φ -> a -> m (Invalidate φ)) supdate -> Source φ m a
```

The sread function takes a focus and reads data according to that specific focus. The supdate function updates the part of the underlying data which is denoted by the focus value in the first argument. The new data is given in the second argument. After the update, it returns the invalidation function, discussed in the previous section.

```haskell
type Invalidate φ = φ -> Bool
```

Reading a Source is straightforward, as the request is just forwarded to the underlying data source (the other alternatives of the read function are given later when the related data constructors of the PView GADT are introduced in detail):

```haskell
read (Source (MkSource sread _)) p = lift (sread p)
```

The update and observe interface functions are more elaborate as those are involved in the notification process. The idea is to save the notification requests in a state monad (the associated functions of the state monad are qualified with ST in the paper), then lookup and trigger the event handlers of the matching ones during updates.

The state of the notification engine is introduced by the PViewT monad transformer, and it keeps the list of notification requests. The observe interface function maintains the list of requests in the state monad, which is used by the update function to trigger notifications.

```haskell
type PViewT m = StateT [NRequest m] m

— a notification request consists of an id, an event handler, — and the observed focus which is encoded in a Dynamic
data NRequest m = NRequest ObserveId (ObserveHnd m) Dynamic
```

To save a request we need a comparable reference to the observed view; as the view contains functions only, saving itself would be pointless. Obviously, a machine address based identification is not feasible in a pure functional language,

173
and generating identifiers is not viable either since we want to define the views in a pure, declarative way; for generating unique identifiers, a state would be required. We could require the developer to provide an identifier along with the views. However, we would like two views, defined at two different parts of the application using the same combinators with same arguments, to be equal. That would be error prone with explicit view ids.

However, the type of the focus domain is a perfect candidate for identification as it assigns a unique semantic information to the view. It is realized using the standard GHC extension, `Data.Typeable`. This extension enables us to associate type representations to (monomorphic) types, then, e.g., compare these representations. This is the reason why the types of the focus domains have to be `Typeable`. Note, however, that using the focus domains for identification, imposes a uniqueness requirement on them: focus domains must be unique between the semantically different parametric views.

In accordance, the reference is encoded as a value of type `Dynamic`; the associated type is the actual reference to the view, while the associated value is the focus value to be observed.

With the exception of the `Source` data constructor, a parametric view is recursively defined by the help of other parametric views, creating in this way a directed acyclic graph. As this graph is created in a pure way, it does not have back edges. When a given node is updated, its parents cannot be informed to evaluate the notification requests which are in their focus domain.

```haskell
observe :: (Monad m, Typeable \( \phi \)) => PView \( \phi \) m a \rightarrow \phi \rightarrow ObserveId \rightarrow ObserveHnd m \rightarrow PViewT m ()
observe v p oid ohnd = do
  rs <- genreqs v p -- generate implicit notification requests
  ST.modify (map (NRequest oid ohnd) rs ++) 
```

To overcome this limitation, the `observe` function puts implicit notification requests on all of the views of the subgraph denoted by the target view. During an update, the most adequate of these, the one which has the least distance to the updated view, is triggered. These implicit notification requests are generated recursively by the `genreqs` function.

The `genreqs` function, when applied to the `Source` data constructor, straightforwardly creates a one-element list storing the focus value provided for the `Source`:

```haskell
genreqs (Source _) p = return [toDyn p]
```

Similarly to the `observe` function, the `update` interface function creates notification events of type `NEvent m` for all the views of the subgraph denoted by the updated parametric view (by `update'`); then, in a subsequent step, it matches up these events with the stored notification requests (by `trigger`,

174
Parametric lenses: change notification for bidirectional lenses

defined further on). Please notice that for the sake of providing concise code, the Writer monad transformer is exploited in update and update’; its associated functions are qualified with \( W \) in the paper.

\[
\text{update } v \ p \ a = W.\text{execWriterT} (\text{update'} v \ p \ a) >>= \text{trigger} . \text{reverse}
\]

The update’ function traverses the graph of views in a recursive manner during an update. It returns the invalidation function of the given view, which is used to create a computed one in the recursion step. It also generates a notification event for every view on the paths to the roots and stores them in a Writer monad. The actual update happens at the Source views in the roots, but all the invalidation functions computed for the non-leaf views have further information on the exact range of affected data.

\[
\begin{aligned}
\text{data } \& \text{NEvent } & m = \exists \phi \ \text{Typeable } \phi \Rightarrow \text{NEvent TypeRep } (\text{Invalidate } m \ \phi) \\
\text{update’ :: } & (\text{Monad } m, \text{ Typeable } \phi) \\
& \Rightarrow \text{PView } \phi \ m \ a \rightarrow \phi \rightarrow a \rightarrow \text{WriterT } [\text{NEvent } m] \ (\text{PViewT } m) \ (\text{Invalidate } m \ \phi)
\end{aligned}
\]

The update’ function, when applied to a Source, just forwards the request to the underlying data source:

\[
\text{update’ } (\text{Source } (\text{MkSource } _ \text{supdate})) p \ a \\
= (\text{lift}.\text{lift}) (\text{supdate } p \ a) >>= \text{Ainval} \rightarrow \text{returnE } p \ \text{inval}
\]

\[
\text{returnE } p \ \text{inval} = W.\text{tell} [\text{NEvent } (\text{typeOf } p) \ \text{inval}] >> \text{return } \text{inval}
\]

Finally, in the trigger function, the notification events generated by update’ are matched with the stored notification requests. It matches every event with every request, but for a given observation id, it takes the most adequate (which happens to be the first match), and the remaining are skipped:

\[
\begin{aligned}
\text{trigger :: } & (\text{Monad } m) \Rightarrow \text{[NEvent } m] \rightarrow \text{PViewT } m \ () \\
\text{trigger es} & = \text{do} \\
& \text{rs} \leftarrow \text{ST.get} \quad \text{— read notification request} \\
& \text{foldM_ } (\lambda \text{skips } \text{event} \rightarrow \text{foldM } (\text{match } \text{event} ) \text{skips } \text{rs} ) \ [] \ es
\end{aligned}
\]

\[
\begin{aligned}
\text{match } & \text{a request to with an event} \\
\text{event handlers are executed only once } & \text{using the skips list}
\end{aligned}
\]

\[
\text{match } (\text{NEvent } \text{vid } \text{inval}) \text{skips } (\text{NRequest } \text{oid } \text{ohnd } \text{dyn}) \\
| \text{notElem oid } \text{skips } \&\& \text{dynTypeRep } \text{dyn} == \text{vid} \\
\text{=} \text{case } \text{fromDynamic } \text{dyn } \text{of} \\
\text{Just } p & \rightarrow \text{when } (\text{inval } p) \ (\text{lift } \text{ohnd}) >> \text{return } (\text{oid}:\text{skips}) \\
\text{Nothing} & \rightarrow \text{return } \text{skips}
\]

\[
\text{match } _{\text{skips}} _{\text{}} = \text{return } \text{skips}
\]
Chapter 8

The basic structure of the notification engine is now sketched out. Based on this foundation, the semantics of the remaining combinators are introduced in the consequent sections step by step as they are needed for the development of the examples.

8.5 Lens combinators

In this section parametric lenses are introduced in the context of parametric views in parallel with the introduction of classical lenses. In this way the two concepts, parametric and non-parametric lenses, can be easily contrasted. With parametric lenses, we are able to develop our first example in the next section.

There are many ways to represent lenses in a functional language like Haskell. For example, the lenses in the popular Control.Lens GHC package [88] are Laarhoven style lenses [127] implemented as functional references based on applicative functors. We could use the same representation for classical lenses, however it is not directly clear how the parametric variant would work with this technique. This is the main reason that we decided for a straightforward, record based representation. Another reason is that in our experience, the classical representation as a pair of functions is easier to comprehend.

Classical lenses

A classical lens is represented by the \texttt{Lens} record type and applied to a view by the \texttt{Project} combinator of the \texttt{PView} GADT. When a lens of type \((\texttt{Lens} \ a \ b)\) is applied to a parametric view of type \((\texttt{PView} \ \phi \ m \ a)\), the resulting view (type of \((\texttt{PView} \ \phi \ m \ b)\)) has the same focus domain inherited from the underlying view. This is expected as the classical theory has nothing to do with focus domains. The types of \texttt{get} and \texttt{put} components of the \texttt{Lens} record type also straightforwardly reflect the classical theory:

\begin{verbatim}
data Lens a b = MkLens {
    get :: a -> b,
    put :: a -> b -> a
}
\end{verbatim}

When a \texttt{Project} node of a parametric view is read, the value read from the underlying view is mapped by the \texttt{get} function of the lens. Similarly, in the write direction, the underlying view is read first, then the \texttt{put} function is used to incorporate the write value into the underlying data, which is updated in a final recursive step.

\begin{verbatim}
read (Project v l) p = get l <$> read v p
update' (Project v l) p a
    = lift (read v p) >>= \s -> update' v p (put l s a)
\end{verbatim}
The Project nodes are simply ignored by the genreqs function as they do not contain any additional focus information:

```
genreqs (Project v _) p = genreqs v
```

### Parametric lenses

The parametric variant of the classical lens is represented by the record type `ParametricLens` and applied to a view by the `Focus` combinator. The combination of a view of type \((PView \phi m a)\) and a lens of type \((ParametricLens \psi a b)\) results in a parametric view of type \((PView (\phi,\psi) m b)\). That is, using the focus domain of the parametric lens, the focus domain of the initial view is refined.

The components of the `ParametricLens` record type, the `getf` and `putf` functions, in accordance with the formal introduction, extend the classical functions with focus information and with the invalidation function:

```
data ParametricLens \psi a b = MkPLens {
    getf :: \psi -> a -> b,
    putf :: \psi -> a -> b -> (a, Invalidate \psi)
}
```

When a `Focus` node of a parametric view is read, first the underlying view is read recursively using the corresponding element of the focus value, then `getf` is applied to the read value along with the second element of the tuple of the focus value (this is the only difference compared to the classical case):

```
read (Focus v l) (p,q) = getf l q <$> read v p
```

The update logic is more different, compared to the classical case, when a `Focus` node of a parametric view is updated. First the original input value is read recursively from the underlying view using the corresponding element of the focus value. After that, `putf`, the put part of the parametric lens, is applied to this original input value and the provided new output value along with the other part of the focus value to compute a new input value; it also returns a partial invalidation function, covering the \(\psi\) type. Then, the underlying view is updated recursively using the newly computed value; the update process also returns another partial invalidation function, covering the \(\phi\) type. Finally, the two partial invalidation functions are combined together to cover the whole focus domain: it states that the validation can be safely decided solely by the invalidation function coming with the parametric lens, if the focus related to the underlying view is the same in the update and in the notification request.
update' (Focus v l) (p,q) a = do
  s <- lift $ read v p
  let (s', invall) = putf l q s a
  invalw <- update' v p s'
  returnE (p,q) (comp invall invalw)
where
  comp invall invalw (p',q')
  | p == p' = invall $ q'
  | otherwise = invalw p'

In genreqs, an implicit notification request for the Focus view is generated prior to the recursion step:

```hs
    genreqs (Focus v _) p = (toDyn p :) <$> genreqs v (fst p)
```

Above we gave the formal semantics of the most important combinators. In the following sections we give a more intuitive insight into their behavior through a series of illustrative examples.

### 8.6 First example: Self service storage

Let us consider that we own a company which provides self-service storage for its customers. The customers hire storage space, or just let us store some objects for them. The storage space is structured in a hierarchical manner: there are multiple buildings which contain rooms, the rooms have shelves or lockers which may contain boxes with some objects or the objects directly. The actual hierarchy is very flexible, and varies between buildings and rooms.

We would like to develop a piece of software to maintain the locations and properties of the objects we handle. The software will be based on the rose tree data structure as it is simple, but still flexible enough to describe any necessary hierarchies. The element type of the tree describes a container which has a type, properties and also can contain object items. Items have properties like name, quantity, etc.

--- The store is a hierarchy of containers

```hs
    data RoseTree a = RoseTree a [RoseTree a]
    type Store = RoseTree Container
```

--- Containers are a collection of properties and object items

```hs
    data Container = Container CType [Property] [Item]
    data CType = Building String | Room String | ... 
    data Property = Owner String | ... 
    data Item = Item {name :: String, ... 
```
There are two important requirements for the system: (1) we want to be able to find a container by an arbitrary property, and (2) we also want to be able to observe the changes of an arbitrary container or subtree. An example for the latter is that we may want to be informed when a specific object is removed from a container. In the following, we show how to implement these requirements based on parametric views.

```
type StoreT = monad transformer for accessing storage data

type StoreView q = PView q StoreT Store

— unique focus domain for storage data

data StoreSource = StoreSource deriving Typeable

store :: StoreView StoreSource
store = Source (MkSource sread supdate) where ...
```

First a base view must be implemented to provide access to the underlying data in a monad. The view has a simple, flat focus domain, StoreSource, to identify the source of the data which is available in the monad StoreT. The base view, the store function is created using the Source combinator (the definition of this function is simple, but depends on the actual monad, which is not important for the main idea, and thus not included here).

The implementation is based on a single parametric lens which enables to go one level down in the hierarchy by focusing on one of the children of a tree. The lens has the Selector focus domain with one data constructor to provide the index of the child:

```
data Selector = S {unS :: Int} deriving Typeable

selectLens :: ParametricLens Selector Store Store
selectLens = MkPLens {getf = get, putf = put} where

  get (S i) (RoseTree _ cs) = cs !! i
  put (S i) (RoseTree r cs) c =
    (RoseTree r (take i cs ++ [c] ++ drop (i + 1) cs),
     \(S j) -> j == i) — invalidation function
```

In the get direction, one of the children is selected straightforwardly based on the current focus value. The put direction is also straightforward, the indicated child is replaced with the given subtree; still, the invalidation function requires some consideration.

As we want to go deeper and deeper in the hierarchy, selectLens must be applied again and again, thus creating a wider focus domain. For example, applying the lens twice to the base view, we get the following:

```
store 'Focus' selectLens 'Focus' selectLens
:: StoreView ((StoreSource, Selector), Selector)
```
Chapter 8

This view enables us to select any child from the second level by providing a specific focus value e.g. ((StoreSource, S 1), S 0) selects the first child of the second child of the root element. This example also gives us the insight that two different values from the same focus domain always select different parts of the underlying data, in other words there is no overlapping. In this case, deciding whether a given update affects a focus or not, simply reduces to comparing the focus values. This is encoded in the invalidation function returned by put. In the ship example developed later on, this property does not hold, which makes the invalidation functions non-trivial.

Finally, the function which finds a container based on a predicate function is as follows:

```haskell
findFirst :: Typeable q => StoreView q -> q -> (Container -> Bool) -> PViewT StoreT (Result)
findFirst v q pred = do
  (RoseTree container children) <- read v q
  if pred container
    then return $ Just $ StoreResult v q
    else case children of
      [] -> return Nothing
      _  -> searchChildren (length children)
  where
    searchChildren 0 = return Nothing
    searchChildren i = do
      res <- findFirst focusView (q, S (i-1)) pred
      case res of
        Nothing -> searchChildren (i-1)
        _        -> return res
    focusView = v 'Focus' selectLens
```

```haskell
data Result where
  Result :: Typeable q => StoreView q -> q -> Result
```

First, the container of the root element is checked, then its children one by one if the container did not satisfy the predicate. For checking the children, the `selectLens` is applied to the view to be able to focus on one more level deeper, then the `findFirst` function is called recursively.

The only part to be considered is the return type. To be able to point out a specific part of the shared data, we need to return a view which has the proper focus domain, and a focus value of the same type. The main problem here is that the type of the focus domain is not known in advance as it depends on the actual structure of the data, and on the location of the sought container inside it.

Thus, the return value of the function cannot be typed directly. However,
it is not even necessary: the API functions depend on the \texttt{Typeable} property of the focus domain only. This idea is encoded in the \texttt{Result} type. After the values from the \texttt{Result} data constructor are unwrapped, they can be used to read or update the returned view, use some combinators on it, or to observe the view.

### 8.7 Joining multiple sources

Before we continue with our main example, one more combinator must be introduced. A very common situation is where the data we are working on, is coming from multiple sources. In the aforementioned example of ships, there are two sources of data, one for the actual ship data and one for the current positions of the ships. It is also common that the data coming from the different sources is dependent; e.g. the position data provides additional properties of the ships. In this case, one may want to create a joint view based on the individual views e.g. a relational join of the ships and positions.

This can be achieved using the \texttt{Product} combinator. This combinator takes two views and simply tuples their focus domains and view types together. It only requires that the views share the same monad type.

When a \texttt{Product} view is read, the two underlying views are read one by one, then the results are simply put together in a tuple:

\[
\text{read (Product } vl \; vr) \; (p, q) = (, ) \; \langle \rangle \; \text{read } vl \; p \; \langle \rangle \; \text{read } vr \; q
\]

During updating, first the underlying views are updated, then a new invalidation function is created which fires when at least one of the invalidation functions, resulting from the recursive updates, fires:

\[
\text{update'} (\text{Product } vl \; vr) \; (p, q) \; (a, b) = \text{do}
\]

\[
\begin{align*}
\text{invall } & \leftarrow \text{update'} vl \; p \; a \\
\text{invalr } & \leftarrow \text{update'} vr \; q \; b \\
\text{return} & \; \lambda (p, q) -> \text{invall} \; p \; || \; \text{invalr} \; q
\end{align*}
\]

In \texttt{genreqs}, no notification event is generated for the combined view, only for the underlying ones. It is not necessary as the underlying views generate their own events which cover the whole focus domain of the combined one.

\[
\text{genreqs (Product } vl \; vr) \; (p, q) = (++) \; \langle \rangle \; \text{genreqs } vl \; p \; \langle \rangle \; \text{genreqs } vr \; q
\]

In the following section, our main example is developed, which gives an intuition on how to use this combinator in practice.
Chapter 8

8.8 Second example: Filtering ships

At this point we have all the tools to pick up the main exercise where we left it in the introduction. As a recap, we have two databases holding ship data and location data. We develop the following views (to develop our motivating example, we could use the second one for updating the positions of ships, and the last one to acquire the data of ships located in a specific region):

- Which ships have a given cargo capacity?
- Where is a given ship?
- Which ships are in a given region of the world?

The purpose of these views is not only to read and update the shared data, but we also would like to observe the data behind them (to be notified when the data of a view has changed).

The structure of the implementation is the following. All the views are built on two base views which access the data sources. First, filtering parametric lenses are applied to the base views. In the next step, a joint view is created from the filterable views using the Product combinator introduced in the previous section. Finally, the joint view is turned into a view of natural join of the ship and position records using a classical, non-parametric lens.

We work with lists of records, which makes the problem very similar to the classical view update problem well-known in the database literature [17, 67]. The view update problem arises from the fact that when the view update is translated to a database update, there exist more than one database update that may correspond to the same view update.

It is out of the scope of this paper to deal with this problem, but fortunately, the view update problem is already investigated in the context of bidirectional lenses [58, 21]. Thus, the lenses in this section are based on the relational lenses developed in [21].

The ship and position data is defined by the following record types:

```haskell
data Ship = Ship
    { s_name :: String
    , s_capacity :: Int
    }

data Position = Position
    { p_ship_name :: String
    , p_position :: Coord
    }

type Coord = (Double, Double)
```
The base views, the `ships` and `positions` functions, use these types and flat focus domains as in the previous example. The implementation of these functions is not included, because it is irrelevant from the perspective of parametric lenses:

```hs
ships :: PView Ships ShipsT [Ship]
positions :: PView Positions ShipsT [Position]
```

— *monad transformer for accessing ship data*

```hs
type ShipsT = -- focus domains
  data Ships = Ships deriving Typeable
  data Positions = Positions deriving Typeable
```

To create filterable views, we use the following generic parametric lens:

```hs
selectLens :: (f -> a -> Bool) -> ParametricLens f [a] [a]
selectLens pred = MkPLens {getf = get, putf = put}
  where
    get = filter . pred
    put f ss vs = (m' ++ vs, inval)
      where
        (m, m') = partition (pred f) ss
        inval f' = any (pred f') (m ++ vs)
```

It is a polymorphic function that creates a parametric lens based on a predicate function. The predicate function decides whether some value of type `a` satisfies some properties, defined by a value of type `f`, or not. Type `f` becomes the focus domain of the parametric lens, while its value type becomes a list of type `a`. With this parametric lens, we can effectively filter a list of data, where the parameter of the filtering is represented by the `f` type.

The implementation of this parametric lens is based on the `selectLens` definition in [21]. In the `get` direction, it simply filters the list using the predicate. In the `put` direction, the original data is split into two: `m` contains those elements of the original list (the source list, `ss`) which are affected by the view, that is, replaced by the view data, `vs`. The unaffected elements are in `m'`. The new source list is the concatenation of the unaffected data `m'` and the view data, `vs`.

The invalidation function encodes the idea that a focus is affected by an update, if any of the replaced or the new records (the list of the unaffected data `m` and the view data, `vs`) satisfy the predicate indicated by the focus value.

The filterable views are easy to define now using the `selectLens` function:
— additional focus domain for Ships

```haskell
data ShipFilter = SNameEQ String
  | CapacityGT Int deriving Typeable

filteredShips :: PView (Ships, ShipFilter) ShipsT [Ship]
filteredShips = ships 'Focus' (selectLens shipPred)

shipPred (CapacityGT c) s = s_capacity s >= c
shipPred (SNameEQ n) s = s_name s == n
```

— additional focus domain for Positions

```haskell
data PositionFilter = WholeWorld
  | PNameEQ String
  | AreaIN (Coord, Coord)
  deriving Typeable

filteredPositions :: PView (Positions, PositionFilter) ShipsT [Position]
filteredPositions = positions 'Focus' (selectLens posPred)

posPred (WholeWorld) = True
posPred (AreaIN ((x,y),(x',y'))) p
  = inside (p_position p)
  where
    inside (a,b) = a >= x && a <= x' && b >= y && b <= y'
    posPred (PNameEQ n) p = p_ship_name p == n
```

The `ShipFilter` and `PositionFilter` focus domains are rather ad-hoc here. The filtering predicates can be arbitrarily complex which makes it difficult to find the proper data type to encode them. Thus one may want to use a complex, generic type to describe filters. In this case the actual filter values can be given by a small DSL like in Groundhog [97]. However, providing any particular tool here would beyond the scope of this paper, and in this simple case, these types are satisfactory for presentation purpose.

The next step is to join the filterable views together using the `Product` combinator:

```haskell
jointView :: PView
  ((Ships,ShipFilter),(Positions,PositionFilter))
  ShipsT
  ([Ship], [Position])
jointView = filteredShips 'Product' filteredPositions
```

Unfortunately, we are not ready yet. The value type of the joint view, `([Ship], [Position])`, is not exactly what we want. When the two lists of records are properly joined together in a relational sense, we expect a value type like `[(Ship, Position)]`. 

184
Depending on the update policy, there are many ways to implement such a relational lens properly [21]. To keep the example illustrative, an oversimplified version of such a lens is given here. Its update policy is that it may add and delete records from both lists, but it is safe only when the relation between the elements of the lists is one to one.

```haskell
joinLens :: (a -> b -> Bool) -> Lens ([a], [b]) [(a, b)]
joinLens by = MkLens {get = get, put = put}
  where
    get (lss, rss) = mapMaybe f lss
      where f ls = (ls,) <$> find (by ls) rss

    put (lss, rss) vs = (lms' ++ lvs, rms' ++ rvs)
      where
        (lvs, rvs) = unzip vs
        lms' = dropPaired pairLeft
        rms' = dropPaired pairRight

    dropPaired as = map fst (filter (isNothing.snd) as)
```

Just like `selectLens`, this function is also polymorphic. It creates a relational join by a predicate, the element types of the lists depend on the argument types of the predicate. This time we create a traditional lens as there is nothing to be parametrized here.

The implementation, in the get direction, takes the elements of the left list and tries to find a connected record (the first such one) from the right list using the predicate function. If there is no such one, the record is skipped.

The put direction is non-obvious even in this simple case. Briefly, we determine which elements from the original lists are skipped by the join (lms’ and rms’). These give one part of the modified input. The other part is coming from the modified output, vs. It is unzipped to separate the elements of the left (lvs) and right lists (rvs). For more details we refer to [21].

```haskell
shipPositions :: PView
  ((Ships, ShipFilter), (Positions, PositionFilter))
ShipsT
[(Ship, Position)]
shipPositions = jointView 'Project' (joinLens by)
  where
    by s = (s_name s ==) . p_ship_name
```

Any of these views can be used to read or update the shared data, depending on the focus one needs. As they are interconnected, updating through
filteredShips for example, triggers notifications for the other affected views, ships and shipPositions as well.

However, one must be aware that the accuracy of the notification system depends on the focus value provided explicitly for the interface functions. As an example, consider the f1 focus value:

\[ f_1 = ((\text{Ships}, \text{SNameEQ} \ "\text{Queen}\"), (\text{Positions}, \text{WholeWorld})) \]

Requesting notification for this focus of the shipPositions view may result in many false notifications. According to the semantics of the Product node, a notification is triggered if the data of the ship “Queen” has been changed in the ship database or any data has been changed in the position database (the WholeWorld focus covers all the records). In this case one should use the f2 focus value for improved accuracy:

\[ f_2 = ((\text{Ships}, \text{SNameEQ} \ "\text{Queen}\"), (\text{Positions}, \text{PNameEQ} \ "\text{Queen}\")) \]

The accuracy of the notification system is also affected by the actual implementation of the used lenses. For example, in this section we took a naive approach developing the lenses which is a hidden source of inaccuracy. Let us consider again that we use the f1 focus value, this time to update the data (capacity and position) of the ship “Queen”. Providing only records related to the ship “Queen”, we expect that we trigger notifications for domains only which contains that ship. The join lens works on the list of records read from the underlying views using the explicit focus value. It means one record from the ship database (the (Ships, SNameEQ "Queen") focus value selects exactly one record) and all the records from the position database (indicated by the (Positions, WholeWorld) focus value). The lens does a good job, it silently replaces the position of the ship “Queen” in the list of position records, then delegates the whole list to the underlying view, filteredPositions. The problem is that from the point of view of filteredPositions, the whole database is changed, thus it triggers notifications for all the notification requests indiscriminately. In the following section we develop a new variant of our generic lenses to overcome this issue.

### 8.9 Revised second example

The source of the problem described in the previous section is that the filtered views cannot take into account the implicit filtering imposed by the join lens. The list of affected records can be easily calculated in joinLens, but how make selectLens aware of this information?

Previously, we calculated the list of affected records in the put function of selectLens and generated the invalidation function based on that. The idea is to calculate this change list in the higher level views and pass it to
Parametric lenses: change notification for bidirectional lenses

selectLens as an argument. Thus the new implementation works on a pair of lists instead of a single list:

```haskell
selectLens :: (f -> a -> Bool) -> ParametricLens f [a] ([a], [a])
selectLens pred = MkPLens {getf = get, putf = put}
  where
    get f ss = (filter (pred f) ss, [])
    put f ss (vs, cs) = (m' ++ vs, inval)
      where
        m' = filter (not . pred f) ss
        inval f' = any (pred f') cs
```

The difference compared to the previous implementation is in the inval function. Instead of calculating, we just use the given change list. Applying it to the ships view we get the following:

```haskell
filteredShips' :: ShipsView (Ships, [ShipFilter]) ([Ship], [Ship])
filteredShips' = ships 'Focus' (selectLens shipPred)
```

This is far from being ideal as it still depends on the change list. However, it can be easily calculated in one more step using an additional generic lens:

```haskell
calcChangeList :: Lens ([a], [a]) [a]
calcChangeList = MkLens {get = get, put = put}
  where
    get (ss, cs) = ss
    put (ss, _) vs = (vs, ss ++ vs)
```

It creates a change list just as selectedLens did previously: the list of new records in addition to the list of replaced ones. The filteredShips view has the same semantics now as previously:

```haskell
filteredShips :: PView (Ships, ShipFilter) ShipsT [Ship]
filteredShips = filteredShips' 'Project' calcChangeList
```

If filteredPositions' is created in the same way and put it together with filteredShips' by means of the Product combinator we get the following view:

```haskell
jointView :: PView
  ((Ships,[ShipFilter]), (Positions,[PositionFilter]))
  ShipsT
  ((([Ship],[Ship]), ([Position],[Position])))
jointView = filteredShips' 'Product' filteredPositions'
```

Its type is rather lengthy, but it is an intermediate view only which is not supposed to be exposed. We still have to modify the joinLens function to calculate and push down a change list. Its type is as follows:

```haskell
joinLens :: (a -> b -> Bool) -> Lens (([a],[a]),([b],[b])) [(a,b)]
```
The actual implementation of this function is not included here, mostly because it is rather long. The modifications are straightforward to develop by \texttt{calcChangeList} and the new \texttt{selectLens} implementation, and it also provided in the online version.

In a final step, the new \texttt{joinLens} must be applied to the new \texttt{jointView} view exactly the same way as previously. This gives back the exact same read and update semantics, but radically improves the accuracy of the notifications in certain cases.

\section*{8.10 Related Work}

There are three main groups of works closely related to parametric views. These are bidirectional lenses, publish/subscribe systems (including their light-weight counterpart in the object oriented world, the observer design pattern) and functional reactive programming (FRP).

\subsection*{Bidirectional lenses}

The first group of related work is bidirectional programming, the so called lenses \cite{58}. A lens is a bidirectional transformation that maps a “concrete” data structure into a simplified “abstract view” and, if the view is updated, maps the modified abstract view, together with the original concrete data, to a correspondingly modified concrete data. In recent years, many extensions (e.g. quotient lenses \cite{60}) and improvements are suggested to the original idea. Most of the improvements would replace the original, \textit{state-based} approach, with a more efficient, \textit{edit-based} one, which work with descriptions of changes to structures, rather than with the structures themselves \cite{18, 21, 132, 75}. As for parametric views, supporting observation of specific changes in the “concrete” data by extended lens definitions, is a novel idea.

\subsection*{Publish/subscribe systems}

With systems based on the publish/subscribe interaction scheme \cite{51}, subscribers register their interest in an event, or a pattern of events, and are subsequently notified of events generated by the publishers. There are three basic variants of publish/subscribe schemes.

The earliest scheme was based on the notion of \textit{topics}. Observers can subscribe to individual topics, which are identified by keywords. A later extension enabled topics to be created in a hierarchial manner, giving more expressive power to the event system. Most systems also allow topic names to contain wildcards, thus enabling to publish and subscribe to several topics in the same time. The most notable of these is TIBCO \cite{125}.
The content-based variant, e.g. Java Message Service [104], extends the expressiveness of the topic-based approach by introducing a subscription scheme based on the actual content of the events. These systems usually offer a subscription language or enables the subscribers to provide a predicate function to filter events at runtime.

Finally, the latest approach, the type-based variant [52] replaces the name based topic classification model by a scheme that filters events according to their type. Subtyping can be used to achieve hierarchical topic descriptions.

Parametric views use an approach that is a mixture of the type- and content-based variants with additional modifications. Events are typed and carry some type specific properties. The semantics of the properties is described by the invalidation function during the definition of the views. An instance of the invalidation function (based on the actual data) is generated by the update process (the publisher) to filter events. In contrast, in content-based publish/subscribe systems, the predicate function is provided by the subscriber. Another difference is that the whole event system is connected, that is two events can be related in our system, even if their type is different. The conversion of events of different types is also described by the views using the predefined combinators.

Publish/subscribe systems are mostly used in inter-process communication. In object oriented programming the observer design pattern [61] is also used for notification purposes when one wants to stay between the boundaries of an application. In the observer pattern, an object, called the subject or observable, maintains a list of observers, and notifies them automatically if its state changes, usually by calling one of their methods.

In contrast to parametric lenses, the observer pattern does not allow the observers to subscribe for a specific focus. Most implementations, however, enable the observable to pass an arbitrary parameter to the observers along with the change notifications. If this parameter describes the nature of the change (the invalidation function in parametric lens terminology), the observer can use it to decide whether it is interested in the actual notification or not. If the observer classes are parametrized over the type of the value passed to the observer (e.g. in the .NET implementation), this also can be achieved in a type safe manner. This behavior is close to parametric lenses, but in this case, the actual notification logic would be decentralized, it should be scattered over all the observable and observer classes, which is error prone to implement.

### Functional reactive programming

The last group of related work is functional reactive programming [49]. FRP is a programming paradigm oriented around time-varying values, called behaviors, or signals, in a functional programming setting. In FRP, the underlying execution model automatically propagates changes, which makes it similar to
our approach. In the original theory a time-varying value is pure, represented as a function of time, thus lenses are not applicable (the data flow is one-way).

There is, however, a state-based approach, when using lenses has a rationale. An example of these frameworks is Scala.React [99], which also can be considered as a superior replacement of the observer pattern. It can help with migrating the observer-based event handling logic with a more declarative implementation.

In Scala.React there are two different kinds of signals: variable signals and expression signals. Expressions signals are restricted to one-way data flow, but variable signals can be edited. This makes it possible in Scala.React to apply lenses to variable signals and to each other, creating a so-called lens cluster [98]. This approach is comparable to parametric views, even though the technical details are very different as the Scala language imposes less restrictions on the developer (e.g. lenses are classes, thus have addresses as comparable identifiers).

Another functional reactive framework, Flapjax [103], is built on JavaScript. Flapjax uses lenses, developed for bidirectional tree transformations in [58], to define user interfaces to JavaScript. A piece of the tree based HTML Document Object Model (DOM) can be attached to some structured data model using these lenses. A signal can be created by connecting an actual part of the DOM (denoted by its DOM id) with an actual model object. The system takes care of propagating the changes of the DOM to the models. This can be considered as a special case of our solution (e.g. the source of the shared data is restricted to the DOM) in an FRP setting.

8.11 Conclusion and Future Work

We have developed parametric views to provide read/write access to some shared data according to a parameter, the focus domain, which describes which part of the shared data one is interested in, wants to focus on. Parametric views are also observable, one can be notified if some part of the underlying shared data is changed, and compositional. They are built up using some combinators either from parametric sources, which describe how to interface with the actual data source in use, or from other parametric views with associated information. To enable observation, we have developed a general extension to bidirectional lenses, called parametric lenses; they are attached to parametric views by one of the predefined combinators. We have presented the executable semantics, a naive implementation of our idea, and some basic examples in Haskell. We also have developed a library in Clean that has been integrated into the iTask system [113]; it is used to develop advanced prototypes for the Dutch Coastguard.

Naturally, the progress we have made on parametric views/lenses raises
many further challenges. The current implementation is naive. For a production environment, efficiency needs to be considered more carefully. Furthermore, as it is shown in Section 8.9, the used lens also affect the performance of the system. We need further research to identify the patterns in the development of lenses which increases the accuracy of the notifications.

Currently, the most traditional state-based lens variant is used in our system, but we also would like to experiment with edit-based lenses [75]. The idea of using the edits easing the creation of invalidation functions is promising, as edits also describe which part of the data is changed.

Another area of further investigation is the design of additional combinators. One direction to be explored is combining two views by connecting the read value of one to the focus value of the other. Although, we are having a reference implementation of such a combinator, it is not clear yet what most practical type parameters for this combinator are.

Finally, we need to investigate how the usage of parametric lenses affects the performance of certain applications. It is obvious, that using parametric lenses, we can reduce the number of status updates in an application. However, the accuracy of the notification system depends on the actual implementation of the used lenses. Furthermore, the more accurate the system, the more computation must be done on some data during the updates. We need a methodology to be able to find a balance between accuracy and computational complexity.
Bibliography


195


Summary

The World Wide Web is one of the important development platforms. Web applications are easy to use and do not need to be installed at the users but readily available to everybody with internet access. With the advent of new technologies and standards, web applications are also getting richer and emphasis has moved from functionality to user experience. To achieve this, more and more code is moved from the server-side to the client-side, making JavaScript, the only commonly available client-side programming language, one of the most used programming languages today.

Web applications are not only getting richer, but, as a consequence, mostly on the client-side, larger and more complex as well. Regarding this, it is extremely unfortunate that JavaScript is infamous for its many dubious language features, including weak typing, an extensive and error prone syntax and the dependence on global variables; its language features do not make it a good candidate to tackle the rising complexity of client-side web applications.

Functional programming, however, especially the strongly-typed, lazy, purely functional languages are considered to be an excellent choice to solve complex problems. What makes these languages suitable for the task is that referential transparency and the strong type system eliminate a major source of bugs, while advanced modularity, fostered by ”the power and elegance of higher-order functions and lazy evaluation”, results in very concise programs.

However, while developing web applications using a functional programming language is a simple task on the server-side, it is challenging on the client-side. It is so because web browsers, the mandatory mediators between the users and web applications, only support JavaScript as client-side programming language.

To foster functional programming in client-side web development, first, there must be a way to execute such a language in a browser. Part I of this thesis introduces a technique for transcompiling the lazy functional language Clean to JavaScript to overcome this problem. The compilation scheme is based on utilizing an intermediate language which has many advantages, e.g. it enables lazy, run-time compilation driven by an arbitrary expression.

The Clean to JavaScript compiler is only the bottom layer of the web development stack. It enables the execution of a Clean program in a browser, but
the compiled program is restricted to very basic user interactions, let alone creating rich content. Part II describes a web component architecture that enables the development of custom, interactive web components in a single language manner. This web component architecture is integrated with iTasks, a Task Oriented Programming framework. In iTasks complex multi-user interactions can be programmed in a declarative style at a high level of abstraction; the iTask system generates the user interfaces by the declarative specification. By integrating it with the Clean to JavaScript compiler and the web component architecture, we made a framework that generates the backbone of the application from a declarative specification, but still enables the development of custom, tailor made, interactive web components when required. Part III presents some advanced use cases to prove the usability of the component architecture.

The last part of this thesis discusses a topic that is a bit different from the rest of the thesis in its theme, but it was inspired by one of the iTasks applications I previously worked with. It introduces parameteric lenses, a general extension to the standard bidirectional lenses that enables the observation of the changes of a specific part of the underlying data. In this extension lenses are partially defunctionalized to extract a first-order parameter, the focus domain, that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain is mapped to the output domain by the lens. This additional focus information does not only enable to read and update but also to observe specific parts of the underlying data. Being able to automatically informed by the changes of the working data can have significant impact on the performance of some type of applications (for example by avoiding polling). In addition to that, being able to observe a specific part of the data, a change that really matters, can further reduce the communication and computation cost of some applications.
Samenvatting

Het World Wide Web is een van de belangrijkste ontwikkelplatforms. Web applicaties kunnen gemakkelijk gebruikt worden en vereisen geen installatie op het systeem van de gebruikers, maar kunnen door iedereen met toegang tot Internet gebruikt worden. Nieuwe technologieën en standaarden zorgen ervoor dat web applicaties een rijkere inhoud krijgen waardoor de nadruk zich verplaatst van functionaliteit naar de gebruiker. Om dit te realiseren wordt een toenemende hoeveelheid programma-code verplaatst van de server naar de clients. Een gevolg hiervan is dat JavaScript, de enige algemeen beschikbare programmeertaal voor clients, een van de meest gebruikte hedendaagse programmeertalen is geworden.

Een gevolg van bovenstaande ontwikkeling is dat web applicaties niet alleen een rijkere inhoud krijgen maar ook groter en gecompliceerder worden. Dit wordt in hoge mate verergerd omdat JavaScript berucht is vanwege diens gebrekkige programmeertaal-eigenschappen zoals een zwak typeringssysteem, uitgebreide en foutgevoelige syntax en de afhankelijkheid van globale variabelen. Deze programmeertaal-eigenschappen dragen niet bij aan het verminderen van de complexiteit van het ontwikkelen van client-side web applicaties. Functionele programmeertalen, en de sterk getypeerde, luie, pure functionele programmeertalen in het bijzonder, worden beschouwd als uitstekende programmeertalen om complexe problemen op te lossen. De kern-eigenschappen referentiële transparantie en sterke typering sluiten een grote bron van fouten uit, en hun geschiktheid om modulaire programma-code te ontwikkelen, mogelijk gemaakt door ”de uitdrukkingskracht en elegantie van hogere-orde functies en luie evaluatie” resulteert in compacte en begrijpelijke programma’s.

Server-side software kan inmiddels ontwikkeld worden met behulp van functionele programmeertalen. De echte uitdaging is het gebruik van functionele programmeertalen aan de client-side omdat web browsers alleen JavaScript ondersteunen als client-side programmeertaal.

Om functioneel programmeren aan de client-side mogelijk te maken zal er ten eerste een methode moeten worden ontwikkeld om functionele programma’s uit te voeren binnen de browser. In deel I van dit proefschrift wordt een techniek
ingevoerd om de luie functionele programmeertaal Clean te transcompileren naar JavaScript. Het compilatie-schema maakt gebruik van een tussentaal die als voordeel heeft dat het compilatie-proces van expressies aangestuurd kan worden tijdens de luie, run-time evaluatie van diezelfde expressies.

De hierboven genoemde Clean naar JavaScript compiler is slechts één schakel in de web ontwikkelketen. Hoewel het hiermee mogelijk wordt om Clean programma’s in de browser uit te voeren, wordt deze beperkt omdat enkel zeer elementaire interactie-mogelijkheden met de gebruiker worden ondersteund, laat staan de gewenste rijke inhoud van moderne web-applicaties. In deel II van dit proefschrift wordt een web component architectuur beschreven die het mogelijk maakt dat rijke, interactieve, web componenten ontwikkeld kunnen worden binnen één programmeertaal. Deze web component architectuur is geïntegreerd met het taak-georiënteerde raamwerk iTasks. In iTasks kunnen complexe, multi-user interacties geprogrammeerd worden op een declaratieve wijze en op een hoog niveau van abstractie. Het iTask systeem genereert hieruit automatisch de gewenste user interfaces. De integratie van iTasks, de Clean naar JavaScript compiler en de web component architectuur resulteert in een raamwerk waarin interactieve, rijke, nieuwe web componenten ontwikkeld kunnen worden naar behoefte. In deel III van dit proefschrift worden een aantal geavanceerde toepassingen hiervan behandeld om de toepasbaarheid van de nieuwe web component architectuur aan te tonen.

In het laatste deel van dit proefschrift worden parametrische lenzen, parameteric lenses, bestudeerd. Hoewel dit onderwerp ogenschijnlijk niets met de rest van dit proefschrift te maken heeft, is de inspiratie hiervoor voortgekomen uit mijn werk met een van de iTask applicaties. Parameteric lenses zijn een algemene uitbreiding van de bekende bidirectional lenses die het mogelijk maken om van een data-collectie alleen de veranderingen binnen een specifiek deel van deze collectie waar te nemen. In de in dit proefschrift besproken uitbreiding worden lenses deels gedefunctionaliseerd ten behoeve van het verkrijgen van een eerste-orde parameter, het zogenaamde focus-domein. Met behulp van het focus-domein kunnen soortgelijke lenzen gegroepeerd worden in één enkele ge-parametriserende lens. De parameter codeert heel precies welk deel van het domein wordt afgebeeld naar het bereik. De toegevoegde focus informatie kan niet alleen gebruikt worden in het lezen en bewerken van gegevens, maar ook om specifieke delen van de onderliggende data te observeren. Deze eigenschappen maken het mogelijk om de performance van sommige applicaties significant te verbeteren (bijvoorbeeld omdat het niet meer nodig is om op gezette tijden de gehele data-collectie te inspecteren). Applicaties kunnen heel precies aangeven welke veranderingen van de data-collectie ze willen waarnemen, waarmee de hoeveelheid communicatie en rekentijd gereduceerd kan worden.
Acknowledgements

I could not write this thesis without the help and support of many people. First of all, I owe a great debt of gratitude to my supervisors Zoltán and Rinus for their invaluable support. I’m also unspeakably grateful to Rinus, Maria-José and Ingrid for all the help with the transition of me and my family to the Netherlands and all the additional support to find our feet here.

All the work in this thesis is related to functional programming. I was first introduced to functional programming at Eötvös Lóránd University and it changed both my personal and professional life in a profound way. My thanks go out in particular to Zoltán and Tamás, but in general to all the people I worked with and used to have fruitful conversations. I particularly need to mention here Attila, Dávid, Barnabás and Péter with whom I developed not only a successful professional but also a meaningful personal relationship.

Coming to Radboud University opened my eyes to a different way of designing and using functional programming languages. During the course of my PhD I have had the pleasure to work with and learn a lot from many people. I want to thank Rinus for his invaluable supervision and to Jan-Martin for introducing me to the theme of this thesis. I also need to highlight the selfless help of Peter with reading my papers and providing excellent feedback, and John for improving my understanding of the implementation of pure and lazy functional languages. My thanks also go out to Jurriën, Bas, Tim, Markus, Steffen, Pieter, and everyone else in iCIS for creating a great environment to work in and by being there when I had questions.

Not only was this document a lot of work to write, it is also a substantial amount to read. My manuscript committee has done so and has expressed their confidence in my work by allowing me to defend it. For this, I would like to thank Ralf, Johan, Clemens and Gabriele.

Finally, I would like to thank my family, Kriszta, Teodor and Igor, for the support and patience during the years, Victor for the great help with fine-tuning the design of the manuscript, and Andi for all the English grammar and spell checking.
Curriculum Vitae

1977  Born on September 8, Budapest, Hungary

1992 – 1996  Highschool, Kalmár László Számítástechnikai SZKI
              Budapest, Hungary

1996 – 2002  Software developer

2002 – 2008  Entrepreneur,
              Programmer Mathematician,
              Eötvös Lóránd University, Budapest, Hungary

2008 – 2011  MsC in Computer Science (cum laude),
              Eötvös Lóránd University, Budapest, Hungary

2011 – 2013  PhD Candidate, Lecturer,
              Eötvös Lóránd University, Budapest, Hungary

2013 – 2018  PhD Candidate, Scientific Programmer,
              Radboud University Nijmegen, The Netherlands