A Cocktail of Tools

Domain-Specific Languages for Task-Oriented Software Development

Jurriën Stutterheim
A COCKTAIL OF TOOLS

DOMAIN-SPECIFIC LANGUAGES
FOR TASK-ORIENTED SOFTWARE DEVELOPMENT

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 17 november 2017
om 10:30 uur precies

door
Jurriënn Stutterheim
geboren op 2 december 1985
te Middelburg
Promotor:

Prof. dr. ir. M.J. Plasmeijer

Copromotoren:

Dr. P.M. Achten
Dr. J.M. Jansen (Nederlandse Defensie Academie, Den Helder)

Manuscriptcommissie:

Prof. dr. R.T.W. Hinze
Prof. dr. S.D. Swierstra (Universiteit Utrecht)
Prof. dr. S.B. Scholz (Heriot-Watt University, Edinburgh, Schotland)

Dit onderzoek, met uitzondering van hoofdstuk 2 is ge-financieerd door de Nederlandse Defensie Academie en TNO via het TNO Defensie Onderzoek AIO programma.

ISBN: 978-94-92380-74-6
Contents

1 Introduction ................................. 1
  1.1 Embedded Domain-Specific Languages .......................... 3
  1.2 Task-Oriented Programming with iTasks .......................... 4
  1.3 Blueprints ........................................ 4
    1.3.1 Static Blueprints .................................. 5
    1.3.2 Dynamic Blueprints .................................. 6
    1.3.3 Generalising Blueprints ............................... 7
  1.4 Thesis Outline ............................... 8

2 Building JavaScript Applications with Haskell ............... 11
  2.1 Introduction .................................... 11
  2.2 Runtime System .................................. 12
  2.3 JavaScript Foreign Function Interface ....................... 15
    2.3.1 The UHC-JavaScript library .......................... 16
    2.3.2 Creating, manipulating and querying objects ......... 17
    2.3.3 Pure objects ...................................... 19
  2.4 The JCU Application ................................ 20
    2.4.1 Implementation Issues ............................. 20
    2.4.2 Performance ...................................... 21
  2.5 Future Work ..................................... 22
  2.6 Related work ..................................... 23
  2.7 Conclusion ....................................... 24

3 Tonic: A Graphical Representation of Tasks .................... 27
  3.1 Introduction ..................................... 27
  3.2 Short Overview of iTasks .............................. 29
  3.3 Static Tonic blueprints ................................ 33
    3.3.1 Generating Tonic blueprints ......................... 33
    3.3.2 Tonic blueprints .................................. 34
    3.3.3 Blueprint production rules .......................... 37
    3.3.4 Discussion ....................................... 42
  3.4 Dynamic Tonic blueprints ................................ 42
    3.4.1 Storing task progress at run-time ................... 42
    3.4.2 Viewing task progress at run-time .................. 44
  3.5 Related work ..................................... 45
  3.6 Discussion and conclusion .............................. 45

CONTENTS
## 4 Purely Compositional Interactive Scalable Vector Graphics 47

4.1 Introduction .................................................. 47

4.2 Compositional Static Images .................................. 49

4.2.1 Image concepts ............................................ 49

4.2.2 Basic images .............................................. 50

4.2.3 Image attributes ........................................... 51

4.2.4 Image composition ......................................... 51

4.2.5 Symbolic span expressions ................................ 53

4.2.6 Image transformations ..................................... 54

4.2.7 Case study: rendering the Ligretto state ................. 55

4.2.8 Discussion .................................................. 61

4.3 Compositional Interactive Images ............................ 62

4.3.1 iTasks essentials ........................................... 62

4.3.2 Enhancing editors with images ............................ 63

4.3.3 Case study continued: interactive Ligretto ............... 64

4.3.4 Discussion .................................................. 68

4.4 Implementation ................................................ 68

4.4.1 Customizable interactive tasks ............................ 68

4.4.2 SVG: Introduction, strengths, and weaknesses .......... 68

4.4.3 Generating SVG ............................................. 70

4.4.4 Discussion .................................................. 72

4.5 Related work .................................................. 73

4.6 Conclusions and future work ................................. 75

## 5 Static and Dynamic Visualisation of Monadic Programs 77

5.1 Introduction .................................................. 77

5.2 Static Blueprints: Examples ................................. 80

5.2.1 Static Blueprints of the I/O Monad ....................... 81

5.2.2 Static Blueprints of the Task Monad ..................... 82

5.3 Building Static Blueprints ................................... 88

5.4 Dynamic Blueprints .......................................... 91

5.4.1 Dynamic Blueprints of the Task Monad .................. 92

5.4.2 Tonic Architecture ........................................ 97

5.4.3 Tonic Wrappers in Action ................................ 101

5.4.4 Dynamic Blueprints in iTasks ............................ 102

5.4.5 The Integrated Tonic Viewer .............................. 102

5.5 Blueprints for All .............................................. 105

5.5.1 Dynamic Blueprints of the I/O Monad ................... 105

5.5.2 Stand-alone Viewer Architecture ........................ 106

5.5.3 Drawing Dynamic Blueprints .............................. 106

5.5.4 Discussion .................................................. 107

5.6 Related work .................................................. 107

5.7 Discussion and Conclusion ................................... 108
6 Towards the Layout of Things

6.1 Introduction ................................................. 111
6.2 The generalized layout language ......................... 113
  6.2.1 Layout language examples ................................. 113
  6.2.2 Implementing the generalized layout language ... 114
6.3 Relating UI components to layout ....................... 117
  6.3.1 UI pattern examples ................................. 118
6.4 The layout of Graphics.Scalable ......................... 119
  6.4.1 Graphics.Scalable domains ............................ 119
  6.4.2 Graphics.Scalable constructor functions .......... 120
  6.4.3 Graphics.Scalable examples ......................... 121
6.5 The layout of ncurses ........................................ 122
  6.5.1 The ncurses domains .................................. 122
  6.5.2 Examples of ncurses .................................. 123
6.6 The layout of Task UIs ..................................... 124
  6.6.1 Task UI domains ...................................... 124
  6.6.2 User interfaces in iTasks .............................. 126
  6.6.3 Task UI constructor functions ..................... 127
  6.6.4 Task UI examples ................................... 130
6.7 Analysis ...................................................... 132
6.8 Related Work ................................................. 133
6.9 Conclusions .................................................. 135

7 Task-Oriented Software Development .................. 137

7.1 Introduction .................................................. 138
7.2 Task-Oriented Programming and Software Development ....... 139
  7.2.1 Domain Modelling Using TFP ......................... 140
  7.2.2 Shared Data Modelling .................................. 141
  7.2.3 Task Modelling ......................................... 142
  7.2.4 User-Interface Customization ....................... 143
7.3 Implementing a C2 System Using TOSD ................. 143
  7.3.1 Modelling a Ship ...................................... 144
  7.3.2 FFDC Simulation ..................................... 149
  7.3.3 Other Application Features ......................... 154
7.4 Related Work .................................................. 155
7.5 Conclusion and Future Work ............................... 157

8 Conclusions ....................................................... 159

8.1 Blueprints ...................................................... 159
  8.1.1 Generating Blueprints .................................. 159
  8.1.2 What Blueprints are Not: Visual Programming .... 160
  8.1.3 Validating Blueprints .................................. 160
  8.1.4 Blueprints for Haskell ................................. 161
8.2 Potential Future Applications of Blueprints .......... 161
  8.2.1 Software Requirements ................................. 161
  8.2.2 Software Verification and Validation ................. 162
Chapter 1
Introduction

Software is all around us. It controls simple devices, such as the alarm clock next to our beds, as well as the powerful smartphones we carry with us and rely on all day. Software can also power even more complex systems than smartphones. Command & Control (C2) systems are complex socio-technical systems with which distributed groups of people and machines can be coordinated, such that they can work together towards a common goal. For instance, the Dutch Coast Guard employs C2 systems to coordinate the work they perform to solve problems at sea, often in cooperation with border patrol and medical services. C2 systems are generally deployed to support complex cooperative tasks, and should be able to respond to dynamically changing situations and unexpected events.

Since the work that is supported by C2 systems is very complex, the C2 software itself is necessarily complex as well. Developing such systems is therefore far from trivial. To successfully develop C2 software, programmers require powerful tools. Such tools should allow them to focus on modelling the complexity of the real world in software, rather than forcing them to overcome possibly necessary, but mundane technical hurdles. This thesis is motivated by the challenge of developing C2 systems for military purposes, in particular systems for use by the Royal Netherlands Navy. We set out to develop a prototype of a part of a C2 system with the goal of further developing the powerful tools needed for this job.

At the centre of this toolset is the general-purpose application framework called iTasks, which is an implementation of a novel functional programming paradigm called Task-Oriented Programming (TOP). iTasks is implemented in the lazy, purely function programming language Clean, and can be used to implement a wide range of software, from alarm clocks to C2 systems. With TOP, programs are specified in terms of tasks. This allows TOP programmers to think about the work that either a human or a machine needs to perform to complete the tasks in the system. When a programmer has created a task specification, a fully working, distributed web-application is generated from this specification. This absolves the programmer from worrying about the implementation details of such an application. Administrative code, such as setting up a web server, managing client-server communication with AJAX, providing live updates to the user-interface in response to changing data, HTML rendering, and many others are take care of by the TOP implementation.

A C2 system is a suitable case study for identifying areas in which the toolset can be improved because of the domain’s complexity, and because the work that is supported by a C2 system can be seen as the tasks that humans and machines have to perform to achieve their goals. With this interpretation, the central concepts in TOP and C2 align well with one another.

Our C2 prototype allows the user to create a crude ship model using a graphical...
CHAPTER 1. INTRODUCTION

editor, and use that model to simulate fire-fighting and damage control (FFDC) scenarios. In such scenarios, multiple people with different roles work together to fight the fires on board that may be the result of battle damage. For example, the system allows officers to prioritize fires and direct sailors towards them to extinguish them.

To figure out which features the C2 system should have, we communicated frequently with non-technical domain experts from the Navy. Particularly in such a complex domain, this is a non-trivial task. There always exists a conceptual gap between the ideas that people have and the implementation of software. This gap hinders mutual understanding between programmers and domain experts, so for successful communication, this gap needs to be bridged somehow. With its focus on the conceptual level of tasks, TOP already narrows this gap to some extent, but it is still a formalism aimed only at programmers. Hence, we want to reduce the conceptual gap even further.

By introducing additional tools that leverage the concept of tasks, we want to facilitate that additional reduction. Both programmers and non-technical stakeholders should have tool support that aids in creating mutual understanding. For this, we developed a tool that generates a graphical representation of the task specification that programmers have created. We call these graphical representations blueprints. Blueprints aim to leverage a person’s intuitive ability to understand pictures to help them understand the programs that have been written by programmers. Blueprints can be either static or dynamic. Static blueprints are a direct graphical representation of the original program code. Dynamic blueprints are blueprints that are augmented with run time information, such as which task is currently being worked on.

Creating blueprints comes with many challenges. One such challenge is the act of rendering the blueprints on screen. Since blueprints are generated, we do not know up-front what they will look like. Declarative programming techniques are helpful in those situations, so we developed a fully declarative image library to draw interactive Scalable Vector Graphics (SVG). Particularly the interactive nature of the images requires interaction with the browser’s DOM, for which JavaScript must be used. Ideally, interaction with the DOM is programmed in the same language as the images and the rest of the application, so we explored ways to compile functional languages to JavaScript and perform interaction with the DOM and other parts of the JavaScript world.

Finally, all of these pieces could come together to create the C2 prototype application. Since this application will also be used by non-technical end-users, its user interface must be appealing as well. We opted to generalise the layout language we had developed for the image library and used it to specify layouts for iTasks user interfaces as well. With so many new tools available, we had to study how they all fit together. The result is a development process we call Task-Oriented Software Development (TOSD). In TOSD, several orthogonal aspects of the TOP application development process are identified, enabling a good separation of concerns, leading to improved testability and an improved ability to develop the software with a team.

The work in this thesis truly is a cocktail of tools. What connects these tools, in
addition to TOP and TOSD, is that they are all related to the concept of embedded domain-specific languages (eDSLs) [57]. eDSLs are programming languages for a particular domain that are embedded as a library in an existing host language. This has the advantage that the programmer needs to learn only one programming language. iTasks is an eDSL for TOP that is embedded in Clean, and blueprints are a graphical representation of this eDSL. The image library is an eDSL for specifying images, while the layout language is an eDSL for layouts. Since eDSLs are such a central concept in this thesis, we will first dive deeper into this subject.

1.1 Embedded Domain-Specific Languages

Typed purely functional programming languages are particularly well-suited for creating eDSLs. For one, the type system can ensure a neat separation between the eDSL and the host language based on types. This gives the developer of the eDSL a large degree of control over the way in which values from the host language enter the eDSL and vice versa. Secondly, the syntax of purely function languages like Clean and Haskell is very minimalistic. Their white-space sensitive syntax eliminates the need for the curly braces and semicolons often encountered in imperative programming languages. The lack of such superfluous syntax gives the eDSL developer more control over the look and feel of the code. Lastly, purely functional programming languages have very powerful abstraction capabilities. This allows the eDSL to be very powerful and complex from the inside, but be very simple to use at the same time.

Generally speaking, there are two ways to embed a domain specific language into a host language: using a shallow embedding or using a deep embedding [17]. Both approaches are used in this thesis. When using a shallow embedding, language constructs from the host language can be used to construct programs in the domain-specific language. A deep embedding precludes the use of many language constructs and functions from the host language. Instead, commonly only constructs from the DSL and algebraic data types may be used to define programs. In practice, this means that a deeply embedded DSL is implemented as an algebraic data type and generally constructs a tree, which is later evaluated by an interpreter explicitly written for the DSL. A shallowly embedded DSL, on the other hand, is directly evaluated by the host language itself.

Both approaches to embedded DSLs have their merits. Shallow embedding may reduce the learning curve for the programmer using the DSL, because the programmer can use familiar language constructs to write the program. The resulting program typically is constructed in such a way that the compiler can effectively apply its optimizations, speeding up the DSL as well. The downside of using a shallow embedding is that it becomes more difficult to implement a different semantics for the same program specification. Doing so will quickly rely on language features like overloading or generic programming.

Deep embedding has the advantage that the entire program becomes tangible in the interpreter. This allows domain-specific optimizations and analyses to be implemented as functions over the program tree. Additionally, by writing different
CHAPTER 1. INTRODUCTION

interpreters for the DSL, the same program can easily be given different semantics. However, in order for the code to be efficient, the DSL developer may need to manually apply program transformations. A trade-off when using a deep embedding is that one loses expressivity compared to a shallow embedding [13].

Which style of embedding to use greatly depends on the requirements one has for the DSL. In this thesis, our main focus is on shallowly embedded DSLs. iTasks is one of them. The languages for SVG and layout are examples of deeply embedded DSLs. First, we take a brief look at iTasks.

1.2 Task-Oriented Programming with iTasks

iTasks is a shallowly embedded DSL that implements TOP in Clean. Clean is a strongly and statically typed purely functional programming language. Tasks in iTasks have the type Task.

Computations of type Task can be composed sequentially using the so-called step combinator (>>*), and in parallel, using iTasks’ parallel combinator. Task programmers typically use combinators derived from these primitive combinators. An example of a derived combinator for sequential task composition is the bind combinator (>>=). It executes its left-hand side task and passes the resulting value on to the right-hand side task. An example of a derived combinator for parallel task composition is (-||). It executes two tasks in parallel, but returns only the value of the left-hand side task. Below we see an example of an iTasks program that supports a mission to counter a pirate threat on a drilling platform on the North Sea.

```
clearPlatform :: (Task Mission) -> Task Mission
clearPlatform planMission = planMission >>= \mission -> ((getSupplies >>= \supplies -> stockShip supplies) -|| (gatherCrew >>= \crew -> briefCrew crew)) >>= \stocked -> if stocked (execute mission) (clearPlatform planMission)
```

In this example, the higher-order task clearPlatform is parametrized by another task called planMission. Whichever task is passed to clearPlatform as planMission is executed first in this example. When the planMission task is completed, it yields a value of type Mission, which we bind in the variable mission. Next, supplies and crew are gathered in parallel, after which the ship is stocked and the crew is briefed. Once the ship has been successfully stocked, the mission is executed. If stocking is unsuccessful, the clearPlatform task recurses.

1.3 Blueprints

Blueprints are a graphical representation of the code that programmers have written. A blueprint is generated from the code, and it captures both the control flow and data flow of a program on the level of tasks. This results in a flowchart-like
rendering, an example of which is shown in Figure 1.1. The boxes represent tasks, while the arrows represent sequential composition.

Blueprints can be both static and dynamic. Static blueprints are blueprints that only show the program structure, but do not contain any run time information when the program for which they are generated is executed. Dynamic blueprints have the option of instantiating blueprints and overlay run-time information to view in which computation the program currently resides in its execution.

Blueprints are not generated as bitmaps or vector images, but rather as a serialized abstract syntax tree (AST) of something akin to a core calculus as commonly found in a compiler for purely functional programming languages. In a sense, blueprints can also be seen as an eDSL. They can be seen as lifting a shallow embedding of any eDSL, including the host language itself, into a deep embedding, making it available for interpretation, enabling the assignment of alternative semantics to the program.

1.3.1 Static Blueprints

Figure 1.1 shows a blueprint of the clearPlatform task we saw in the previous section. In the static blueprint, the planMission task in the body of the function is rendered as a variable, signified by a dashed border. The grey box on the arrow coming out of the planMission task represents the variable that is bound in the lambda on the right-hand side of the bind combinator. The boxes with solid edges and rounded corners represent tasks. If a task function is applied to any arguments, they are displayed in a box beneath the box containing the task’s name. Tasks that are executed in parallel are grouped in a clearly marked container, representing the parallel combinator. The lines touching the right-hand side of the container graphically show which task values are returned from the parallel combinator. Task function names are displayed on the top of the box. Conditional expressions and case blocks are represented by a diamond shape. The grey boxes on the arrows coming out of a diamond represent the patterns in the case blocks, or in this case the true and false branches in the conditional. Recursive tasks are rendered as regular task application nodes. This blueprint is generated from the program code and rendered in the blueprint viewer application, which itself is implemented in iTasks.

In order for blueprints to actually add value to communication with domain experts, they need to have certain properties. For one, they need to contain
CHAPTER 1. INTRODUCTION

1.3. Dynamic Blueprints

Static blueprints represent the static structure of a program. It is an alternative representation of the code programmers have written. While static blueprints may already be useful in documenting code and for communication with less technically inclined people, they lack the rich body of information that is available at run time.

Dynamic blueprints are blueprints that include run time information. For example, we may want to know which tasks are currently being executed and by whom, and give the corresponding nodes in the blueprint a different colour. Perhaps we also want to inspect active tasks and observe its current task value. With this kind of information, we can start to view dynamic blueprints as a novel graphical tracing/debugging tool for iTasks programs.

Figure 1.2 shows an example of a dynamic blueprint that illustrates these
features. Starting from the top, we see that the task is being executed by Alice. We are looking at an instance of the blueprint that has the unique identifier [1, 34]. In this example, the `clearPlatform` task is applied to a task called `planAll`. The `planMission` variable node is therefore replaced by a concrete task node for this task at run time. A small square next to a task allows for inspecting the task’s current value. Clicking it opens an inspector window, an example of which is shown in Figure 1.3. Tasks composed in a parallel are not necessarily executed at the same time, as is the case here. The left branch of the parallel task execution has already completed the `getSupplies` task, while the right branch is still busy with the `gatherCrew` task. Blue task nodes indicate that tasks are done and will not change any more. Green task nodes represent active nodes of which the value can still change.

Our choice to create blueprints in the Clean compiler makes implementing dynamic blueprints a challenge. Implementing them requires us to relate the static world at compile time to the dynamic world at run time. To achieve this, we looked at applying program transformations at compile time. Additionally, we need to be able to deal with complex situations at run time, such as parallel tasks and higher-order tasks.

### 1.3.3 Generalising Blueprints

We would like to be able to generalise the idea of blueprints in order to make them usable outside the context of iTasks as well. Ideally, we should be able to generalise both static and dynamic blueprints. To do so, we need to identify the class of programs for which this can be done.

A well-known concept in functional programming is that of monads. Monads provide a means to sequentially compose computations using the bind combinator (\(\gg\gg=\)). Values can be lifted into the monadic domain by using the `return` function. A type is a monad when there is a definition of bind and `return` for that type that fulfils the monad laws \[103\]. The `Task` type in iTasks does so and is therefore a monad.

Rather than just generating blueprints for the `Task` monad, we want to generate blueprints for all monads. While generalising static blueprints to arbitrary monads is straightforward, lazy evaluation makes it hard to generalise dynamic blueprints. We need to carefully identify the class of monads for which this is possible. This issue is revisited in Chapter \[8\].

---

![Figure 1.3: Blueprint value inspection](image)
1.4 Thesis Outline

We have documented the construction of the C2 prototype, including the layouts and its separation of concerns, the concept of blueprints and how they are implemented and drawn, in several papers. Below we list the chapters of this thesis and how they relate to the publications on which they are based. Chapter 8 concludes by reflecting on the work in this thesis and by looking towards the future.

Chapter 2: Building JavaScript Applications with Haskell

We needed a way to use the Graphics.Scalable library in the browser. To do so, we had to be able to interact with the HTML DOM, for which we used an approach similar to the one used in the paper “Building JavaScript Applications with Haskell” [31], upon which this chapter is based. We explore a way to run Haskell in a browser while maintaining its lazy semantics. Of particular interest in this chapter is a new way to use Haskell’s foreign-function interface (FFI) constructs to import JavaScript functions, even when they are members of a JavaScript object. With these new FFI capabilities, we are able to bridge the conceptual gap that exists between object-orientation and purely functional programming. My main contribution in this work is the new FFI syntax for interfacing with JavaScript objects. Additionally, I wrote most of the paper, I programmed most of the application we used for the case-study, and I developed a large part of the standard library used by the case-study application.

Chapter 3: Tonic: A Graphical Representation of Tasks

This chapter is based on the paper “Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks” [98]. We explore for the first time how we can visualize the structure of iTasks applications in the form of blueprints. As we have seen in Section 1.3, generating blueprints is complicated by the fact that iTasks is a shallowly embedded domain-specific language and we want to include elements of the host language Clean in the blueprints. Rendering the blueprints in the browser is done using existing JavaScript libraries, which are interfaced with using techniques based on those presented in Chapter 2. Most of this chapter is my own contribution.

Chapter 4: Purely Compositional Interactive Scalable Vector Graphics

Rendering blueprints in a browser can be challenging. This chapter is based on the paper “Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics” [6]. It presents a fully declarative image library, written in Clean, that makes it easier to render interactive scalable vector graphics (SVG). The declarative nature of the library makes it easier to render arbitrary blueprints. However, our desire to enable the programmer to program all images in Clean imposes several challenges. These challenges require us to heavily involve the user’s browser in the SVG rendering process. My primary contribution to this work is the translation from the image language to SVG. Additionally, I have
made some contributions to the conceptual design and the API design, and I have contributed several paragraphs to the final paper.

Chapter 5: Static and Dynamic Visualisation of Monadic Programs

iTasks uses a monadic API, hinting at the possibility to generalize Tonic to arbitrary monads. This chapter is based on the paper “Static and Dynamic Visualisations of Monadic Programs” [99]. It generalizes Tonic to be able to generate static blueprints for all monads, while showing dynamic blueprints for both iTasks and the IO monad. Doing so requires more general solutions to the challenges already encountered in Section 1.3 and Chapter 3. In this chapter, Tonic’s graphics have been updated to work with the Graphics.Scalable library, as presented in Chapter 4. Both the implementation of the generalised Tonic system and a large part of the writing are my contributions to this chapter.

Chapter 6: Towards the Layout of Things

One of the contributions of this thesis is a prototype of a part of a C2 application. During the development of that application, we wanted to apply a custom layout to the application, but we lacked an intuitive language to express this layout. The Graphics.Scalable library already had a layout language that we liked, so we generalized it. This chapter, based on the paper “Towards the Layout of Things”, shows how we did so and how we use this more general language for laying out the graphical user interface elements of iTasks programs, as well as command-line programs based on the ncurses library. Each of these different domains behaves slightly differently. For example, SVG images are constructed in a bottom-up fashion, whereas iTasks user-interfaces are constructed in a top-down fashion. The challenge then is to find the right language to describe these different kinds of layout in one language, ideally enabling layout-reuse at the same time. My main contribution is the generalization of the layout language to a set of type classes. Additionally, I ported the type classes to Haskell and implemented layouts for the ncurses library. Lastly, I also wrote part of the iTasks layout implementation, and I wrote parts of the paper.

Chapter 7: Task-Oriented Software Development

Using the graphics library from Chapter 4 and the layout language from Chapter 6, we can develop iTasks programs incrementally while maintaining a good separation of concerns. This chapter is based on the paper “Maintaining Separation of Concerns Through Task-Oriented Software Development”, for which we have yet to find a suitable venue for publication. The chapter shows how separation of concerns can be used to systematically construct iTasks programs by using a new engineering approach called Task-Oriented Software Development (TOSD). The demonstrator application developed in this chapter allows the user to design a model of a ship and simulate fire-fighting and damage control simulations. This application is not trivial. It supports many users interacting with the same shared data at the same time. It also provides a rudimentary decision support
system that allows reasoning about the consequences of damage on board of a ship. My contributions are a significant part of the application shown in the paper, the name Task-Oriented Software Development, the explicit identification of the TOSD parts, and a large part of the writing.
Chapter 2
Building JavaScript Applications with Haskell

We introduce the Utrecht Haskell Compiler JavaScript backend, which allows one to compile Haskell code to JavaScript so it can be run in the browser. To interface with JavaScript and overcome part of the conceptual mismatch between the two languages, we introduce the Foreign Expression Language; a small subset of JavaScript for use in Foreign Function Interface imports. Finally we discuss the implementation of a JavaScript application, completely written in Haskell, with which we show that we can write an entire web application without writing JavaScript by hand.

2.1 Introduction

When developing interactive clients for web applications, JavaScript is often the language of choice due to native support in every major web browser. In contrast to other client-side programming languages, no plugins are needed to execute JavaScript. Unfortunately, JavaScript is currently the only client-side programming language that is supported by all major browsers. People wishing to use other programming languages or paradigms have to rely on using existing plugins such as Flash or Java Applets, writing custom browser plugins, or modifying the browsers themselves. None of these options is ideal, since they either require a lot of work, or force the use of strict, imperative programming languages. Instead of choosing between the aforementioned options, we use the Utrecht Haskell Compiler (UHC) [29, 30] to compile Haskell code to JavaScript, effectively using JavaScript as a high-level byte-code, and allowing us to side-step the problems identified with the other approaches.

Since Haskell and JavaScript are based on two completely different programming paradigms, there is a conceptual mismatch between the two languages. To overcome this mismatch, we have extended UHC’s FFI with a small JavaScript-like expression language we call the Foreign Expression Language (FEL). With these enhancements to the FFI, we claim that it is now possible to write complete JavaScript applications using only Haskell. We back up this claim by porting a web-based Prolog “proof assistant” from JavaScript to Haskell. While this chapter focusses on Haskell, the ideas should be relatively easy to implement in similar languages, such as Clean [87].

In this chapter, we make the following contributions:

- We introduce the UHC JavaScript backend, a compiler backend that allows one to compile any Haskell code supported by UHC to JavaScript and execute
it in the browser, while maintaining Haskell’s lazy semantics.

- We introduce the Foreign Expression Language (FEL), which allows for interfacing with object-oriented languages via the FFI.

- We provide evidence that it is now possible to write a web application completely in Haskell.

- We provide a basic library with bindings to common JavaScript APIs.

The rest of this chapter is structured as follows: Section 2.2 introduces the UHC JavaScript runtime system (RTS). Section 2.3 covers the FFI with our additions, after which Section 2.4 shows how we have implemented a fully working JavaScript application completely in Haskell. Sections 2.5 and 2.6 discuss future and related work respectively, after which Section 2.7 concludes.

We assume at least some familiarity with the Haskell Foreign Function Interface (FFI) and JavaScript.

## 2.2 Runtime System

There exists an obvious mismatch between Haskell and Object-Oriented (OO) languages, such as JavaScript, which has been addressed in various ways over time (Section 2.6). One approach to addressing this mismatch is to use the OO language mechanisms as available in JavaScript, in particular prototype based objects, in Haskell. We only mention this topic in passing. Another approach is to use existing JavaScript libraries. We deal with this in the next section by exploiting the freedom offered by Haskell’s FFI. Yet another option is to map the runtime machinery required for Haskell to an imperative language. Such an approach will need to deal with the lazy evaluation strategy imposed by Haskell. We will discuss this approach in the rest of this section.

The design of any backend for a lazy functional language needs to deal with functions, their (lazy) application to arguments, and reducing such applications to Weak Head Normal Form (WHNF). The design should also cater for under- and over saturated function applications as well as tail recursion.

In the UHC’s JavaScript backend, functions and their applications are both represented by objects. Here we omit implementation details and only expose the programmatic interface as used by the runtime system. The actual implementation can be found in the UHC Git repository.

We start with the `Fun` object:

```javascript
Fun.prototype = {
  applyN : function ( args ) ...,
  needsNrArgs : function() ...
}

function Fun( fun ) { ... }
```

[https://github.com/UU-ComputerScience/uhc/](https://github.com/UU-ComputerScience/uhc/)
A Fun object wraps a JavaScript function so that it can be used as a Haskell function. The applyN field is only used when a function application is being evaluated (forced), because only then is it necessary to know the needsNrArgs number of arguments it requires. Otherwise it stays unevaluated as a Fun object wrapped inside an App or AppLT closure object, which will be explained below.

Closures stemming from partially applied (and thus under-saturated) functions need to store already passed arguments and how many arguments are still missing. An AppLT (LT stands for less than) object encodes this. We provide its programmatic interface first:

```javascript
AppLT.prototype = {
   applyN: function (args) ...
   needsNrArgs: function() ...
}
function AppLT(fun, args) {...}
```

An AppLT only wraps other AppLT objects or Fun objects.

Finally, for all remaining saturation cases an App object is used. Knowledge about the degree of saturation is delegated to the encapsulated function object, which may be another App, AppLT, or Fun.

```javascript
App.prototype = {
   applyN: function (args) ...
}
function App(fun, args) {...}
```

With this interface we now can embed Haskell functions. For example, assuming an elementary JavaScript representation of the Haskell function \( \text{id} \), the function \( \lambda x \rightarrow \text{id}(\text{id} x) \) is encoded as follows:

```javascript
new Fun(function(x) {
   return new App(id, [new App(id, [x])]);
})
```

Evaluation is forced by a separate function eval, which assumes the presence of an eOrV (evaluator Or Value) field in all Haskell runtime values. This fields tells us whether the JavaScript object represents a Haskell non-WHNF value which needs further evaluation or not. In the former case it will be a JavaScript function of arity 0 which can be called. A Haskell function or application object does not evaluate itself, since the tail recursion involved will cause the stack of the underlying JavaScript engine to overflow. Instead, we introduce the eval function. This function allows non-WHNF values to be returned, thus implementing a trampoline mechanism:

```javascript
function eval( x ) {
   while ( x && x.eOrV ) {
      if ( typeof x.eOrV == 'function' ) {
         x = x.eOrV();
      }
   }
}
```
<table>
<thead>
<tr>
<th>Haskell</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int, Double, Float</td>
<td>Number</td>
</tr>
<tr>
<td>Integer</td>
<td>BigInt (non-native, offered by a library)</td>
</tr>
<tr>
<td>PackedString</td>
<td>String</td>
</tr>
<tr>
<td>otherwise</td>
<td>RTS representation</td>
</tr>
</tbody>
</table>

Table 2.1: Mapping from Haskell Types to native JavaScript types

```javascript
} else {
    x = x.eOrV;
}
```

Even normal JavaScript values can be passed to `eval`, provided they do not contain an `eOrV` field. The actual `eval` function is somewhat more involved as it provides some protection against null values and also updates the `eOrV` field for all intermediate non-WHNF objects computed in the evaluation loop.

As usual, the evaluation is driven by the need to pattern-match on a value, e.g. as the result of a case expression or by a built-in JavaScript primitive which is strict in the corresponding argument. For example, JavaScript’s actual multiplication function (*) is wrapped in a `Fun` object. Note how its arguments are explicitly evaluated before they are multiplied.

```javascript
new Fun( function(a, b) {
    return eval(a) * eval(b);
} )
```

Depending on the number of arguments provided, either an undersatured closure is built, or the function is directly invoked using JavaScript’s `apply`. In case too many arguments are provided, a JavaScript closure is constructed, which subsequently is evaluated in the evaluation loop of `eval`. The implementation of `AppLT` is similar to that of `Fun`. `App`’s implementation of `applyN` simply delegates to `applyN` of the function it applies to. Also omitted are the encodings of nullary applications, used for unevaluated constants (CAF, Constant Applicative Form) and indirection nodes required for mutual recursive definitions. Data types and tuples are straightforwardly mapped onto JavaScript objects with fields for the constructor tag and its fields. If available, record field names of the corresponding Haskell data type are used. We map `Int`, `Double`, `Float`, and `PackedString` values to JavaScript objects, shown in Table 2.1. Despite the mapping to JavaScript objects, the expressions of these types are lazy. Currently, Haskell arrays are not yet translated to JavaScript arrays.
2.3 JavaScript Foreign Function Interface

We have extended the FFI with the Foreign Expression Language (FEL), a small JavaScript-like language that greatly simplifies interfacing with the JavaScript world from Haskell. The FEL allows one to number and reorder the function arguments, explicitly use them as arguments to JavaScript functions, or use them as objects. Other features include hard coding of literals, accessing array indices, and a built-in mechanism for converting data types to JavaScript objects. The new grammar for importing functions is shown in Figure 2.1. In the current implementation, only string literals are supported, although there are no fundamental issues preventing implementation of numeric, boolean, undefined and null literals.

```
| exp ::= '{' | (arg | ident) post' | exp with FEL expression
| post ::= '.' ident | [ ] exp | ( ) args' | post with FEL expression
| args ::= e | arg ( , arg)*' | args with possible arguments
| arg ::= % ('*' | int) | literal text
| ident ::= a valid JavaScript identifier
| int ::= any integer
| str ::= any string
```

Figure 2.1: Import entity notation for the JavaScript calling convention

Common FFI features, such as the dynamic and wrapper imports, work as expected, allowing one to use higher-order JavaScript functions in the same way as C function pointers.

As an example of how to use the FEL to import a JavaScript function, suppose we want to import the subString method from the JavaScript String class, where myStr is a concrete JavaScript string object:

```
myStr.subString(start, length);
```

This method is called on a JavaScript string object, and returns a substring, based on the integer value for a start offset and an integer value for the length of the substring, both of which are passed as arguments to the method. Importing this method shows the FEL’s added value in several ways: the method is called on a JavaScript object, it takes multiple arguments, and it requires conversion from a Haskell String type to a native JavaScript string type. The latter takes a bit of work, because Naively using a Haskell String would give us a JavaScript representation of a list of characters, rather than a JavaScript string. To obtain
a native JavaScript string, we require the Haskell \texttt{String} to be converted to a \texttt{JSString}, which is a type synonym for \texttt{PackedString}. An example of importing the \texttt{subString} method is shown below:

\begin{verbatim}
foreign import js "%1.subString(%2, %3)"
subString :: JSString -> Int -> Int -> JSString
\end{verbatim}

In addition to the \texttt{js} calling convention, the other noticeable difference with, for example, a C import, is the import definition in the string. Rather than having the FFI place all arguments in one position, we number the arguments and allow them to be placed in different positions in the imported method. Manually ordering arguments enables us to treat one of the arguments as an object, while treating the rest of the arguments as parameters to a method call on that object. In our example, the first argument, indicated by \texttt{%1}, before the dot, is treated as an object in the generated JavaScript code. The number of the argument corresponds to the position of the arguments in the type signature. The two remaining arguments are placed between parentheses, so that they become arguments in the method call in the generated JavaScript code.

An alternative way of writing this import is shown below, where we replace the last two explicit argument positions with a wildcard. This says that all remaining arguments should be placed where the wildcard is, saving the programmer some work. Using a wildcard has as added advantage that it becomes easy to import variadic JavaScript methods; the function’s arity is then only determined by the type signature, without the need to modify the foreign expression.

\begin{verbatim}
foreign import js "%1.subString(%*)"
subString :: JSString -> Int -> Int -> JSString
\end{verbatim}

Exporting a function does not make use of the FEL, so it is not much different from exporting a function for the C FFI. The only concerns to keep in mind are using the \texttt{js} calling convention, and specifying a JavaScript-compatible type in the type signature.

\subsection*{2.3.1 The UHC-JavaScript library}

We provide the UHC-JavaScript library\footnote{https://github.com/UU-ComputerScience/uhc-js.} to streamline the development of JavaScript applications with UHC. It contains bindings to standard ECMAScript \footnote{https://jquery.org/}, the formal standard behind JavaScript, as well as bindings to the jQuery library.\footnote{https://jquery.org/} The library aims to provide a bare-metal interface that is consistent with the JavaScript functions. Eventually, this library should form a core upon which more (functional) abstractions are built. We shall make use of this library in the rest of this chapter.
2.3.2 Creating, manipulating and querying objects

Being a purely functional programming language, Haskell has no notion of objects. JavaScript, however, does. Objects come in two flavours: anonymous and named objects. The former is denoted in JavaScript as {}, while the latter is created by defining a constructor function. Objects can then be instantiated with the new keyword, e.g. new MyObj(). Each constructor function also has a prototype object. New object instances will automatically have the same values and functions as the prototype.

UHC offers support for creating, manipulating and querying objects, using several new primitive functions in the runtime-system (RTS). Instead of showing the rather uninteresting function definitions in JavaScript, the code below shows the Haskell type signatures which need to be used when importing these primitives with the FFI:

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>primMkCtor</td>
<td>JSString -&gt; IO ()</td>
</tr>
<tr>
<td>primMkObj</td>
<td>JSString -&gt; IO (JSPtr c)</td>
</tr>
<tr>
<td>primMkAnonObj</td>
<td>IO (JSPtr c)</td>
</tr>
<tr>
<td>primGetAttr</td>
<td>JSString -&gt; JSPtr c a</td>
</tr>
<tr>
<td>primSetAttr</td>
<td>JSString -&gt; a -&gt; JSPtr c I0 (JSPtr c)</td>
</tr>
<tr>
<td>primModAttr</td>
<td>JSString -&gt; (a -&gt; b) JSPtr c -&gt; I0 (JSPtr c)</td>
</tr>
<tr>
<td>primGetProtoAttr</td>
<td>JSString -&gt; JSString -&gt; I0 a</td>
</tr>
<tr>
<td>primSetProtoAttr</td>
<td>JSString -&gt; a -&gt; JSString -&gt; I0 ()</td>
</tr>
<tr>
<td>primModProtoAttr</td>
<td>JSString -&gt; (a -&gt; b) JSString -&gt; I0 ()</td>
</tr>
</tbody>
</table>

JSString is a type synonym for PackedString, the builtin type corresponding to JavaScript strings. The primMkCtor function creates a new constructor function if it does not yet exist in the window scope, where window is the variable containing everything pertaining the current window or tab. This function is usually only called from within the other functions listed above. The primMkAnonObj function creates an anonymous object {}, while primMkObj accepts a string with the class name of the object to be created. If the class does not exist yet, it is created using an empty constructor. The other functions manipulate objects and prototypes, using a mechanism inspired by lenses [55, 64], an abstraction over accessors and mutators. The first argument is always the name of the object attribute of interest passed as a string. In case of the set-functions, the second argument is the value that needs to be set. Since JavaScript is a loosely typed language, this can be any type, even when interfacing with it from the Haskell world. The mod-functions take as second argument a function which modifies the attribute specified in the first argument. Modifying an attribute may change its type, hence the a -> b type for the function. Finally, the last argument is either a reference to an object, or the name of a class as a string, in case of prototypes. These functions can be used by importing them as primitives:

```
foreign import prim "primGetAttr"
_getAttr :: JSString -> JSPtr p -> IO a
```

Objects are represented by a JSPtr type. It has no constructors, so they cannot be instantiated directly. The only way an object can be obtained is by getting it
via the FFI. A JSPtr takes one phantom type as a parameter, which specifies the type of the JavaScript object. This should again be a type without constructor. Suppose we want a pointer to a JavaScript Book object, for which we have some definition in JavaScript. We define it in Haskell as follows:

```haskell
data BookPtr
type Book = JSPtr BookPtr
```

We can now define functions on the Book type, giving us a type-safe way to deal with JavaScript objects. A similar approach is often taken in GHC’s C FFI to deal with pointer types.

We offer the Language.UHC.JS.Primitives module in the UHC-JavaScript library, which defines primitive imports and abstracts away from JSString. Using these functions we can now create, manipulate and query an object:

```haskell
main = do
  o <- mkObj "Book"
  setAttr "pages" 123 o
  modAttr "pages" (+1) o
  p <- getAttr "pages" o
  print p -- Prints 124
```

While defining objects as shown in the previous example works fine, the process is rather verbose and tedious, especially when dealing with several object attributes. It would therefore be ideal if we could use Haskell data types to achieve the same results. In some ways, data types and JavaScript objects have a lot in common, especially when the data type has record selectors. Suppose we have a simple Book type in Haskell:

```haskell
data Book = Book
  { author :: JSString,
    title :: JSString,
    pages :: Int }
```

A concrete Book value would look as follows:

```haskell
myBook = Book
  { author = toJS "me",
    title = toJS "story",
    pages = 123 }
```

The representation of myBook closely resembles an object with the same data in JavaScript:

```javascript
myBook = { author : "me",
           title : "story",
           pages : 123 }
```

We use JSString here so that the resulting Haskell record relates more closely to the JavaScript object.
In fact, a JavaScript object very similar to the one shown above is already being generated by the UHC. However, since it is generated as an application of a constructor to some values, the generated data type values are not directly usable in other JavaScript libraries. We require a mechanism to convert the Haskell representation of the data type into a JavaScript representation. This idea is similar to that of the FFI’s wrapper import feature. Using a similar mechanism to the wrapper, we can make Haskell data types available as JavaScript objects. This mechanism is exposed via the FEL simply as `{}`:

```haskell
foreign import js "{}"
mkObj :: a -> IO (JSPtr b)
```

It takes a value of data type `a` and converts it to a plain JavaScript object, resulting in a pointer to the new object. If the data type contains record selectors, they will be used as the object’s indices. When no record selectors are available, an integer is used instead.

Creating the object is achieved by recursively evaluating and cloning the data inside the data type to a new, empty object, disposing of RTS-specific information in the process. Cloning is required, because modifications on the new object by plain JavaScript code must not be reflected in the original data type value. Using the object wrapper, we can simplify our example above:

```haskell
main = do
  let b' = myBook { pages = pages myBook + 1 }
b <- mkObj b'
p <- getAttr "pages" b
print p -- Prints 124
```

Note that even though this example is only one line shorter, we also have the two strings available in our JavaScript object, which would have taken two more lines in the original example. More importantly, Haskell’s type system is in a much better position to catch programmer mistakes, since record selectors are used in the modification of the `pages` value instead of strings.

### 2.3.3 Pure objects

Objects in JavaScript are mutable by nature. By modifying an object, you modify it for everything that has a pointer to that particular object. This forces any update operation to be defined in `IO`. In order to escape the `IO` monad, update operations need to become non-destructive, which is achieved by creating a copy of an object before modifying it. The RTS exports a primitive to do exactly this:

```haskell
primClone :: JSPtr a -> JSPtr a
```

By cloning an object first, all pointers to the original object remain untouched when modifying the clone. This enables pure variants of the `primSetAttr` and `primModAttr` functions:

```haskell
primPureSetAttr :: JSString -> a -> JSPtr c -> JSPtr c
primPureModAttr :: JSString -> (a -> b) -> JSPtr c -> JSPtr c
```
CHAPTER 2. BUILDING JAVASCRIPT APPLICATIONS WITH HASKELL

Since a potentially large graphs of objects will be cloned by these pure functions, they should be used with care. The cloning method used is a modification of the cloning method used by jQuery.

2.4 The JCU Application

To explore the limitations, and to demonstrate the features of the UHC JavaScript backend in a real-life scenario, we ported the ‘JCU Prolog Proof Assistant’ [101], a web application developed to aid in teaching [100] Prolog at the Junior College Utrecht, to Haskell. It is a tool developed for students to learn about important concepts in computer science, such as proofs, trees, unification, and backtracking, by means of proving Prolog queries manually. Students enter a Prolog query, after which they can build a proof of this query by dragging and dropping Prolog rules and facts on top of the query, and by applying substitutions manually throughout the proof tree.

The application was originally programmed in CoffeeScript [5], a layer of syntactic sugar for JavaScript, and used the Brunch [6] framework. In the original implementation, all Prolog logic was implemented server-side in Haskell, using the NanoProlog [7] library. We rewrote the application in Haskell using UHC and the UHC-JavaScript library. We import jQuery via the FFI for interacting with the Document Object Model (DOM). We use sequential non-blocking communication with the server. The resulting application has the same functionality and stability as the original implementation. As is expected of applications that interact heavily with a graphical user interface, a large part of the application’s code lives in the IO monad.

With the ability to compile Haskell to JavaScript comes the possibility of running any Haskell library that compiles on UHC in the browser, without modification. We use this feature in the JCU web application to run the NanoProlog library in the browser, allowing us to perform proof checking and unification at the client-side, eliminating the need for many AJAX requests. In a further step we eliminated the need for a server altogether by storing the set of rules and facts using HTML5 Local Storage, a browser-based database supported by most modern browsers, instead of in a database on the server. With this modification, the assistant can be run with only the requirement of a modern web browser; no Internet connection is required. A live demo is available online [8].

2.4.1 Implementation Issues

Most of the problems we encountered in porting the JCU application to Haskell were due to the lack of advanced language features in UHC, such as functional dependencies and type families. Practically, this implies that only part of the

---

libraries available on Hackage today can currently be compiled to JavaScript using the UHC JavaScript backend.

Another issue arises from JavaScript’s scoping rules. In JavaScript, the keyword `this` is dynamically scoped, while all other variables are lexically scoped. Since we emulate lazy evaluation by native JavaScript functions encapsulated by objects, the `this` keyword can in some cases point to the runtime system rather than the expected scope, exposing the runtime system to the programmer. Hence, simply importing `this` as a function using the FFI is not an option. Still, we might need access to `this`, because an imported JavaScript library expects the programmer to make use of the keyword in a callback function. The jQuery library, for example, expects event callbacks to get the active DOM-node using the `this` keyword. One way to still get a reference to the expected object when using `this` is to create a wrapper function that captures the expected scope and passes it to the wrapped function as explicit argument. We have implemented this solution in the `wrappedThis` function, which is part of our RTS.

Figure 2.2 shows how the `wrappedThis` function can be used to obtain the value of an HTML input field. The code above the definition of `bindInput` is copied from the JavaScript library. `valString` is a function that gets the value of a jQuery object as a `String`. We query the DOM using jQuery, retrieving all `input` elements, such as text fields, in the DOM. We define a function `alertHndlr` that takes the string value of a jQuery object and then shows it in an alert box. Note the explicit `this` parameter. We then wrap it so it becomes a JavaScript function, after which we partially apply it to an explicit `this` parameter using `wrappedThis`. Finally, we bind the event handler to all input fields retrieved by our jQuery selector.

A last example of implementation difficulties is found in the lack of threading support in our current implementation of the proof assistant, and in the current implementation of the UHC JavaScript backend. In addition to the web-based proof exerciser, we offer a web-based user interface to NanoProlog’s interpreter. In some cases, the interpreter can get stuck in an infinite recursion when trying to unify a rule. For example, trying to prove the query `silly(X), silly(X)` will never terminate. Originally, we spawned a new thread on the server, which we would terminate after a given amount of time. Our current approach, however, does not yet offer threading, risking blocking the client-side process causing a tab or the whole browser to hang. JavaScript’s WebWorkers might provide a solution to this problem, although we have yet to investigate this option. Another solution would be to change the implementation to limit its recursion depth.

### 2.4.2 Performance

In general, the performance of the web application is on par with the original implementation in JavaScript, but only when using a state of the art JavaScript engine, as is found in Google Chrome or Safari. The largest bottleneck seems to be memory management. Building up lazy Haskell expressions leads to a large number of JavaScript objects. The quick creation and then successive destruction
CHAPTER 2. BUILDING JAVASCRIPT APPLICATIONS WITH HASKELL

```haskell
data JQueryPtr
  where JQuery = JSPtr JQueryPtr

type ThisEventHandler = JQuery -> JQuery -> JEventResult
type JEventHandler = JSFunPtr (JQuery -> JEventResult)
type JThisEventHandler = JSFunPtr ThisEventHandler

foreign import js "%1.bind(%*)"
    bind :: JQuery -> JSString -> JEventHandler -> IO ()

foreign import js "wrappedThis(%1)"
    wrappedThis :: JThisEventHandler -> IO JEventHandler

valString :: JQuery -> IO String
mkJThisEventHandler :: ThisEventHandler -> IO JThisEventHandler

bindInput = do
    let alertHndlr :: ThisEventHandler
        alertHndlr this _ = valString this >>= alert
    inputField <- jQuery "input"
    eh <- mkJThisEventHandler alertHndlr >>= wrappedThis
    bind inputField (toJS "blur") eh
```

Figure 2.2: Code for adding an event handler to an input field

of these large expressions places a strain on the memory manager and garbage collector. Other popular browsers, such as Firefox, Opera, and Internet Explorer, perform significantly worse than the aforementioned browsers, although this has only been tested informally.

2.5 Future Work

While we have shown that it already is possible to implement an entire JavaScript application in Haskell, there is still a lot of room for improvement. As mentioned before, UHC itself lacks support for the more advanced Haskell features, such as type families and functional dependencies. This prevents us from compiling many packages from Hackage directly to JavaScript. In order to make the backend more widely usable, we would need to implement these language features in the UHC.

Our current UHC-JavaScript library relies on the programmer to use imported functions correctly. The object-wrapper import, for example, will try to wrap anything, possibly failing at runtime. Extra constraints could be added, although the RTS cannot currently deal with them. Eventually, one could imagine a higher-level library being built on top of the low-level imports to provide improved type-safety. Such libraries may be based on generic programming to eliminate repetition, functional reactive programming [42, 104] to interact with the DOM, or they may be an entire user-interface toolkit, such as wxHaskell [66].
Working with WebWorkers as a JavaScript counterpart to Haskell threads is not investigated yet. Our JCU application would become significantly more usable with a threading alternative.

Communication with the server is currently encoded manually. One could imagine an approach inspired by Cloud Haskell’s typed channels, where communication proceeds over type-safe communication channels, abstracting away from the actual AJAX call.

Currently the only way of converting a data type to a JavaScript object is to do so at runtime. This, however, is a process with time complexity linear in the number of data type records. Future work could focus on generating (parts of) JavaScript objects at compile-time, so that only dynamic values will need to be copied to the object at runtime.

Cross-compiling Haskell to a different platform means that some assumptions following from using a single platform only are no longer valid. First, a different platform means a different runtime environment. For example, almost all of the UNIX functionality is available for the usual Haskell UNIX runtime, but is naturally not available inside a web browser. Vice versa, specific JavaScript libraries like jQuery are not available on a UNIX platform. Some library modules of a package (partially) cannot be built on some platforms, while others (partially) can. To cater for this, UHC rather ad-hoc marks modules to be unavailable for a backend by a pragma {-# EXCLUDE_IF_TARGET js #-}. Of course CPP can still be used to select functionality inside a module. However, in general, awareness of platform permeates all aspects of a language system, from the compiler itself to the library build system like Cabal. In particular, Cabal needs a specification mechanism for such variation in target and platform to allow for selective compilation of a collection of variants. Currently this means that UHC compilation for the JavaScript backend cannot be done through Cabal.

Currently, we generate JavaScript from the compiler’s core language. It might be possible to generate faster code which uses native JavaScript language features when generating JavaScript at a later stage in the compiler pipeline, where the intermediate code is more imperative in nature.

2.6 Related work

The idea of running Haskell in a browser is not new. To our knowledge, the first attempts to do so using JavaScript were made in the context of the York Haskell Compiler (YHC)\(^9\) The DOM inside a browser was accessed via wrapper code generated from HTML standard definitions\(^10\). However, YHC is no longer maintained, and direct interfacing to the DOM nowadays is replaced by libraries built on top of the multiple DOM variations.

GHCJS \(^72\) \(^70\) is an attempt to use the GHC API to create a dedicated Haskell to JavaScript compiler. It uses the C calling convention, rather than a dedicated js calling convention. A major advantage of using the GHC API

\(^9\) https://wiki.haskell.org/Yhc/JavaScript
\(^10\) https://wiki.haskell.org/Haskell_in_web_browser
is that a mature, production-ready compiler, with support for advanced type-
system features is at the programmer’s disposal, solving some of the issues we are
currently experiencing due to lack of these features in UHC. Currently, GHCJS
does not support an import system like the one described in this chapter, so
its ability to use external APIs is limited. GHCJS’ authors remarked on the

**glasgow-haskell-users** mailing list (13 November 2012) that adding an FEL-
like import mechanism to GHCJS should be relatively straight-
forward.

A recent and promising-looking attempt at compiling Haskell to JavaScript is
the Fay language\(^\text{11}\) by Chris Done, which aims to support a subset of Haskell and
compile to JavaScript. It, too, makes extensive use of GHC, giving it a production-
ready Haskell compiler and type-checker to build on. In designing Fay’s FFI, Done
drew some inspiration from the work we present here, namely the FEL.

We ran a benchmark between UHCJS, GHCJS, Fay and Native JavaScript
and noticed that the code generated by UHCJS performs the worst by far. This
is largely due to excessive memory allocation of objects and subsequent garbage
collection. The full details of this benchmark can be found in our git repository\(^\text{12}\).

Another recent attempt is Haste \(^\text{37}\) by Anton Ekblad. It, too, builds on top
of GHC, and it attempts to be easy to use and generate “relatively lean code”. It
comes with a small reactive library for interacting with the DOM.

Rather than focusing on source-to-source compiling, JavaScript libraries like
“Functional JavaScript”\(^\text{13}\) offer APIs for a more functional style of programming
in JavaScript. “Haskell in JavaScript”\(^\text{14}\) offers an interpreter for Haskell, written
in JavaScript.

The workflow framework iTasks \(^\text{34}\), written in the purely functional program-
ing language Clean, uses a minimalist platform-independent functional language
called SAPL, which is interpreted in the browser by code written in Java. The
latest interpreter incarnations are written in JavaScript \(^\text{61, 18, 85}\). Although
currently a Haskell front-end exists for Clean, the use of it in a browser appears
to be limited to iTasks. The intermediate language SAPL also does not provide
any facilities similar to our Haskell FFI.

### 2.7 Conclusion

We have shown that UHC is capable of supporting the development of complete
client-side web applications, opening the door to Haskell-only web development.
In the process we added the FEL to UHC and provided a library that exposes
the JavaScript world to Haskell. Considering the increasing maturity of the GHC-
based solutions, we can conclude that the two biggest contributions of this chapter
are the FEL, and our evidence that writing a complete, non-trivial web application,
on optionally using external JavaScript libraries is now possible in Haskell. Since UHC
does not support advanced Haskell language features, and GHC’s development is

---

1. [https://github.com/faylang/fay/wiki](https://github.com/faylang/fay/wiki)
4. [https://github.com/johang88/haskellinjavascript](https://github.com/johang88/haskellinjavascript)
faster and more consistent, it remains to be seen whether our implementation in UHC can grow to become a mature tool for developing JavaScript applications. While still keeping this option open, we also call on authors of GHC-based solutions to consider using the contributions of this chapter in their work.

When it comes to libraries for writing JavaScript applications in Haskell, better abstractions are still required to reduce the amount of code that lives in the \texttt{IO} monad directly, and to give programming with the UHC JavaScript backend a more functional feel. While performance, in most cases, is acceptable, it needs to be improved if computationally heavy functions are to be run on the client. In order for most of the frequently used Hackage libraries to be run on the client, additional work on UHC and Cabal will have to be performed.
Chapter 3
Tonic: A Graphical Representation of Tasks

In Task Oriented Programming (TOP), tasks, as performed by both humans and computers, are the core concept. TOP is implemented by the iTask system (iTasks) as a shallowly embedded Domain Specific Language (DSL) in the functional programming language Clean. iTasks is used in industry for rapid prototyping in complex sociotechnical domains. However, for non-technical stakeholders, an iTask specification is too difficult to understand. Stakeholders like to communicate their ideas informally, using drawings and natural language, while TOP programmers model tasks in Clean. We propose a way to eliminate this communication gap by translating a textual iTasks specification into a graphical one, called a blueprint, which should be understandable by non-technical stakeholders. Blueprints abstract from Clean language details as much as possible, yet contain enough information to be understandable independently. Furthermore, we show how blueprints are instantiated at runtime, resulting in an animated trace, showing how end-users progress with which tasks. The Clean compiler has been adjusted to generate blueprints, as well as inject wrapper code that relates run-time information to the compile time specification. A Tonic viewer application, written in iTasks, uses this wrapper code to visualize the traces.

3.1 Introduction

Task Oriented Programming (TOP) [86] is a new style of functional programming intended for developing reactive web-based multi-user applications. It is implemented by the iTask system [84] (iTasks) as a Domain Specific Language, shallowly embedded in the strongly typed, lazy, purely functional programming language Clean [87]. TOP allows a programmer to focus on the high level specification of the tasks that need to be done. One does not need to worry about the implementations details commonly faced when writing web applications. For example, GUI generation and handling, data storage and retrieval, persistency and the communication between the participating parties are all taken care of automatically. This is achieved by using advanced functional programming techniques such as type-driven generic programming [62, 8], uniqueness typing [15], and a hybrid static/dynamic type system, which allows transferring and storing function applications [82]. With iTasks, developing reactive web-applications for supporting collaboration on the Internet becomes equivalent to task modelling, giving all project stakeholders a common notion: the tasks that have to be performed by human beings and their computers.
CHAPTER 3. TONIC: A GRAPHICAL REPRESENTATION OF TASKS

One of the strengths of the system is its suitability for rapid prototyping. By defining different task models, one can study alternative, more efficient ways to let people and systems collaborate in complex and exacting settings. iTasks is being used in industry for this purpose. For example, it is used to prototype software for Crisis Management and Command and Control centres (see [69]).

Finding the best way of working together on tasks requires close collaboration between all stakeholders, such as managers, domain experts, and programmers. Multi-disciplinary collaboration is a well-known and hard challenge, due to the differences in expertise and knowledge levels. But, since TOP applications can be defined at a high level of abstraction in terms of the common notion of tasks, it might become possible to bridge this classical communication gap. Doing so is not easy, however. A domain expert probably wants to define tasks informally, e.g., using a combination of natural language and diagrams of boxes connected by arrows (as used in BPMN [76, 77], for example). An iTasks programmer defines tasks textually in a precise, formal, mathematics-based notation (i.e. Clean). Most domain experts do not possess sufficient technical skills to define these tasks formally in Clean or to understand the code that programmers have written.

In this chapter we present Tonic: “Task-Oriented Notation Inferred from Code”. We aim to bridge the communication gap by offering a common graphical language that both programmers and non-programmers can understand. The idea is twofold: on the one hand we generate a diagram of a program’s tasks, called a blueprint, from the formal iTasks specification. On the other hand, the same graphical notation can be used by non-programmers to convey their ideas using pen and paper.

A blueprint must strike a balance between showing the task structure at a sufficient level of detail on the one hand, without overwhelming non-programmers with details of programming in Clean on the other hand. We realize that it will take several design iterations of Tonic’s graphical syntax before the ideal balance is found. In this chapter we focus on the technical challenges that have to be solved to make Tonic possible.

A modified Clean compiler is used to generate blueprints. Blueprints will not only give insight in the tasks that have been defined statically, it will also be used to explain what goes on when tasks are being executed.

There are many challenges that have to be addressed. Firstly, tasks are functions. Representing functions graphically and predicting the exact order of evaluation in a lazy context are known to be hard problems in functional programming. However, tasks are functions that have result-type (Task a) and can therefore easily be distinguished from other functions. In addition, the task combinators have known and predictable monadic operational behaviour, making it possible to show how tasks depend on each other.

Secondly, the host language Clean, much like Haskell, offers many language constructs. Since iTasks is a shallowly embedded DSL, any host language construct can be used to define tasks. Because blueprints must not overwhelm non-programmers with the details of programming in Clean, we have to carefully choose

1Ideally we would use a compiler API instead, but the Clean compiler’s architecture currently does not allow for this
which host language constructs can appear in a blueprint and which not. We ex-
pect that when task engineers use frequently occurring function patterns in a
consistent way, supported by a coding discipline or tool, it is possible to generate
independently understandable blueprints, even for complex applications.

Thirdly, since iTasks is being used to model complex task-collaborations, the
dynamic behaviour of the tasks needs to be validated by all stakeholders: What
happens when the tasks are executed? Which tasks are executed? Who is doing
what? What is the progress? Is the described way of working the best way to
achieve the goals? What happens if unexpected tasks need to be done? To answer
these questions we need to dynamically relate the statically generated blueprints
to the run-time behaviour of their corresponding task function applications during
the execution of the iTasks application. This requires a solution akin to debuggers
and tracers, which are yet another well-known and hard challenge in functional
programming. Fortunately, the iTasks run-time system keeps track of all tasks
under evaluation, their current state, their workers, and so on. In order to re-
late this run-time information to the static blueprints, we let the modified Clean
compiler also transform the task definitions and applications to pass compile time
information to the iTasks run-time. This allows us to show how the blueprints are
instantiated using the Tonic viewer, which is also written in iTasks.

In this chapter we focus on the technical challenges and make the following
contributions:

• We define at what level of abstraction we generate blueprints, striking a
balance between the level of detail and understandability by domain experts;

• We define the blueprints by means of formally defined production rules;

• We discuss implementation issues in implementing blueprint generation;

• We show how we technically manage to instantiate blueprints at run-time
such that we can show which tasks in the blueprint are finished, are currently
being executed, and what their actual parameters and current results are.

The remainder of this chapter is structured as follows. In Section 3.2 we give
a short overview of the iTasks DSL and present a running example. In Section 3.3
we show how and which static blueprints are generated. Section 3.4 explains how
we manage to show the instantiation of blueprints at run-time. Section 3.5 reviews
related work. Conclusions are presented in Section 3.6.

3.2 Short Overview of iTasks

The TOP paradigm, as embodied in iTasks, builds on a few core concepts: tasks,
which define the work that needs to be done; editors, which are tasks that facilitate
user interaction; combinators, to compose tasks from simpler ones; and shared data
sources (SDSs), to handle shared information in a uniform way. Tasks are reactive
and their current state can be observed. A task of type (Task a) processes a task
value of type a, which may change over time while the work takes place. The value
is either \textit{absent}: no value is available (yet), \textit{unstable}: some value is available but it might change in the future or even become absent, or \textit{stable}: the value is final. We illustrate these concepts by means of a small case study that models part of the operation of an emergency call center.²

\begin{verbatim}
:: Emergency = { time :: DateTime, info :: CallInfo }
:: CallInfo = { contact :: String, phone :: String, location :: Address,
  situation :: String, authorities :: [Authority] }
:: Address = { city :: Maybe String, street :: Maybe String, no :: Maybe Int }
:: Authority = Ambulance | FireBrigade | Police
:: Verdict = Success | Fail String | FakeCall

derive class iTask Emergency, CallInfo, Address, Authority, Verdict

requiresAuthorities :: Emergency -> Bool
requiresAuthorities call = not (isEmpty call.info.authorities)

makeEmergency :: DateTime CallInfo -> Emergency
makeEmergency now data = {time=now, info=data}

emergencies :: Shared [Emergency]
emergencies = sharedStore "emergencies" []

main :: Task Verdict
main = handleEmergencyCall processPhoneCall

processPhoneCall :: Task Emergency
processPhoneCall = get currentDateTime >>= logCall

logCall :: DateTime -> Task Emergency
logCall now = makeEmergency now <$> enterInformation "Enter call information:" []

handleEmergencyCall :: (Task Emergency) -> Task Verdict
handleEmergencyCall intake = UserWithRole "call-intaker" @: (intake >>* [OnAction ActionContinue (ifValue requiresAuthorities coordinate), OnAction (Action "Fake call" []) (always (return FakeCall))])

coordinate :: Emergency -> Task Verdict
coordinate call
  = upd (add call) emergencies
  >>| UserWithRole "call-coordinator" @: (allTasks (alertAuthoritiesAbout call) >>= showSuccessOfVerdicts)
  where
    add call calls = [call : calls]

alertAuthoritiesAbout :: Emergency -> [Task Verdict]
alertAuthoritiesAbout call=: {info={authorities}} = map (alert call) authorities
  where
    alert :: Emergency Authority -> Task Verdict

²Consult \cite{no} for a concise overview of the syntactic differences between Clean and Haskell.
\end{verbatim}
3.2. SHORT OVERVIEW OF iTASKS

```haskell
alert {info={situation}} authority = enterInformation info []
where
  info = join "\n" [\"Alert\", toSingleLineText authority, situation]
showSuccessOfVerdicts :: [Verdict] -> Task Verdict
showSuccessOfVerdicts verdicts
  = case collectFailures of
    [] -> return Success
    fails -> let msg = join "\n" fails
              in viewInformation \"Uninformed authorities:\" [] msg
                     >>= return (Fail msg)
where
  collectFailures = [str \n Fail str <- verdicts]
```

In iTasks, the entities of the problem domain are typically modeled with common data types (record types, algebraic data types, synonym types, basic types). Pure functions are used to define relationships between these entities.

In the emergency call center example, emergency calls (of type Emergency) are received at some date and time (DateTime is a predefined type). During the intake of a call, an employee finds out and records the information (of type CallInfo) that is provided by the caller (lines 1-7). In addition, it must be determined which authorities need to be notified about the emergency. A call-intaker has been trained to determine if a call is fake. This results in a Verdict how the emergency call has been dealt with. The predicate requiresAuthorities checks whether an emergency calls for help by at least one authority. The function makeEmergency creates an Emergency value.

On line 8, instances of the iTask class for the indicated types are derived. This class consists of the predefined type driven generic functions that are used by the iTasks run-time system to handle GUI rendering, (de)serialization, persistent storage, and comparison of values, amongst others. Automatic deriving allows task engineers to concentrate on specifying the intended behaviour correctly when communicating with domain experts. The remainder of the specification defines the tasks that need to be performed when receiving an emergency call.

The main task handles an emergency call. To emphasize the fact that the way in which the call information is received is less relevant, the main task is implemented as the higher-order task function handleEmergencyCall, which is parameterized with a task function that abstracts from the exact way in which the call is received. In this case, processPhoneCall models how phone calls are processed. In processPhoneCall, currentDateTime is an SDS that holds the current date and time. The task function get obtains the current value of the SDS. This value is passed to the logCall task using the monadic >>= combinator. The logCall task is interactive: enterInformation is an editor that creates a user interface with which, in this case, only values of type CallInfo can be created (see Figure 3.1 (left)). Editors never reach a stable task value because the user can, at any time, decide to continue working on the current task value. We map makeEmergency over the editor to turn the CallInfo task value into an Emergency structure using the earlier retrieved date and time value. As a result, the context
in which the `processPhoneCall` task is executed, `handleEmergencyCall`, observes the `Emergency` value, if any, to decide how to proceed.

The `handleEmergencyCall` task assigns (\@:) the task of handling an emergency call to a `call-intaker`, who executes the `intake` task prescription and decides whether the call must be acted upon or is a fake.

The core combinator that observes a task and specifies the follow-up tasks is the `step` combinator `>>*`. Follow-up tasks can either be handled fully autonomously by iTasks, or require triggering by the current worker before they are handled autonomously. The autonomous part is a computation that, depending on the observed task value, potentially returns a follow-up task `Maybe(Task b)`. The example uses two predefined patterns: `ifValue c t` tests only (un)stable task values with `c` and if the condition holds, proceeds as `t`; `always t` always returns `t` regardless of the (availability of the) task value. Worker-triggers are specified as `(OnAction a f)` values that provide the worker with a user-interface based on `a` (usually a button), that initiate evaluation of the autonomous computation `f` to compute the follow-up task. In Figure 3.1 (left), the two actions are rendered as buttons at the lower-right bottom of the screen. Follow-up task specifications that consist only of an autonomous part `f` are specified as `(OnValue f)`. It should be noted that the monadic `>>=` combinator is implemented in terms of `>>*`. It proceeds to its right-hand side task in case the left-hand side task has a stable value, or when the user clicks its “Continue” button.

The first job of the `coordinate` task is to log the call in the `emergencies` SDS. Next, a worker with role `call-coordinator` contacts all requested authorities. This is an example of a `parallel` task composition. The core combinator in these situations is `parallel`. The expressive power of this general purpose combinator is not always used (it can handle a dynamic number of potentially distributed tasks), so frequently occurring parallel work patterns are offered by iTasks (e.g.
3.3. STATIC TONIC BLUEPRINTS

In this section we describe how (Section 3.3.1) and which (Section 3.3.2) static Tonic blueprints are generated from iTask specifications. This section concludes with a brief discussion about the chosen selection of Tonic blueprints (Section 3.3.4).

3.3.1 Generating Tonic blueprints

The example in Section 3.2 illustrates that the iTask DSL is shallowly embedded in the host language Clean. Clean is a full-fledged, modular, strongly typed, lazy functional programming language supported by an industrial strength compiler and IDE. In general, a task specification consists of several modules and depends on the iTask SDK, as well as the Clean SDK (prelude and many other modules).

The goal of Tonic is to generate blueprints of those parts of the task specification that are deemed meaningful for stakeholders. This amounts to implementing many requirements that are already implemented by the Clean compiler: project management in order to identify and locate all required source files; parsing in order to recognize proper iTask specifications; and typing to accept only statically correct specifications, distinguish task functions from other functions, and identify task expressions. Instead of reimplementing these features, we extend the Clean compiler with an (optional) additional Tonic pass (see Figure 3.2).

The Tonic pass must take place after the typing pass because type information
is required in deciding what to render in the blueprints. Figure 3.2 shows that this has drawbacks as well. Like most advanced compilers, the Clean compiler transforms the source code to a core language as soon as possible. For instance, lambdas are lifted, list comprehensions are desugared, function patterns and guards are transformed to cases, macros are expanded, and where-clauses are desugared to lets. The iTask specification enjoys no special status with respect to the iTask SDK and Clean SDK, so the Tonic pass must figure out which parts of which modules need to be rendered. Although the Tonic pass renders blueprints of the tasks that are transformed to core Clean, this affects the task structure much less than the host language specifics.

We assume that types being used need no special explanation and that the types displayed in blueprints make sense to the domain expert as well. The predefined generic functions allow to show values of these types, as we have seen in Figure 3.1. This technique is used to show task values and results appearing in blueprints with the Tonic viewer as explained in Section 3.4.

### 3.3.2 Tonic blueprints

Tonic must strike a balance between showing the task structure on a sufficient level of detail on the one hand without overwhelming non-programmers with all kinds of host language constructs on the other hand. It is crucial that our target audience of non-programmers understands the generated blueprints. Therefore, we have informally verified the blueprint’s understandability with this target audience. We expect that the blueprints improve over time when more user applications have been experimented with. In the remainder of this section we present the blueprints generated for the example program of Section 3.2. We have formalised the generation of blueprints in a set of rules, which are presented below in Section 3.3.3. Specification guidelines for the task engineer emerge in a natural way. These are

---

**Figure 3.2:** The Tonic phase in the Clean compiler.
presented in this section as well.

Our blueprint examples start with the blueprint of the **main** task.

```
main yields an Emergency

handleEmergencyCall

processPhoneCall
```

Given a task specification, Tonic renders all top-level task functions. Task functions are always displayed in a rounded box with two or three compartments (rules 3.3.3.1-2.). The top-most compartment shows the task function name and the type of the task result. If the task function has arguments, then these are enumerated in the middle compartment. The bottom compartment contains the rendering of the task function’s body, delimited with ▶ and ▼. The body of **main** is a single task application. Task application uses the same rounded box shape to emphasize that it involves a task function (rules 3.3.3.1-2.). Local task functions are not rendered in Tonic, so the task engineer needs to be aware of this.

The **processPhoneCall** blueprint shows how shared data structures and the monadic style combinator >>= are rendered.

```
processPhoneCall yields an Emergency

read

currentUser

currentDateTime

now

logCall

now

now
```

To emphasize the ‘external’ nature of shared data sources, they are depicted using the conventional symbol for disk storage and branch in-to and/or out-of the task flow. We still need to develop more appropriate symbols for shared data sources such as **currentDateTime**. When using >>=, the task engineer can explicitly bind the result of the first task to a pattern which is used to label the edge (rule 3.3.3.2.1). If the pattern can be extracted from the task function specification, then Tonic adds it as a label (rule 3.3.3.2.2). This rule is applied in the blueprint. In any other case, the right hand side task is just included verbatim (rule 3.3.3.2.3). Unlabeled edges are used for >>| (rule 3.3.3.2.4).

The **logCall** blueprint shows that in the current Tonic version editors have no special visual presentation and are rendered as task applications. Task value transformers are not task applications and require separate attention (rules 3.3.3.3).

```
logCall yields an Emergency

now is a DateTime

enterInformation

"Enter call information"

[ ]

makeEmergency

now

now
```

The task value transformer function is displayed within a rotated, stretched chevron. Tonic supports lambda abstraction and partial functional application. Variable names from a function’s definition are reified when it is partially applied.

The **handleEmergencyCall** blueprint renders the task assignment @: combinator (rules 3.3.3.7) and the step >>= combinator (rules 3.3.3.6).
CHAPTER 3. TONIC: A GRAPHICAL REPRESENTATION OF TASKS

handleEmergencyCall yields a Verdict
intake is a (Task Emergency)

Any user with role call-intaker

continue

call
requiresAuthorities
call
coordinate
call
Fake call

FakeCall

Many distributed systems have extensive worker schemes that determine who is allowed to do what. iTasks’ support for these cases are distinguished in Tonic. The core step >*
combinator is a powerful tool that captures many task patterns, because in general, both the list of follow-up tasks and these tasks themselves can be arbitrary computations. This implies that it is out of the scope of the Tonic project to render arbitrary applications of this combinator. Follow-up tasks are rendered separately only if all of them are statically enumerated. The second restriction is that only follow-up task specifications that use one of the frequently occurring computation patterns (always and ifValue, see Section 3.2) are detected and rendered. The rendering visualizes the case analysis that needs to take place between the task engineers and domain experts: what is the follow-up task in case of the three sorts of possible task values, absent Ø, unstable ✧, and stable ⚪? In case the follow-up task requires triggering by a worker then this is rendered with ◆ and labeled with the action □.

In rendering the coordinate blueprint we encounter similar issues with the use of the core parallel task combinator, of which allTasks is an instance.

coordinate yields a Verdict
call is an Emergency

update emergencies

add call

Any user with role call-coordinator

alertAuthoritiesAbout call: +<verdicts showSuccessOfVerdicts verdicts

verdicts

Instead of attempting to render the parallel task combinator, Tonic detects two groups of frequently occurring parallel patterns. The first is delimited with ◆ and represents product-style patterns that acquire all sub-task values (rules 3.3.3.5.1-4.). The second is delimited with ◆ and represents sum-style patterns that acquire the first available sub-task value (rules 3.3.3.5.5-10.). Fall-through rules 3.3.3.5.4
and 3.3.3.5 10 are used in case the list-versions of the parallel combinators do not enumerate all sub-tasks statically.

The final blueprint shows host language elements (rules 3.3.3.4).

<table>
<thead>
<tr>
<th>showSuccessOfVerdicts yields a Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>verdicts is a [Verdict]</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{collectFailures} & \quad \\
\text{fails} & \\
\text{msg} = \text{join "\n" fails} & \\
\text{viewInformation} & \\
(\text{"Uninformed authorities"}) & \\
\text{msg} & \\
\text{Success} & \\
\text{Fail msg} & \\
\end{align*}
\]

For rendering the case and if blocks we are inspired by the BPMN notation for decision nodes, representing them as diamonds containing the case expression with labeled edges going from the diamond to the cases. The let blocks are represented as plain boxes in which the definitions are rendered verbatim.

### 3.3.3 Blueprint production rules

In this subsection we list all blueprint production rules. Rules for related language and task constructs are grouped together. Each rule has its own unique identifier, which is located on the equals-sign.
3.3.3.1 Task definitions and task applications

\[ f = e \]
iff \( f :: \text{Task} \alpha_0 \)
\& \( f \) is top-level

\[ f \, p_1 \ldots p_n = e \]
iff \( f :: \alpha_1 \ldots \alpha_n \rightarrow \text{Task} \alpha_0 \)
\& \( f \) is top-level

\[ f \]
iff \( f :: \text{Task} \alpha \)
\[ f \, e_1 \ldots e_k \]
iff \( f :: \alpha_1 \ldots \alpha_n \rightarrow \text{Task} \alpha_0 \)
\& \( n \geq k \)

3.3.3.2 Monadic combinators

\[ e_0 >>= \lambda p \rightarrow e_1 \]
\[ e_0 >>= f \, e_1 \ldots e_k \]
iff \( f \, p_1 \ldots p_k \, p_{k+1} \ldots p_n = e_{n+1} \)

\[ e_0 >>= e_1 \]
\[ e_0 >>= e_1 \]
\[ \text{return} \, e \]

3.3.3.3 Transformation combinators

\( \lambda p \rightarrow e_1 <\$> \, e_0 \)
\[ f \, e_1 \ldots e_k <\$> \, e_0 \]
iff \( f \, p_1 \ldots p_k \, p_{k+1} \ldots p_n = e_{n+1} \)
3.3.3.4 Clean language constructs

\[
\begin{align*}
\text{let } & p_1 = e_1 \\
& \vdots \\
\text{in } & e_0 \\
\text{iff } & e_0 :: \alpha_1 \ldots \alpha_k \rightarrow \text{Task } \alpha_0 \\
\text{iff } & e_1 \ldots e_n :: \alpha_1 \ldots \alpha_k \rightarrow \text{Task } \alpha_0 \\
\text{iff } & e_0 :: \alpha_1 \ldots \alpha_n \rightarrow \text{Task } \alpha_0 \\
\text{iff } & e_1, e_2 :: \alpha_1 \ldots \alpha_n \rightarrow \text{Task } \alpha_0 \\
\text{iff no other case matches} \\
\end{align*}
\]

3.3.3.5 Parallel combinators

\[
\begin{align*}
\text{allTasks } e & \\
\text{if } & e_0 \&\& e_1 \\
\text{where } & 0 \leq i \leq n \\
\text{no other case matches} \\
\text{anyTask } e & \\
\text{if } & e_0 \mid\mid e_1 \\
\text{if } & e_0 \mid\mid e_1 \\
\text{if } & e_0 \mid\mid e_1 \\
\text{where } & 0 \leq i \leq n \\
\text{no other case matches} \\
\end{align*}
\]
3.3.3.6 Step

\[ [e_0 \gg\gg [e_1, \ldots, e_n]] \]
where \( 1 \leq i \leq n \)

\[ [e_0 \gg\gg e_1] \]
\[ [\text{OnValue } f]_{\text{step}} \]
\[ [\text{OnAction (Action btn _)} f]_{\text{step}} \]
\[ [\text{OnException } f]_{\text{step}} \]
\[ [\text{OnAllExceptions } f]_{\text{step}} \]
\[ [\text{always } e]_{\text{step}} \]
\[ [\text{hasValue } f]_{\text{step}} \]
\[ [\text{ifStable } f]_{\text{step}} \]
\[ [\text{ifUnstable } f]_{\text{step}} \]
\[ [\text{ifCond } e f]_{\text{step}} \]

\[ [\text{ifValue } (f e_0 \ldots e_k) g]_{\text{step}} \]
iff \( f \ p_0 \ldots p_k \ p_{k+1} \ldots p_n = e_{n+1} \)

\[ [e]_{\text{step}} \]
iff no other case matches
\[ ([\lambda p \rightarrow e])_{\text{edge}} \]

\[ [f e_0 \ldots e_k]_{\text{edge}} \]
iff \( f \ p_0 \ldots p_k \ p_{k+1} \ldots p_n = e_{n+1} \)

\[ [e]_{\text{edge}} \]
iff no other case matches

\[ p \rightarrow [e] \]

variable names in \( p_{k+1} \) are fresh

variable names in \( p_{k+1} \) are fresh
3.3.3.7 Assign combinator

\[ [u \otimes e] \]

\[ [\text{AnyUser}]assign \]
\[ [\text{UserWithId ident}]assign \]
\[ [\text{UserWithRole r}]assign \]
\[ [\text{SystemUser}]assign \]
\[ [\text{AnonymousUser \_}]assign \]
\[ [\text{AuthenticatedUser ident r}]assign \]
\[ [\text{ident}]assign \]

1. \( [u]assign \)
2. \( e \)
3. \( [e]assign \)
4. \( [\text{Any user}]assign \) with role \( r \)
5. \( [\text{User}]assign \) with role \( r \)
6. \( [\text{Any user}]assign \) with role \( r \)
7. \( [\text{User}]assign \) with role \( r \)
8. \( [\text{User}]assign \)

3.3.3.8 Shares

\[ [\text{get e}] \]
\[ [\text{set e1 e2}] \]
\[ [\text{upd e1 e2}] \]

\[ [\text{sharedStore e1 e2}]share \]

\[ [f e_1 \ldots e_n]share \]
\( \text{iff } f :: \alpha_1 \ldots \alpha_n \rightarrow \text{ReadWriteShared } r \ w \)

\[ [f]share \]
\( \text{iff } f :: \text{ReadWriteShared } r \ w \)

3.3.3.9 Fallthrough

\[ [e] \]
\( \text{iff } \) no other case matches
3.3.4 Discussion

The blueprints shown Section 3.3.2 that are produced by the rules in Section 3.3.3 demonstrate that Tonic attempts to capture the task structure of a task specification. We have to find out, using the feedback of our users, what is the best way to inform the domain experts and end users. Some constructs are technically challenging. We need to address task combinators that use lists of sub-tasks more precisely to better capture the structure of the list. Tonic has no special rendering for recursive task structures. We need to verify with the domain experts if it is necessary to visually emphasize that a task structure is recursive. Despite the above issues, we feel that the current Tonic blueprints are helpful when communicating with domain experts. We observe that task engineers, when confronted with the blueprints that are generated from their task specifications, tend to refactor them by moving and naming elements that are not immediately related to the task structure to the where-clause.

3.4 Dynamic Tonic blueprints

iTasks does not only offer an API for defining tasks, it also includes a run-time system which coordinates the tasks. iTasks’ generic machinery generates a web-based GUI with which users can interact with the system. Each time an end user interacts with the iTask system, the consequences of that user event are calculated by the run-time system. Commonly, an event not only affects the corresponding user or system, but also all other participants who are observing the tasks via a step combinator or via shared information. Push technology is used to automatically update their view on the observed tasks. In this section we show how the Tonic infrastructure uses and extends the iTasks system for the purpose of viewing blueprints during run-time. In Section 3.4.1 we describe which information is readily available in the iTasks run-time and which information must be generated by Tonic to trace the execution of tasks, their generated blueprints and their interconnection. In Section 3.4.2 we show how this information is used by yet another task, the Tonic viewer, to display blueprints at run-time.

3.4.1 Storing task progress at run-time

While a static blueprint shows the definition of a task, we are now interested in its dynamic behaviour: how it is instantiated. At run-time we want to show the blueprint corresponding with a task under execution; the actual values of the parameters of a particular task application; the current value of its result which can change over time; and for all tasks shown in the blueprint whether they are finished, active, or not yet activated. The blueprints of all task instances in the blueprint should be recursively inspectable in the same way.

To be able to show the instantiated blueprints, we have to link compile-time task information to run-time information of the tasks being executed. We have modified both the Clean compiler and the iTasks run-time system to make this
3.4. DYNAMIC TONIC BLUEPRINTS

possible. Besides generating a blueprint for every defined task, the Tonic pass in
the modified compiler inserts calls to two wrapper functions.

\[
\begin{align*}
tonicWrapTaskBody &:: (ModuleName, TaskName) [(VarName, Task ())] (Task a) \rightarrow Task a \\
tonicWrapApp &:: (ModuleName, TaskName) Int (Task a) \rightarrow Task a
\end{align*}
\]

tonicWrapTaskBody is used to wrap the body of a task definition, i.e. a Clean
function of type \texttt{Task a}. It is used to inform the run-time system about the value
of the actual parameters the task is called with, the current state of the task
value, as well as the name of the task. The wrapper adds a record of type TonicRT
to a shared store, tonicSharedRT of type \([\text{TonicRT}]\). The record links compile
time to run-time information. Each field in the code snippet below contains an
accompanying comment. For each field, the comment either contains \texttt{RT}, to signify
that the information in that field is only available at run-time, or \texttt{CT}, to indicate
that the information in that field is already known at compile-time.

\[
\begin{align*}
:: \text{TonicRT} = \\
\{ & \text{trt\_taskId} :: \text{TaskId} \quad // \text{id of the task, RT} \\
& \text{trt\_bpref} :: (ModuleName, TaskName) \quad // \text{blueprint id, CT} \\
& \text{trt\_params} :: [(VarName, Task ())] \quad // \text{editors for parameters, CT} \\
& \text{trt\_parentTaskId} :: \text{TaskId} \quad // \text{task id of parent, RT} \\
& \text{trt\_output} :: \text{Maybe( Task () )} \} \quad // \text{editor to show result, CT}
\end{align*}
\]

The unique \texttt{TaskId}, which is only known at run-time, needs to be linked to its
corresponding blueprint. A blueprint is uniquely identified by the pair of module
name and task name \((\text{ModuleName, TaskName})\), known only at compile-time.

tonicWrapApp is wrapped around an application of a task to inform the run-
time system which node in the corresponding blueprint of the parent task is being
activated. Nodes are counted and can be identified by an \texttt{Int}.

We want to show what the actual values of the parameters of a task are at
run-time. The formal names of task parameters (of type \texttt{VarName}) are used to
identify them. To display a parameter value, which can be of any type, we make
use of iTasks editors, since they are conceptually capable of showing values of
any (first order) type. As long as the type of a parameter satisfies the iTasks
context restriction, we can use an editor such as \texttt{viewInformation} to show the
parameter's value. As a result, the value will be displayed in the format which
is commonly used for this particular type. \texttt{trt\_params} maps formal parameters
to their corresponding editor. The compiler can determine whether the context
condition holds and generates an editor displaying a default message otherwise.

Every task called is an instance somewhere in a blueprint of a parent, with the
exception of the top-most task. So, to be able to update the parents blueprint with
the proper information, we need to know the parent’s task-id. For this purpose,
a call stack is added and maintained in the iTasks run-time system. While stack-
tracing is a hard problem in functional languages in general, we can produce
reliable stack traces due to iTasks’ monadic execution model, which maintains an
internal state.

Finally, to show the current value of a task executed at run-time, we also store
an editor for the output value in \texttt{trt\_output}.
CHAPTER 3. TONIC: A GRAPHICAL REPRESENTATION OF TASKS

3.4.2 Viewing task progress at run-time

With help of the wrapper functions, all information we need to enable dynamic blueprints is available at run-time in shared stores. Below, we show the main structure of the Tonic viewer with which end-users can select a blueprint instance and view its current state and arguments. Figure 3.3 gives a screenshot of using this viewer on a blueprint instance of logCall.

```
viewDynamic :: Task ()
viewDynamic
  = enterChoiceWithShared "Active_blueprint_instances" [] tonicSharedRT >>=
    \trt=(\trt_bpinstance,\trt_activeNodeId)-> maybe (return ())(bp -> viewInformation (title trt bp) [] ()
      ||- args trt bp
      ||- viewSharedInformation "Blueprint:"
        [ViewWith (\_ -> toniclet bp trt_activeNodeId)]
      tonicSharedRT 0! () \trt_bpinstance
where
  title trt bp = snd trt.trt_bpref ++ "\_yields\_" ++ bp.tt_resty
  args trt bp = enterChoice "Task_arguments" [ChooseWith (ChooseFromList fst)]
    (zipWith (\(arg,type) (\_,view) -> (arg++"\_is\_"++ type,view))
      bp.tt_args trt.trt_params) >>= withSelection snd
```

A detailed discussion of the viewer task is out of scope of this chapter, so we focus on the main components. First of all, the end-user can continuously select any of the currently registered TonicRT records from the tonicSharedRT shared data source (line 3). Given a selected instance, its title is displayed (line 6 and 12), the arguments are enumerated and can be viewed (line 7 and 13–15), and the blueprint itself is rendered (line 8–10). The toniclet task (line 9) uses JavaScript to render the blueprint.
3.5 Related work

We are not the first to propose a graphical syntax for functional programming languages. In 1994, Poswig et al. [88] and Reekie [89] independently proposed graphical syntax for visually writing functional programs, called VisaVis and Visual Haskell, respectively. In 2002, Hanna [50] proposed an interactive visual functional programming environment called Vital. Later, Elliott [41] proposed “Tangible Values” as a way to define typed, interactive visualizations for representing values, which can then be graphically composed by end-users. Several years after that Henrix et al. [53] presented a graphical programming language specifically for iTasks programs, called GiN. Our work and our choice for Tonic’s name are greatly inspired by GiN. Tonic can be seen as the inverse of GiN.

What sets our work apart from the aforementioned works is that, rather than enabling people to write programs in a visual language, we generate a graphical representation from TOP source code instead. In addition, our work focusses on giving non-programmers insight in what programmers have written, rather than providing an alternative for programming itself.

Work on graphical formalisms has been done outside of functional programming as well. Petri nets [80] are a graphical representation of automata, dating back to 1966. In industry, UML [76, 77] is used to visually describe the implementation of a program and BPMN [105] is used to graphically model business processes. Some tools, like Microsoft Visual Studio and Visual Paradigm, offer the ability to generate UML from Java or .NET languages. They do not offer the ability to visualize run-time information, however. Reverse engineering Java to Petri nets has been proposed by Fuhs and Cannady [47], but again, they only focus on static representations.

3.6 Discussion and conclusion

In this chapter we have presented Tonic, a novel system that generates a graphical representation, called a blueprint, of an iTasks program in order to narrow the communication gap between non-technical project stakeholders and programmers. We have shown that it is possible to generate static blueprints from the source of an iTasks program. By visualizing programs on the monadic abstraction level of tasks and by visualizing only a limited set of Clean language constructs, we have reduced the number of graphical elements in a blueprint, making it easier to understand for non-programmers. We also generate wrapper code and extended the iTasks run-time system so that we can visualize the actual instantiation of the blueprints and the value of task arguments at run-time.

Limiting ourselves to small set of Clean language constructs has been done deliberately to keep the graphical language simple for the non-technical stakeholders. Although the blueprint design has been done in collaboration with these stakeholders, we have not been able to test its practical usability systematically. The limited set of language constructs has as disadvantage that the iTasks programmer must obey a certain coding convention, avoiding the use of advanced
syntactical constructs in the body of a task definition.

Currently, we are using special graphical syntax for a specific set of task combinators. All other tasks are visualized as task application. End-users should be able to define custom graphical syntax for their own custom tasks, enabling domain-specific visualizations.

Real iTasks applications may contain several thousands of tasks. At runtime, one has to be able to browse through a huge collection of instantiated blueprints. We need to think about how to do so, such that a manager, for example, can easily keep track of what is going on.

Tonic may be more generally applicable than for iTasks only. We may be able to generate blueprints for a larger class of programs, e.g., all monadic programs, and use them for debugging and tracing of those applications.
Chapter 4
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.

4.1 Introduction

The iTask system \[86, 68\] (iTasks) is an implementation of the Task Oriented Programming (TOP) paradigm in the strongly typed, lazy, purely functional programming language Clean \[87\]. 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 \[51\], and indeed, the form-like GUIs generated by iTasks adhere to this property.

For many application domains, such as status displays or games, 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 [98]. 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 a fire’s developments or a leak’s water levels. We think 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 chapter, 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 use SVG as graphical user interface.

A real-world use-case, which we address specifically in this chapter, concerns multi-player, distributed games such as Trax [1] and Ligretto. We demonstrate how the latter card game can be created with the Graphics.Scalable image library and iTasks.

In this chapter we make the following contributions:

- We present the purely compositional Graphics.Scalable library.
- We define interaction on images using pure functions.
- We integrate Graphics.Scalable in iTasks in an orthogonal way.
• We demonstrate its usage by implementing a game called Ligretto.

• We map Graphics.Scalable images to the Scalable Vector Graphics (SVG) standard [27].

• We show how we have overcome the technical challenges imposed by the Internet’s client/server architecture using iTasks’ editlet infrastructure.

We start our explanation by first concentrating on static, purely compositional images as provided in Graphics.Scalable in Section 4.2. 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 4.3). We show how static images are made interactive in Section 4.3 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 4.4.

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 4.5 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 4.6.

### 4.2 Compositional Static Images

In this section we describe the compositional image library (Sections 4.2.1–4.2.6). The concepts are illustrated step by step by rendering the entire state of the Ligretto card game (Section 4.2.7).

#### 4.2.1 Image concepts

Conceptually, an image is an infinitely large, perfectly transparant ‘slide’ that renders a value of some model type \( m \). This is captured with the opaque type 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 4.2.3). 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 4.2.4). 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.

4.2.2 Basic images

The image library supports common shapes as basic images:

:: Span // an opaque data type

px :: Real -> Span // (px x) represents x pixels

empty :: Span Span -> Image m
circle :: Span 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, hence 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 occuring 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, i.e.: using a stroke of one pixel and filled with the default colour black. These can be changed with the image attributes (Section 4.2.3). Finally, lines are also drawn with a default stroke of one pixel and use the colour 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).
4.2.3 Image attributes

Image attributes alter the appearance of visual elements \textit{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 \textit{name-value} pairs. We adopt the SVG names:

\[
\begin{align*}
:: \text{StrokeAttr} m &= \{ \text{stroke} :: \text{SVGColour} \} \\
:: \text{StrokeWidthAttr} m &= \{ \text{strokeWidth} :: \text{Span} \} \\
:: \text{XRadiusAttr} m &= \{ \text{xradius} :: \text{Span} \} \\
:: \text{YRadiusAttr} m &= \{ \text{yradius} :: \text{Span} \} \\
:: \text{FillAttr} m &= \{ \text{fill} :: \text{SVGColour} \} \\
:: \text{OpacityAttr} m &= \{ \text{opacity} :: \text{Real} \} \\
:: \text{DashAttr} m &= \{ \text{dash} :: [\text{Int}] \} \\
:: \text{MaskAttr} m &= \{ \text{mask} :: \text{Image m} \}
\end{align*}
\]

Each type constructor is made an instance of a type constructor class \texttt{tuneImage}, having trivially derived operators and function.

\[
\begin{align*}
\text{class tuneImage} \ &\text{attr} :: (\text{Image m}) (\text{attr m}) \rightarrow \text{Image m} \\
\text{:>}> \ &\text{infixr 2} :: (\text{attr m}) (\text{Image m}) \rightarrow \text{Image m} | \text{tuneImage attr} \\
\text{::<} \ &\text{infixl 2} :: (\text{Image m}) (\text{attr m}) \rightarrow \text{Image m} | \text{tuneImage attr} \\
\text{tuneIf} :: \ &\text{Bool} (\text{Image m}) (\text{attr m}) \rightarrow \text{Image m} | \text{tuneImage attr}
\end{align*}
\]

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

\[
\begin{align*}
\text{class toSVGColour a} :: a \rightarrow \text{SVGColour} \\
\text{instance toSVGColour String, RGB} \\
:: \text{RGB} = \{ \text{r} :: \text{Int}, \text{g} :: \text{Int}, \text{b} :: \text{Int} \}
\end{align*}
\]

4.2.4 Image composition

Images are composed by stacking. The images that are to be stacked are given in a \textit{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 \textit{x}-axis and \textit{y}-axis unspecified. This relative layout can be defined with or without a \textit{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 \textit{x}-span and \textit{y}-span value. The initial layout of images is \textit{always} computed without the offsets. The final layout is obtained by adding the \textit{i}-th offset to the initial position of the \textit{i}-th image.

\[
\begin{align*}
:: \text{Layout m} &=\{ [\text{ImageOffset}] \rightarrow [\text{Image m}] \rightarrow (\text{Host m}) \rightarrow \text{Image m} \\
:: \text{Host m} &=\{ \text{Maybe (Image m)} \\
:: \text{ImageOffset} &=\{ \text{Span, Span} \}
\end{align*}
\]

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. This way, we can keep the specification of the image list separate from other concerns, such as offsets and alignments (used 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:

\[
\text{collage} :: \text{Layout} \ m
\]

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:

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

\[
\text{ImageAlign} ::= (\text{XAlign}, \text{YAlign})
\]

\[
\text{XAlign} = \text{AtLeft} | \text{AtMiddleX} | \text{AtRight}
\]

\[
\text{YAlign} = \text{AtTop} | \text{AtMiddleY} | \text{AtBottom}
\]

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:

\[
\text{grid} :: \text{GridDimension} \ \text{GridLayout} \ [\text{ImageAlign}] \rightarrow \text{Layout} \ m
\]

\[
\text{GridDimension} = \text{Rows Int} | \text{Columns Int}
\]

\[
\text{GridLayout} ::= (\text{GridMajor}, \text{GridXLayout}, \text{GridYLayout})
\]

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 the final positions for all images.

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.

\[1\text{Although internally, other layout combinators are modeled explicitly as well for reasons of efficiency.}\]
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:

```haskell
class margin a :: a (Image m) -> 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.

### 4.2.5 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:

```haskell
:: ImageTag

  // Symbolic span values:
texxspan :: FontDef String -> Span // text width
imageyspan :: ImageTag -> Span // image width
imageyspan :: ImageTag -> Span // image height
columnspan :: ImageTag Int -> Span // column width
rowspan :: ImageTag Int -> Span // row height

  // Symbolic span arithmetic:
instance zero Span
instance + Span
instance - Span
instance ~ Span
instance abs Span

class (\*) infixl 7 a :: a n -> a | toReal n
class (/\_) infixl 7 a :: a n -> a | toReal n
instance * Span, Real, Int
instance / Span, Real, Int

minSpan :: [Span] -> Span
maxSpan :: [Span] -> 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:

```haskell
tag :: *ImageTag (Image m) -> 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 (`+`, `-`, `abs`, and `~`). 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 `/`, for multiplication and division with a scalar value, and `minSpan` and `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 Section 4.4.3). This expressive power is also available for the image author who can use the same language to define new image layout patterns.

### 4.2.6 Image transformations

Any (composite) image can be subject to transformation:

```haskell
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
```
4.2. COMPOSITIONAL STATIC IMAGES

:: 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.

4.2.7 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 4.2.7.1) and then show how it is rendered (Section 4.2.7.2).

4.2.7.1 Ligretto model types

Ligretto is a card game for two, up to twelve players. In this chapter we restrict ourselves to a maximum of four players. Each player has forty cards that come in four front colours: red, green, blue, and yellow. The ten cards of one colour are numbered on the front side from one through ten. For identification purposes, the back sides of the cards have a unique colour for each player. These facts can be modeled in a straightforward way:

:: Card = { back :: Colour, front :: Colour, no :: Int }
:: SideUp = Front | Back
:: Colour = 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 4.1(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 4.1(j)). In the middle, piles of cards of the same front colour 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 colour and top-most card having number \( n \). Although players are uniquely
identified via their colour, we also keep track of their name and render it in the
game. These facts are modeled as follows:

\[
\begin{align*}
\text{:: NoOfPlayers} & :== \text{Int} \\
\text{:: Middle} & :== \text{[Pile]} \\
\text{:: Pile} & :== \text{[Card]} \\
\text{:: Player} & == \{ \text{colour} :: \text{Colour} \\
& \quad , \text{name} :: \text{String} \\
& \quad , \text{row} :: \text{RowPlayer} \\
& \quad , \text{ligretto} :: \text{Pile} \\
& \quad , \text{hand} :: \text{Hand} \\
& \quad , \text{seed} :: \text{Int} \}
\end{align*}
\]

\[
\begin{align*}
\text{:: RowPlayer} & :== \text{[Card]} \\
\text{:: Hand} & == \{ \text{conceal} :: \text{Pile}, \text{discard} :: \text{Pile} \}
\end{align*}
\]

\[
\begin{align*}
\text{no_of_cards_in_row} & :== \text{NoOfPlayers} -> \text{Int} \\
\text{colours} & :== \text{NoOfPlayers} -> \text{[Colour]}
\end{align*}
\]

The complete Ligretto game state consists of the middle card piles and the par-
ticipating players:

\[
\text{:: GameSt} == \{ \text{middle} :: \text{Middle}, \text{players} :: \text{[Player]} \}
\]

We can now turn our attention to rendering this game state.

### 4.2.7.2 Ligretto rendering

The Ligretto game state is rendered step by step in a compositional way. The
individual images are shown in Figure 4.1. 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:

\[
\begin{align*}
\text{card_width} & = \text{px} 58.5 \\
\text{card_height} & = \text{px} 90.0
\end{align*}
\]

The shape of a Ligretto card is that of a rectangle with rounded corners (Figure
4.1(a)):
4.2. COMPOSITIONAL STATIC IMAGES

card_shape = rect card_width card_height
  <@> {xradius = card_height / .18}
  <@> {yradius = card_height / .18}

For rendering the text on cards we use the font family *Verdana* in several sizes:

cardfont size = normalFontDef "Verdana" size

The model colours need to be mapped to SVG colours that best match the physical cards. We select the following SVG colours:

*instance* toSVGColour Colour where
  toSVGColour Red = toSVGColour "darkred"
  toSVGColour Green = toSVGColour "darkgreen"
  toSVGColour Blue = toSVGColour "midnightblue"
  toSVGColour Yellow = toSVGColour "gold"

We abbreviate *white* and *black*:

white = toSVGColour "white"
black = toSVGColour "black"

The number on the front side of a card is displayed in a large font (Figure 4.1(b) shows big_no 7 Red):

big_no no colour = text (cardfont 20.0) (toString no)
  <@> {fill = white}
  <@> {stroke = toSVGColour colour}

At the back side of the card, the text *Ligretto* is displayed (Figure 4.1(c) shows ligretto Red):

ligretto colour = text (cardfont 12.0) "Ligretto"
  <@> {fill = toSVGColour "none"}
  <@> {stroke = toSVGColour colour}

With these image functions, we can render the front side (Figure 4.1(d)) or back side (Figure 4.1(e)) of a card:

```
card_image :: SideUp Card -> Image m

card_image side card
| side == Front
  = let no = margin (px 5.0) (big_no no (no_stroke_colour card.front))
    in overlay [(AtMiddleX, AtTop), (AtMiddleX, AtBottom)] []
                  [no, rotate (deg 180.0) no] host
  | otherwise = overlay [(AtMiddleX, AtBottom)] []
                      [skewy (deg -20.0) (ligretto card.back)] host

where
host = Just (card_shape <@> {fill = if (side == Front)
                               (toSVGColour card.front)
                               white})
```

The stroke colour of the card number depends on the card colour:

```
no_stroke_colour :: Colour -> Colour

no_stroke_colour Red = Blue
```
We introduce an ‘empty card’ that serves as a visual placeholder for an empty pile (Figure 4.1(f)).

```
no_card_image :: Image m
no_card_image = overlay [(AtMiddleX, AtMiddleY)] []
    [text (pilefont 12.0) "empty"] host
  where
    host = Just (card_shape <$> {fill = toSVGColour "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 4.1(g)):

```
pile_of_cards :: SideUp Pile -> Image m
pile_of_cards side pile = overlay [] [(zero, card_height / 1.18 * 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 4.1(h)).

```
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 colour. Instead of scaling long or short names, we use masking to prevent long names from running outside of the host image (Figure 4.1(i) shows the result for a player named *alice* playing the red cards).

```
ame_image :: Player -> Image m
name_image {name, colour} = overlay [(AtMiddleX, AtMiddleY)] []
    [text {cardfont 16.0 & fontweight = "bold"} name
      <$> {fill = if (colour === Yellow) black white}
    ] host
  <$> {mask = rect width height <$> {fill = white}
    <$> {stroke = white}}
  where
```

58
4.2. COMPOSITIONAL STATIC IMAGES

width = card_height * 1.8
height = card_width * 0.4
host = Just (rect width height @$@< {fill = toSVGColour colour})

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 of a circle with radius $r$:

$$\text{circular} :: \text{Span Real \{Image m\}} \rightarrow \text{Image m}$$
$$\text{circular} \ r \ a \ imgs = \text{overlay} (\text{repeat} (\text{AtMiddleX,AtMiddleY}))$$
$$[ (r * \ \text{cos} \ \text{angle}, r * \ \text{sin} \ \text{angle})$$
$$\ \ \ \| \ \ i \ \ <- [0.0, \ \text{sign_a..}]$$
$$\ \ \ \| \ \ \ \text{angle} \ <- [i * \ \text{alpha} - 0.5 * \ \pi]$$
$$\ \ \ \| \ \ \ \text{rotate} (\text{rad} (i * \ \text{alpha})) \ \text{img}$$
$$\ \ \ \| \ \ i \ <- [0.0, \ \text{sign_a..}]$$
$$\ \ \ \| \ \ & \ \text{img} \ <- \ \text{imgs}$$
$$\ \ \ \| \ \ (\text{Just} \ (\text{empty} (r * 2.0) (r * 2.0)))$$

where

$$\text{sign_a} = \text{toReal} \ (\text{sign} \ a)$$
$$\text{alpha} = \text{toRad} \ (\text{normalize} \ (\text{rad} \ a)) / \text{toReal} \ \text{(length imgs)}$$

The circular image is created by stacking all images with their centres (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:

$$\text{middle_image} :: \text{Span Middle} \rightarrow \text{Image m}$$
$$\text{middle_image} \ r \ \text{middle} = \text{circular} \ r \ (2.0 * \ \pi) \ (\text{map} \ \text{pile_image Front}) \ \text{middle}$$

Figure 4.2(a) shows the result of the initial middle for three players. It consists of twelve empty piles, as each player has the potential to start four piles.

The middle tier, names_image, distributes all player names along a full circle:

$$\text{names_image} :: \text{Span} \ [\text{Player}] \rightarrow \text{Image m}$$
$$\text{names_image} \ r \ \text{players} = \text{circular} \ r \ (2.0 * \ \pi) \ (\text{map} \ \text{name_image players})$$

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).

$$\text{hand_images} :: \text{Hand} \rightarrow \text{[Image m]}$$
$$\text{hand_images} \ \{\text{conceal}, \text{discard}\} = [ \ \text{pile_image Back conceal}$$
$$\ \ \ \| \ \text{, pile_image Front discard} ]$$

$$\text{row_images} :: \text{RowPlayer} \rightarrow \text{Image m}$$
$$\text{row_images} \ \text{row} = \text{map} \ \text{card_image Front} \ \text{row}$$

The player cards are placed along a circle segment that is slightly less than a quarter of a circle (Figure 4.2(b)):
CHAPTER 4. PURELY COMPOSITIONAL INTERACTIVE SCALABLE VECTOR GRAPHICS

Figure 4.2: Compositional rendering of the Ligretto game state: composed sub-pieces

\[
\text{player}_\text{arc} = 0.45 \times \pi
\]

\[
\text{player}_\text{image} :: \text{Span Player} \to \text{Image m}
\]

\[
\text{player}_\text{image} \ r \ \{ \text{row, ligretto, hand} \} \\
= \text{circular } r \ \text{player}_\text{arc} \ ( \ \text{row}_\text{images} \ \text{row} \\
++ \ [ \text{pile}_\text{image} \ \text{Front ligretto} \\
:\ \text{hand}_\text{images} \ \text{hand} ] )
\]

The outermost tier, \text{players}_\text{image}, distributes all player cards along a full circle:

\[
\text{players}_\text{image} :: \text{Span [Player]} \to \text{Image GameSt}
\]

\[
\text{players}_\text{image} \ r \ \text{players} \\
= \text{rotate (rad angle) (circular zero (2.0 * pi) (map (player}_\text{image} \ r) \text{players))}
\]

\[
\text{where} \\
\text{angle} = \text{player}_\text{arc} / (\text{toReal } (2 * \text{no})) - \text{player}_\text{arc} / 2.0 \\
\text{no} = 3 + \text{no_of_cards_in_row (length players)}
\]

Without the additional rotation, the first player’s cards are displayed as shown in Figure 4.2(b). We prefer the layout of Figure 4.3 and therefore rotate the entire image by half the \text{player}_\text{arc}, decreased with half the angle required for one card.

Finally, the entire image overlays the three tiers (Figure 4.3 gives the result of a typical initial Ligretto game state for three players):

\[
\text{game}_\text{image} :: \text{GameSt} \to \text{Image m}
\]

\[
\text{game}_\text{image} \ \{ \text{players, middle} \} = \text{overlay (repeat (AtMiddleX, AtMiddleY)) []}
\]

\[
\text{[ middle}_\text{image} \ (\text{card_height * 2}) \ \text{middle} \\
, \ \text{names}_\text{image} \ (\text{card_height * 3.2}) \ \text{players} \\
, \ \text{players}_\text{image} \ (\text{card_height * 4}) \ \text{players} \\
] \ \text{host}
\]
4.2. COMPOSITIONAL STATIC IMAGES

Figure 4.3: Compositional rendering of the Ligretto game state: final picture

```plaintext
where
host = Just (empty (card_height *. 12) (card_height *. 12))
```

4.2.8 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.
4.3 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 4.3.1). 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 4.3.2). Finally, we show how to turn the static *Ligretto* images interactive, and create a complete TOP specification of a game of *Ligretto* (Section 4.3.3).

4.3.1 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 monadic type `(Task a)`, which has an associated *task value* of type `a`. By inspecting the current task value, other task (functions) can get informed about the state of the task (in progress or finished). Tasks can be composed sequentially, using the *step combinator* `>>*`, or in parallel, using the *parallel combinator*. Examples of their use are given when we continue with the case study in Section 4.3.3.

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:

\[
\begin{align*}
\text{: : ViewOption } a &= E.v: \text{ ViewWith } (a \rightarrow v) & \& \text{ iTask } v \\
\text{: : UpdateOption } a b &= E.v: \text{ UpdateWith } (a \rightarrow v) (a \rightarrow v \rightarrow b) & \& \text{ iTask } v
\end{align*}
\]

`viewInformation` :: `Title [ViewOption m] m` `->` `Task m` `|` `iTask m`

`updateInformation` :: `Title [UpdateOption m m] m` `->` `Task m` `|` `iTask m`

In both cases, the third parameter is the type of the *initial value* that is displayed 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:

\[
\begin{align*}
\text{: : ViewSharedInformation } &= \text{ Title } [\text{ViewOption } r] (\text{ReadWriteShared } r w) \\
& \quad \rightarrow \text{ Task } r & \& \text{ iTask } r \\
\text{: : UpdateSharedInformation } &= \text{ Title } [\text{UpdateOption } r w] (\text{ReadWriteShared } r w) \\
& \quad \rightarrow \text{ Task } w & \& \text{ iTask } r & \& \text{ iTask } w
\end{align*}
\]

\[E.v:\] introduces an *existentially quantified type variable* `v`, while `& iTask v` places a type-class constraint for class `iTask` on `v`. 

---

62
4.3. COMPOSITIONAL INTERACTIVE IMAGES

Figure 4.4 shows the result of applying these editors to a value of type Card (Section 4.2.7.1).

Figure 4.4: Generic Card view- and updateInformation tasks.

Clearly, neither resulting interface is the one that is required for the case study (Figure 4.1(d)). It should be noted that without special support from 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.

4.3.2 Enhancing editors with images

We first integrate static images with editors by introducing a new option for view(Shared)Information editors:

```haskell
imageView :: (r -> *[ImageTag, *ImageTag]) -> ViewOption r | 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 4.4, the following editor:

```haskell
viewInformation "A Ligretto card"
    [ViewWith (imageView \card -> card_image Front card)]
    red_green_7_card_model
dl
```

displays the card graphically, as in Figure 4.1(d).

Interactive images require more effort. First, we introduce a new option for update(Shared)Information editors:

```haskell
imageUpdate :: (r -> v) (v -> *[ImageTag, *ImageTag]) -> Image v (r -> v -> w) -> UpdateOption r w | 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 4.2.3 we have omitted one image attribute:
4.3.3 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 4.2.7.2 into an interactive image (Section 4.3.3.1) and then proceed with the final iTask specification of the entire game (Section 4.3.3.2). In this section we assume the presence of the following pure functions:

\[
\begin{align*}
\text{play_row_card} & : \text{Colour Int GameSt} \rightarrow \text{GameSt} \\
\text{play_concealed_pile} & : \text{Colour GameSt} \rightarrow \text{GameSt} \\
\text{play_hand_card} & : \text{Colour GameSt} \rightarrow \text{GameSt}
\end{align*}
\]

\(\text{(play_row_card player no game)}\) 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. \(\text{(play_concealed_pile player game)}\) 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, \(\text{(play_hand_card player game)}\) 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 4.2.7.1. They ensure that only legal moves can be made.

4.3.3.1 Interactive Ligretto images

The \text{game.image} function defined at the very end of Section 4.2.7.2 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 \text{player_perspective} function which is parametrized with the colour of the player. This colour parameter is also used to make certain that this player can only play her own cards.

\[
\begin{align*}
\text{player_perspective} & : \text{Colour GameSt} \rightarrow \text{Image GameSt} \\
\text{player_perspective colour gameSt} &
\end{align*}
\]
4.3. COMPOSITIONAL INTERACTIVE IMAGES

```haskell
= rotate (rad ((toReal my_no * angle))) (game_image colour gameSt)
where
  angle = 2.0 * pi / (toReal (length gameSt.players))
  my_no = hd [ i \ player <- gameSt.players
             & i <- [0 ..] | player.colour == colour ]
```

(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 colour to the outermost image tier that renders all playable and non-playable cards. The other two image tiers remain static.

```haskell
game_image :: Colour GameSt -> Image GameSt
game_image colour {players, middle} = overlay (repeat (AtMiddleX, AtMiddleY)) []
([ middle_image (card_height *. 2 ) middle
  , names_image (card_height *. 3.2) players
  , players_image (card_height *. 4 ) colour players
  ])
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 Colour [Player] -> Image GameSt
players_image r colour players = rotate (rad angle) (circular zero (2.0 * pi))
[ player_image r (player.colour == colour) 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 specifies whether the image is interactive or not. 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.colour])
```

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 \texttt{play\_concealed\_pile} and \texttt{play\_hand\_card}, but only if the images are interactive:

\begin{verbatim}
hand\_images :: Bool Hand Colour -> [Image GameSt]
hand\_images interactive {conceal,discard} colour
  = [ tuneIf interactive (pile\_image Back conceal)
      {onclick = play\_concealed\_pile colour}
      , tuneIf interactive (pile\_image Front discard)
      {onclick = play\_hand\_card colour} ]
\end{verbatim}

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.

### 4.3.3.2 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 colours. 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:

\begin{verbatim}
play\_Ligretto :: Task (Colour, String)
play\_Ligretto
  = getUser >>=
  \me -> invite\_friends >>=
  \them -> let us = zip2 (colours (1 + length them)) [me : them]
           num\_us = length us
           in allTasks (repeatn num\_us (get random\_Int))
  >>=
  \rs -> let gameSt = { middle = repeatn (4 \* num\_us) [],
                       players = [ initial\_player num\_us c
                                     (toString u) (abs r)
                       \\\ (c, u) <- us \& r <- rs]
                    in with\_Shared gameSt (play\_game us)
\end{verbatim}

currentUser is an SDS that contains a User value describing which user is currently performing the task. random\_Int is another SDS that holds random numbers. (with\_Shared \(v\) \(t\)) creates an SDS with initial value \(v\), and passes it to \(t\). The invite\_friends task terminates only with the correct number of friends.

\begin{verbatim}
invite\_friends :: Task [User]
invite\_friends
  = enter\_Shared\_Multiple\_Choice "Select\_friends\_to\_play\_with" [] users
  >>=
  \you -> if (not (is\_Member (length you) [1 .. 3]))
    ( view\_Information "Oops" []
      "Number\_of\_friends\_must\_be\_1,2,or\_3"
    )
  || invite\_friends
  (return you)
\end{verbatim}
4.3. COMPOSITIONAL INTERACTIVE IMAGES

users is an SDS that contains all known users of the system. A selection of this list can be made with \texttt{enter(Shared)MultipleChoice}.

All players receive a new task to play a game of Ligretto:

\[
\text{play\_game} :: \left[\text{[(Colour, User)]} \right] \text{ (Shared GameSt)} \rightarrow \text{Task (Colour, String)}
\]

\[
\text{play\_game \ users \ game\_st}
\]

\[
= \text{anyTask} \left[u @: \text{play (c, toString u)} \text{ game\_st} \ \| \ (c, u) \leftarrow \text{users}\right]
\]

\text{anyTask} is a parallel task combinator that terminates as soon as one of its sub-tasks terminates. Here, each sub-task, \text{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.\footnote{This is a simplification of the rules of the game in which the remaining points need to be calculated. For brevity we omit this.}

\[
\text{play} :: \text{(Colour,String)} \text{ (Shared GameSt)} \rightarrow \text{Task (Colour, String)}
\]

\[
\text{play (colour,name) game\_st}
\]

\[
= \text{updateSharedInformation name}
\]

\[
\quad \text{[imageUpdate id (player\_perspective colour) (const st)]} \\
\quad \text{game\_st}
\]

\[
\quad \gggg [\text{OnValue (game\_over colour game\_st)}] \\
\text{where}
\]

\[
\text{game\_over me game\_st (Value gameSt _)}
\]

\[
= \text{case and\_the\_winner\_is gameSt of}
\]

\[
\text{Just \{colour, name\} } \rightarrow \text{let won = (colour, name)}
\]

\[
\quad \text{in \ Just (accolades won me game\_st } \gggg | \text{ return won)}
\]

\[
\quad \rightarrow \text{Nothing}
\]

The \text{play} task is an editor enhanced with the player perspective function that has been developed in Section 4.3.3.1. 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 clickable and can be played in any order. The model functions presented in Section 4.3.3 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 \(\gggg\) 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 \text{play} task (and therefore also the \text{anyTask} application in \text{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 disallow further editing of the game state, it is merely rendered as a view.

\[
\text{accolades} :: \text{(Colour, String) Colour (Shared GameSt)} \rightarrow \text{Task GameSt}
\]

\[
\text{accolades won me game\_st} = \text{viewSharedInformation ("The\_winner\_is\," } \gggg | \text{ won)}
\]

\[
\quad \text{[imageView (player\_perspective me)]} \text{ game\_st}
\]
4.3.4 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.

4.4 Implementation

In this section, we explain how images are incorporated in iTasks’ architecture (Section 4.4.1). We give an introduction to SVG and briefly evaluate its strengths and weaknesses (Section 4.4.2). Finally, we discuss how we generate SVG from images (Section 4.4.3).

4.4.1 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 4.3.1, one can also specialize such an editor for a specific concrete type. One can even define rich client tasks, by using editlets [34], 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 [33]. This mechanism allows us to execute any Clean function in the browser. 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.

We have created an SVG editlet to integrate Graphics.Scalable with iTasks. The editlet 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.

4.4.2 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, SVG 1.1 Second Edition is the most recently 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 fallback font. The fallback 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.

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 centre and then translating it yields a different result than first translating the image and then rotating it around its centre. Figure 4.5 shows the problem graphically.

\[\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C}
\end{array}\]

Figure 4.5: SVG rotation and translation in different orders
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 centre. However, when we first perform the rotation and then the translation, we end up with square C. We compensate for this behaviour 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 4.6.

![Figure 4.6: A text’s origin and ascent, descent, and x-heights.](image)

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.

### 4.4.3 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 4.7.

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 the code below. We then only have to concern ourselves with rendering SVG for collages.

```haskell
getXAlign _ _ AtLeft = zero
getXAlign maxX xspan AtMiddleX = (maxX / 2.0) - (xspan / 2.0)
getXAlign maxX xspan AtRight = maxX - xspan

getYAlign _ _ AtTop = zero
getYAlign maxY yspan AtMiddleY = (maxY / 2.0) - (yspan / 2.0)
getYAlign maxY yspan AtBottom = maxY - yspan

toSVG (BasicImage ..) = .. // Omitted for brevity
toSVG (Overlay aligns offsets images host) =
  let allSpans = getAllSpans images
      (maxX, maxY) = getMaxSpans allSpans host
      alignOffsets = [ (getXAlign maxX xspan align, getYAlign maxY yspan align)
        \ (xspan, yspan) <- allSpans
        & align <= aligns ]
    in toSVG (Collage positionOffsets images host)
toSVG (Grid offsets aligns imagess host) =
  let spanss = getAllGridSpans imagess
    offsets = calculateGridOffsets (getColumnXSpans spanss)
        (getRowYSpans spanss) aligns imagess offsetsss =
        fst (foldr (mkRows cellXSpans) ([], zero)
        (zip4 aligns imagess cellYSpans offsetsss))
    mkRows cellXSpans (aligns, images, cellYSpan, offsets) =
        (allOffsets, accYOff) =
        let cols = fst (foldr (mkCols cellYSpan accYOff) ([], zero)
            (zip4 aligns images cellXSpans offsetsss))
        in ([cols : allOffsets], accYOff + cellYSpan)
    mkCols cellYSpan accYOff (align, image, cellXSpan, (manualXOff, manualYOff)) =
        (allOffsets, accXOff) =
        let (imageXSpan, imageYSpan) = getImageSpans image
            alignXOff = getXAlign cellXSpan imageXSpan align
            alignYOff = getYAlign cellYSpan imageYSpan align
            offsetPair = (alignXOff + accXOff + manualXOff,
                          alignYOff + accYOff + manualYOff)
```

Figure 4.7: The SVG generation pipeline
The code omits the implementation details of basic images, since they have a one-to-one correspondence to basic SVG shapes.

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. 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.

### 4.4.4 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 a 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 [12] 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.

4.5 Related work

Peter Henderson’s Functional Geometry (FG) is a seminal approach to purely compositional images [51]. Henderson states [52] 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 beside(p,p) and p are equal. In Graphics.Scalable (and most other approaches), the span box of (beside [] [] [p,p] Nothing) has twice the width of the span box of p. In FG, overlaying images consists of taking the union of graphic elements (52) 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 [44, 45], 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 [5], TkGofer [22], and wxHaskell [67]. More recently, the Diagrams approach by Brent Yorgey [106], 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 [60, 43], 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 4.2.4). 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 colour-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 [38], enhancing it with interaction, resulting in Fran [39] which gave birth to the paradigm of functional reactive programming (FRP) and, amongst others, Yampa [24, 58]. 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 behaviour.

Another different path has been taken by Magnus Carlsson and Thomas Hallgren in their work on the Fudgets system [21]. Just like FRP and TOP, it features combinators to structure the top-level behaviour 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 [45].
4.6 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 modelling 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 behaviour as an iTask and integrates visualization within editors. We have observed this same pattern of working in an earlier experiment [4] 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.
Chapter 5
Static and Dynamic Visualisation of Monadic Programs

iTasks is a shallowly embedded monadic domain-specific language written in the lazy, functional programming language Clean. It implements the Task-Oriented Programming (TOP) paradigm. In TOP one describes, on a high level of abstraction, the tasks that distributed collaborative systems and end users have to do. It results in a web application that is able to coordinate the work thus described. Even though iTasks is defined in the common notion of “tasks”, for stakeholders without programming experience, textual source code remains too difficult to understand. In previous work, we introduced Tonic (Task-Oriented Notation Inferred from Code) to graphically represent iTasks programs using blueprints. Blueprints are designed to bridge the gap between domain-expert and programmer. In this paper, we add the capability to graphically trace the dynamic behaviour of an iTasks program at run-time. This enables domain experts, managers, end users and programmers to follow and inspect the work as it is being executed. Using dynamic blueprints we can show, in real-time, who is working on what, which tasks are finished, which tasks are active, and what their parameters and results are. Under certain conditions we can predict which future tasks are reachable and which not. In a way, we have created a graphical tracing and debugging system for the TOP domain and have created the foundation for a tracing and debugging system for monads in general. Tracing and debugging is known to be hard to realize for lazy functional languages. In monadic contexts, however, the order of evaluation is well-defined, reducing the challenges Tonic needs to overcome.

5.1 Introduction

When developing non-trivial software, one frequently needs to gather the correct requirements and frequently evaluate whether the right software is being built. This can be a hard and time-consuming activity when stakeholders with different backgrounds are involved. This is in part due to the communication gap that exists between experts in unrelated fields.

Task-oriented programming (TOP) is a style of functional programming that, amongst other things, aims to reduce the communication gap between various parties by developing programs in terms of the common notion of tasks. TOP is implemented by iTasks, a shallowly embedded monadic domain-specific language in the general-purpose, lazy, purely functional programming language Clean. iTasks is used to compose multi-user web-based applications. Common technical issues related to distributed client-server settings, such as communication, synchro-
nization, user interface generation, and user interaction, are handled automatically by applying advanced functional programming techniques. These include type driven generic functions, and the ability to store, load and communicate closures in a type safe way using Clean’s dynamic system.

As a result, the iTasks application writer is able to concentrate on the main issues: the tasks that have to be done by the end users in collaboration with the computer systems they use. Although one can now, when writing the application code, concentrate on the things that matter, there still exists a communication gap between various stakeholders. Commonly, domain experts, managers and end users are not used to read and understand textual source code. They prefer pictures, diagrams and natural language instead. Yet it is vital that they are able to evaluate the software that has been built, preferably more quickly than by simply running the program in a testing or production environment.

One way to bridge the communication gap between stakeholders and programmers is to utilise graphical notations. Well-known examples of such notations are BPMN [75] and UML [78]. However, such notations have as disadvantage that they are not part of the actual implementation and cannot practically be used as such. Additionally, since they are not part of the implementation, commonly manual labour is required to keep the models synchronized with the implementation. In practice, these models are prone to becoming outdated, because the cost of maintaining them may be higher than the benefit gained from the up-to-date documentation [11].

In previous work [98] we introduced our own graphical notation, called Tonic (Task-Oriented Notation Inferred from Code). Rather than specifying programs graphically, however, we made a specialised version of the Clean compiler called the Clean-Tonic compiler. This compiler generates a graphical representation, called a blueprint, of the tasks that have been defined in Clean. Since blueprints are generated, they always provide up-to-date documentation of the source code. Implementing Tonic in the Clean compiler is necessary, since iTasks is shallowly embedded in Clean. As a result, programmers can use any Clean language construct to write iTasks programs. Implementing Tonic in the Clean compiler allows us to capture these language constructs in the blueprints we generate.

It is neither practical nor informative to show all the details of the original source code in the blueprints; they would become huge and unreadable. Instead we abstract from certain details yet provide enough information such that one should be able to understand by looking at the pictures which tasks have been defined and understand how these tasks depend on each other. We hope that by doing so, blueprints are easier to understand for non-programmers than the (Clean) code they are generated from.

The first version of the Clean-Tonic compiler, however, did suffer from a number of drawbacks. For one, we could only generate static blueprints. Secondly, it had a hard-coded connection between the compiler and iTasks, which is not desirable for a general-purpose compiler. Thirdly, since the compiler was modified specifically for iTasks, Tonic’s features were not usable in other contexts. Lastly, there was

1 Available in the latest development releases at [https://clean.cs.ru.nl/Download_Clean](https://clean.cs.ru.nl/Download_Clean)
no way to customize the rendering of specific tasks without modifying both the compiler and iTasks.

In this paper, we set out to solve all of the aforementioned problems. We transform monadic programs such that dynamic information can be added to blueprints at run-time, creating dynamic blueprints. With these one can monitor what is happening with the monad during execution. In principle this can be done for any monad, but some programming effort is required to link its execution at run-time to the blueprints generated at compile-time. Our focus in this paper is one specific yet challenging example, the dynamic blueprints for iTasks, which is a highly complex and dynamic system.

iTasks is a challenging example because it is used for developing complex distributed systems. In the real world, people and systems often don’t do their work as planned. Therefore it would be of great help if one were able to inspect what is going on at run-time. This aids, for example, programmers in debugging, domain experts in seeing whether the application works as designed, and managers and end users in tracking progress of workflows.

In essence, we have developed a kind of monitoring, tracing and debugging tool. This is commonly known to be a very challenging tool to make for a lazy functional language, particularly if one realizes that Clean applications are not interpreted, but compiled.

The Clean compiler is a state-of-the-art compiler, well known for the efficient code it generates. Due to the many transformations performed by the compiler to obtain such efficient code, and the laziness of the language, it is in general near to impossible to relate the execution of an application to a specific part of the original source code. The advantage we have here is that, since we restrict ourselves to monadic contexts, we statically know their order of evaluation.

A particular challenge is how to relate run-time behaviour to the corresponding parts of static blueprints. The difficulty comes from run-time calculations and higher order functions. To do so, we modify the generated code by adding wrapper functions to the monadic applications. These wrappers tell which part of the original source code is being evaluated, so that it can be related to the correct part of the static blueprint.

With the dynamic blueprints we can show, at run-time, for any iTasks application, dynamic aspects such as: which tasks have been started, which are finished, which are running, how are they instantiated, what are the actual arguments, who is working on what, and which information is currently being produced by a specific task. The graphical representation of dynamic blueprints has to be modified at run-time to reflect the current program state.

In this paper we address the issues mentioned above and make the following contributions:

- We generalise the notion of blueprints to not only capture iTasks programs, but monadic programs in general. Using this new-found generalisation, we remove the hard connection between the Clean-Tonic compiler and iTasks, making Tonic a general solution.

- We show how static and dynamic blueprints are being made for the Task
monad. Furthermore we discuss how our approach can be used for any monad, such as e.g. the IO Monad.

- We explain what kind of code transformations are made by the compiler such that we are able to map run-time behaviour to static information generated from the source program.

- We explain how we created a Tonic Task which allows an end-user of any iTasks application to browse through the dynamic blueprints, and to inspect values of arguments and results of any task executed in the past or currently under evaluation.

- We explain that with a simple control-flow analysis and code transformation we can show the reachability of information (monads/task) in the blueprints. In this way we are able to show which future task can or cannot be executed given the current state of affairs.

- Tonic’s end users can now customize how tasks are rendered using the declarative Graphics.Scalable library [6].

The rest of this paper is structured as follows: Section 5.2 shows several examples of static blueprints, and Section 5.3 shows how these are made. Section 5.4 shows how we instantiate blueprints at run-time and incorporate run-time information in them. Finally, Section 5.6 discusses related work, and Section 5.7 discusses current challenges and concludes.

5.2 Static Blueprints: Examples

In this section we explain, with the help of a number of examples, what kind of static blueprints we generate from Clean source text[2]. In the introduction we already made clear that it is not a good idea to turn a complete Clean program into a graphical counterpart. First of all, Clean, much like Haskell, contains many language constructs. A pictorial representation isomorphic with the source code would only be huge and would not contribute to a better understanding of the code than the text of the source program itself. Secondly, there are technical obstacles that currently prevent us from showing all language features in a meaningful, graphical way. This is due to the fact that the Clean compiler generates highly efficient code, applying many transformations during compilation. Some of the original code is simply no longer available.

For all these reasons, we decided to restrict ourselves to generating graphical representations for certain top level abstractions and a limited number of language primitives. We want to capture the major structure of the application being defined. We do not want nor need to provide all details of the application. We therefore decided to focus on monads. Monads are a frequently used abstraction in functional programming. In Haskell, for example, the IO monad is the principal

---

[2] All blueprints in this paper are generated from the example programs.
way to perform side-effecting operations. As well as being useful, monads provide the ability to hide tedious book-keeping operations under the bind combinator, making code easier to read and reason about. In addition, the evaluation order of a sequentially composed monadic computations is well defined and strict for a subset of all monads, including the IO and iTasks’ Task monads. The laziness of the language does not provide problems here.

We distinguish two sets of monads: one for which we want to generate a blueprint, and one for which we don’t want to generate blueprints, but that may be part of a blueprint. We call the first set blueprint monads and the second contained monads. A blueprint monad is always a contained monad, but not the other way around. For iTasks, for example, the set of blueprint monads contains the Task monad, while the set of contained blueprints contains both the Task and the Maybe monad. To distinguish between the two sets of monads, we introduce two new type classes. We discuss the implementation and application of these classes in Section 5.3.

What makes generating blueprints challenging is that in any combinator definition, any Clean language construct may be used. As explained above, and further illustrated in the examples below, we limit ourselves to Clean language constructs which we are able to visualize in a meaningful manner, and hide those which are too complicated to visualize. We support if-blocks, case-blocks, pattern matching, let-blocks, recursion, higher-order functions and list definitions in the case that the number of list elements are statically known. For all other language constructs and cases we do not offer special graphical support in a blueprint. If we cannot graphically represent an expression, we pretty-print the original source code. Let’s have a look at some examples.

### 5.2.1 Static Blueprints of the I/O Monad

The example in Figure 5.1 shows a simple interactive program implemented in Clean’s IO monad. It asks the user to enter a number, confirms which number has been entered and then tells the user whether the number is prime or not. An example of its output is shown in Figure 5.2.

```
primeCheck :: IO ()
primeCheck = putStrLn "Enter number:"
  >>= getLine
  >>= \numStr -> putStrLn ("Entered: " ++ numStr)
  >>= if (isPrime (toInt numStr))
    (putStrLn ("Is prime: " ++ numStr))
    (putStrLn ("Isn’t prime: " ++ numStr))
```

Figure 5.1: IO implementation of the primeCheck example.

---

3Clean does not have do-notation, so binds are explicitly written out.
CHAPTER 5. STATIC AND DYNAMIC VISUALISATION OF MONADIC PROGRAMS

Enter number:
42
You have entered 42
42 is not prime

Figure 5.2: Example of IO performed by primeCheck

Figure 5.3 (a) shows the blueprint we generate for this program. The graphical representation of the top-level primeCheck computation acts as a container for the other graphical elements. Each IO function application is represented by its own function-application box. The applied function’s name is presented in bold on the top of the box, while its arguments are presented below it. Binds are represented by edges between two boxes. If the right-hand side of a bind is a lambda, the expression in the lambda is pretty-printed as edge-label.

(a)

(b)

Figure 5.3: Static blueprints of the primeCheck function implemented in both the IO and the Task monad.

5.2.2 Static Blueprints of the Task Monad

In this subsection we look at several example iTasks programs and the blueprints we generate for them. The goal of this subsection is to give an intuition for Tonic and its blueprints, while at the same time explaining the basics of iTasks.

5.2.2.1 Prime Number Checker

An iTasks version of the primeCheck program is shown in Figure 5.4 with its output shown in Figure 5.5 and its corresponding blueprint shown in Figure 5.3 (b). iTasks’ bind combinator automatically adds a “Continue” button to the user interface to progress to the right-hand side task. Since iTasks is shallowly embed-
5.2. STATIC BLUEPRINTS: EXAMPLES

ded in Clean, all Clean language features can be used to construct iTasks programs. Some of these, e.g. conditionals, we also want to include in the blueprints. Tasks are defined as functions with monadic result type (Task a) for some a. Sequential task composition is accomplished with the monadic bind combinator. 

`enterInformation` and `viewInformation` are examples of basic predefined editor tasks, which generate a web-based graphical user interface for a given type using generic programming techniques. The former editor allows the user to enter data using generically generated web forms, while the latter editor renders a textual read-only representation of the data.

```haskell
primeCheck :: Task Int
primeCheck
  =  enterInformation "Enter_number" []
  >>= \num -> viewInformation "Entered:" [] num
  >>= \(isPrime num)
    (viewInformation "Is_prime:" [] num)
    (viewInformation "Isn't_prime:" [] num)
```

Figure 5.4: iTasks implementation of the `primeCheck` example.

![Web forms generated by `primeCheck`](image)

Despite the fact that the previous two programs are defined in different monads, their blueprints are similar, since they share the common abstraction level of a monad.
5.2.2.2  Step

User definable buttons can be created by using the step combinator (>>*), shown in Figure 5.6 (with its output in Figure 5.10(a) and blueprint in Figure 5.7). The step’s left-hand side is a task that is executed first, while its right-hand side is a list of conditions paired with a follow-up task. If a condition is met, the corresponding follow-up task is executed.

```haskell
palindrome :: Task String
palindrome
  = enterInformation "Enter a palindrome" []
  >>=*
    [ OnAction (Action "Ok" [])
    (ifValue isPalindrome (\palindrome -> return palindrome)) ]
where
  isPalindrome :: String -> Bool
  isPalindrome s = let
                    s' = [toLower c | c <- s | c /= '\']
                    in
                    s' == reverse s'

Figure 5.6: iTasks implementation of the palindrome example.
```

In this example one such condition is provided in the form of the OnAction constructor, which causes a button to be rendered in the left-hand side task’s user interface. OnAction takes two arguments: an action, which describes the button’s text and a list of button meta-data, and a continuation to proceed to the next task once the corresponding button is pressed. The continuation is of type (TaskValue a) -> Maybe (Task b). If the continuation returns Nothing the button is disabled. If it returns Just, the button is enabled and pressing it will progress the workflow to the inner Task b. Several convenience functions are available to write these continuation functions. In our example, the ifValue function is used. It takes a predicate isPalindrome over the left hand-side task’s value, enabling the corresponding button only if the predicate returns True.

The step combinator is strictly more powerful than the bind combinator. In fact, the bind combinator is implemented in terms of the step combinator, as shown in Figure 5.8.

If the left-hand side task has a value, the “Continue” button is enabled. Additionally, if the left-hand side task has a stable value, i.e., if the value is guaranteed
5.2. STATe BLUEprints: EXAMPLES

\[
(\gg\gg) :: (\mathrm{Task} \ a) \ (a \to \mathrm{Task} \ b) \to \mathrm{Task} \ b \mid \mathrm{iTask} \ a \& \mathrm{iTask} \ b
\]
\[
(\gg\gg) \ \text{taska taskbf} = \text{taska} \gg\gg \{ \text{OnAction (Action "Continue" []) (hasValue taskbf)} \}
\]
\[
, \text{OnValue (ifStable taskbf)} \}
\]

Figure 5.8: Implementation of the bind combinator.

to never change again, it also proceeds to the right-hand side task.

5.2.2.3 Recursion and Higher-Order Tasks

Tasks can be passed as argument to other tasks: one can define higher-order tasks. Other functional concepts translate to TOP as well, such as recursive tasks. Both of these concepts are demonstrated in Figure 5.9 in the \texttt{add1by1} task. Its blueprint is shown in Figure 5.11 One new graphical element is that of the \texttt{let} binding; they are rendered as sign-posts.

Here we see that \texttt{add1by1} has two arguments; a higher order task, called \texttt{task}, of type \((\mathrm{Task} \ a)\), and an accumulator \texttt{listSoFar} of type \([a]\). On demand of the end user, \texttt{add1by1} recursively evaluates the higher order task and accumulates the results. When finished, the accumulator is yielded as result. Notice that \texttt{add1by1} is not polymorphic in \(a\), but overloaded. In Clean, context restrictions are specified at the end of a type definition \((\mid \mathrm{iTask} \ a)\). This context restriction is synonymous for several generic functions that take care of the type driven rendering of GUIs and the communication between the web server and the client (i.e. the web browser). This can automatically be derived by the Clean compiler for any first order type. Context information is considered to be too much detail to mention in a blueprint and is therefore left out in the types shown in the blueprint.

\[
\texttt{add1by1} :: (\mathrm{Task} \ a) \ [a] \to \mathrm{Task} \ [a] \mid \mathrm{iTask} \ a
\]
\[
\texttt{add1by1} \ \text{task listSoFar}
\]
\[
= \text{task}
\]
\[
\gg\gg \ \text{\{elem -> let newList = [elem : listSoFar] in}
\]
\[
\text{viewInformation } "\text{New list:;}" \ [] \ \text{newList}
\]
\[
\gg\gg \ \{ \ \text{OnAction (Action } "\text{Add another}" [] )
\]
\[
\ \{ \ \text{always (add1by1 task newList))}
\]
\[
\ \{ \ \text{OnAction (Action } "\text{Done}" [] )
\]
\[
\ \{ \ \text{ifValue hasManyElems (\xs -> return \xs)) } \}
\]
\[
\text{where}
\]
\[
\text{hasManyElems :: [a] -> Bool}
\]
\[
\text{hasManyElems} \ \text{xs} = \text{length} \ \text{xs} > 1
\]
\[
\texttt{addPalindromes} :: \mathrm{Task} \ [\text{String}]
\]
\[
\texttt{addPalindromes} = \texttt{add1by1 palindrome} []
\]

Figure 5.9: Implementation of the \texttt{add1by1} task.
The task \texttt{add1by1} also has a step function. In this particular example, we can see that step functions are rendered differently from binds. Each condition in the step’s right-hand side’s list is rendered in its own branch. Continuation convenience functions as found in iTasks’ standard libraries are rendered in a special way as well. Here, the \texttt{hasManyElems} predicated is rendered as a diamond, implying that this condition should be met before the work-flow can continue. The action is rendered as well, together with a small figure showing that it relates to a user action. It is possible to customize the way blueprints are rendered (see Section 5.4.2).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/chapter5/add1by1_palindrome.png}
\caption{GUIs when applying \texttt{add1by1} to the \texttt{palindrome} task}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/chapter5/add1by1_blueprint.png}
\caption{Static blueprint of \texttt{add1by1} with a higher-order task and recursion}
\end{figure}

The higher order task \texttt{task} is executed first. Statically we only know the type of \texttt{task}, but we do not know what its concrete value will be. For this we use a dashed frame. We do know that the task yields a value of proper type \texttt{a}. This value can be added to the accumulator (when the “Add another” button is pressed), after which \texttt{add1by1} recursively calls itself. Alternatively, the task can
be terminated by pressing “Done”, but this option can only be chosen when at least two values are collected in the list.

5.2.2.4 Parallel Tasks

iTasks allows several tasks to be executed in parallel. In the parallelChat example, shown in Figure 5.13, the user of the task (currentUser) starts a chat by first selecting \( n \) friends to chat with from a list of administrated users. Next, \( n + 1 \) makeChat tasks are started in parallel using the library combinator allTasks. This function expects a lists of tasks to be executed in parallel and ends when all its tasks are ended. parallelChat’s output is shown in Figure 5.12 and its blueprint is shown in Figure 5.14.

![Parallel Chat Example](image)

Figure 5.12: parallelChat program execution

Each makeChat task enables user \( i \) to have a chat with the others via a shared data source chatBox of type Shared [String]. Shared Data Sources (SDS) allow tasks to share information. The shared list used here contains as many strings as there are chatting users, where the \( i \)-th element of the list represents the information typed in by the \( i \)-th chat user. In iTasks, shared data structures are maintained automatically. Whenever someone is changing the content of a shared data structure, any task that is looking at its structure is informed and updated automatically. This notification system works for any first order data type, not just shared strings of text. In this example, chatting users automatically see what is written by someone else. Chat users can only update their part of the shared structure. In UpdateSharedInformation the \( i \)-th element is selected (selectChat) to be updated in the function defined in UpdateWith while in ViewSharedInformation the other elements are selected (dropChat) in ViewWith and shown read-only.

In this particular example it is statically undecidable how many parallel task there will be, since it depends on the number of chosen friends. We will later see that at run-time we can in fact show these tasks in a dynamic blueprint, and see who is chatting with whom and inspect what they are chatting about. In general, one can statically not deduce how many elements are contained in a list. In a static blueprint we therefore only show the elements of a list when it statically contains a fixed number of elements and it is not generated by a list-comprehension or
CHAPTER 5. STATIC AND DYNAMIC VISUALISATION OF MONADIC PROGRAMS

```
parallelChat :: Task [[String]]
parallelChat
  =  get currentUser
  >>= \me -> enterMultipleChoiceWithShared "Select_friends" users
  >>= \friends -> let users = [me : friends] in
         withShared (repeatn (length users) "")
         (\chatBox -> allTasks (chatTasks users chatBox))

where
  chatTasks :: [User] (Shared [String]) -> [Task [String]]
  chatTasks users chatBox = [ chatTask user i users chatBox
                             \ i <- [0..] & user <- users ]
  chatTask :: User Int [User] (Shared [String]) -> Task [String]
  chatTask user i users chatBox = user : makeChat i users chatBox

makeChat :: Int [User] (Shared [String]) -> Task [String]
makeChat i users chatBox
  =  updateSharedInformation [selectChat i]
      (users !! i +++> "is chatting:") chatBox
  ||- viewSharedInformation [dropChat i] "with:
where
  selectChat i = UpdateWith (\chatBox -> chatBox!!i)
  (\chatBox chat -> (updateAt i chat chatBox))
  dropChat i = ViewWith (\chatBox ->
                          [ user +++> "s: " +++> chat
                          \ (user, chat) <- removeAt i (zip2 users chatBox) ])```

Figure 5.13: Implementation of the `parallelChat` task.

other list-producing expression. This holds for the list of step continuations used in the `add1by1` task, but it does not hold for the list of chat tasks used in the `parallelChat` task. Since lists are the most frequently used data structure in a functional language, several convenient language constructs are offered in Clean to handle them, such as dot-dot notation and list comprehensions.

5.3 Building Static Blueprints

Figure 5.15 shows the architecture of the modified Clean-Tonic compiler. In addition to the code the compiler normally generates (Intel, Arm and JavaScript), it now also generates a file containing blueprint information for each Tonic-enabled Clean module. This information can be read in by a tool called the Tonic Viewer. The viewer is implemented in iTasks itself and can render blueprints in any HTML5 compatible browser.

As explained in the previous section, not all functions are automatically turned
5.3. BUILDING STATIC BLUEPRINTS

into a blueprint, only those with a blueprint monadic return type. Likewise, not all monadic function applications are turned into blueprint nodes, only those with a contained monadic return type. To differentiate between these sets of monads we introduce two type classes, **Blueprint** and **Contained**, both shown in Figure 5.16.

```haskell
class Contained m | Monad m
class Blueprint m | Contained m
```

Whenever a programmer provides an instance of the **Blueprint** class for a certain type, a blueprint is generated by the compiler for every function which returns a monad of that type. Whenever an instance of the class **Contained** is provided, the application of the function in a blueprint is treated special. Any type with a **Blueprint** instance also requires a **Contained** instance, which is enforced by the former class’ context restriction. Not all modules are considered for blueprint generation. Only modules that explicitly import the Tonic framework are searched for top-level blueprints. This approach offers a course-grained control over the
blueprint generation process. For example, none of the iTasks core modules import the Tonic framework, so no core tasks are turned to blueprint.

All blueprints are built from a small and general core language, shown in Figure 5.17. At compile-time, we generate blueprints per Clean module (TModule). For every function of a blueprint monad we create a TFun record. This record contains meta-information, such as the comments, module name, function name, the function definition’s line number, its result-type, the argument names and types and the function body. Every type or expression is represented by the TExpr data type.

```
:: ModuleName ::= String
:: FuncName ::= String
:: Pattern ::= TExpr
:: TypeName ::= String
:: PExpr ::= String
:: ExprId ::= [Int]
:: VarName ::= String
:: VarPtr ::= Int

:: TModule = { tm_name :: ModuleName , tm_funcs :: Map FuncName TFun }

:: TFun = { tf_comments :: String ,
            tf_module :: ModuleName ,
            tf_name :: FuncName ,
            tf_icllLineNo :: Int ,
            tf_resty :: TExpr ,
            tf_args :: [(TExpr, TExpr)] ,
            tf_body :: TExpr }

:: TExpr = TVar ExprId PExpr VarPtr |
            TPEExpr PExpr |
            TMApp ExprId (Maybe TypeName) ModuleName FuncName [TExpr] TPriority (Maybe VarPtr) |
            TFApp ExprId FuncName [TExpr] TPriority |
            TLam [TExpr] TExpr |
            TLet [(Pattern, TExpr)] TExpr |
            TIf ExprId TExpr TExpr TExpr |
            TCase ExprId TExpr [(Pattern, TExpr)]

:: TPriority = TPrio TAssoc Int | TNoPrio
:: TAssoc = TLeftAssoc | TRightAssoc | TNoAssoc
```

Figure 5.17: Algebraic data type definitions for blueprints.

TExpr contains the usual suspects for a small core language, such as variables, literals, lambdas, lets and cases. Function application, however, is represented
5.4 DYNAMIC BLUEPRINTS

by two distinct constructors: \texttt{TMApp} and \texttt{TFApp}. The former represents function application of all contained monads (hence the M), the latter all other function applications. Several constructors contain additional meta-data. An \texttt{ExprId}, found in the \texttt{TVar}, \texttt{TMApp}, \texttt{TFApp}, \texttt{TIf}, and \texttt{TCase} constructors, uniquely identifies those expressions in a blueprint. This turns out to be very useful later on when we will make blueprints show dynamic behaviour (Section 5.4). \texttt{TMApp} also contains the type of the monad (if the function is monomorphic in its monadic return type) and the name of the module in which the function being applied is defined. This is to disambiguate functions with the same name. In addition to the function’s arguments and priority, it has an optional \texttt{VarPtr} in case the function being applied is variable.

To get an intuition of what a static blueprint looks like in code, lets look at a blueprint for the iTasks \texttt{primeCheck} example (Section 5.2.2.1). The blueprint code is shown in Figure 5.18. Note how the unique node numbering allows for a deterministic lookup of a node’s parents and siblings. Despite the presence of meta-data such as the unique node identifiers and the unique variable identifiers, the blueprint remains compact, making it suitable for transmission over a network.

5.4 Dynamic Blueprints

A static blueprint gives a graphical view of how monadic functions are used in the source code. Now we want to be able to trace and inspect the execution of the resulting application, making use of these blueprints. Although the monadic parts of the program may be just a small part of the source code, they are an important part and they commonly form the backbone of the architecture of the application. If we can follow their execution and see how their corresponding blueprints are being applied, we will already have a good impression of the run-time behaviour of the application. We want to show which monadic computation is currently being executed, how far along the program’s flow we currently are, the current value for a given argument or variable, the result of a completed computation, and which program branches will be taken in the future. Before delving into the technical challenges associated with addressing these requirements, we look at our previous examples and how their static blueprints are used at run-time.

When a function with a \texttt{Blueprint}-monadic type is applied, we make an instantiation (a copy) of its corresponding static blueprint, creating a \textit{dynamic blueprint}. On top of it we can show who is calling it, we can inspect its actual arguments, and visualize the progress in the flow when the body is being executed. The Tonic viewer can show and inspect these dynamic blueprints. Notice that the Tonic viewer can show the blueprints in real-time, i.e. when the application is being executed. The Tonic viewer also allows inspecting the past, and it can sometimes predict the future. Since we output blueprints in SVG, most blueprints in this section are imported SVG files. In some cases, however, we use a screen-shot instead. This is so we can include other DOM elements, such as the Tonic viewer’s value inspector windows, as well.
CHAPTER 5. STATIC AND DYNAMIC VISUALISATION OF MONADIC PROGRAMS

5.4.1 Dynamic Blueprints of the Task Monad

In this section we will look at how we augment the blueprints of the previous examples with run-time information.

5.4.1.1 Prime Number Checker

In the primeCheck example we saw sequential composition using a bind combinator. Since bind determines the order in which computations are executed, it is a great place for us to track progress in a program’s flow. Figure 5.19 shows the dynamic blueprints for the primeCheck iTasks program as it is executed.

When the program starts and the user is presented with the input field, its corresponding blueprint instance is that of Figure 5.19(a). Immediately the blueprint
is different from its static incarnation in several ways. A pair of numbers is added in the top bar, next to the task name. This is the task ID, uniquely identifying this task instance within the iTasks run-time system. Next to it is the image of a person, together with the name of the person that is currently executing this particular task instance. Going to the lower half of the blueprint, we see that the upper area of the task-application node is coloured green. Green means that the task is currently actively being worked on. We also say that the enterNumber node is active. Additionally, the task ID of the enterNumber task instance is added to the blueprint and positioned next to the task name.
Next to each node, a square is drawn. Clicking on this square allows us to inspect the task’s value in real-time. Its colour also indicates the stability of the task’s value. In Figure 5.19(a), there is no value yet, hence the square is white. This is confirmed by a pop-up window when we click the white square. However, as soon as a number is entered by the end user in the editor’s text field, or whenever the number is changed, the current input is directly shown in the inspection window (Figure 5.19(b)).

On the right side of the blueprint there is a diamond-shaped conditional node, followed by two viewInformation nodes, which now have green borders. These border colours tell us something about the future, in particular which program branch might be taken. Since the program has only just started, all branches might still be reached. However, when we enter the number 42 to the enterNumber task’s text field – which is not a prime number – we can already predict that the True branch will not be reached. This is represented by red borders, as seen in Figure 5.19(b). If we would change the number in the box to, e.g., 7 the tasks in the False branch would receive a red border instead. We call this feature dynamic branch prediction. Once the user has entered a number and has pressed “Continue”, the work-flow progresses to the second task and the blueprint instance is updated accordingly (Figure 5.19(c)). The first node is no longer highlighted. Instead, it is frozen and given a blue colour. A frozen blueprint node for a given task instance will not change again. Additionally, the edge between the first and second node is now coloured green. For edges, green does not indicate activity, but the stability of the previous task’s value. A green edge means an unstable value, while a blue edges means a stable value. In iTasks, tasks may have a stable or unstable value, or even no value at all. It reflects the behaviour of an end user filling in a form. The form may be empty to start with or some information may be entered which can be changed over time. Once values are stable they can no longer change over time. When the “Continue” button is pressed again, we reach the False branch, as predicated earlier. Since the True branch is no longer reachable, its nodes now get a grey header (Figure 5.19(d)).

5.4.1.2 Recursion and Higher-Order Tasks

Yet other dynamic behaviour is found in the blueprints of add1by1 (as applied in addPalindromes), in which we have to deal with a task as argument, a step combinator, and recursion. Its dynamic blueprints are shown in Figure 5.20 and Figure 5.21. Notice that the task variable is now replaced by a task-application node containing the name of the palindrome task (Figure 5.20(a)). When a valid palindrome has been entered, the workflow continues to the viewInformation task. The step combinator at that point presents the user with two buttons: “Add another” and “Done”. The former can always be pressed, whereas the latter is only enabled when at least two values are accumulated in newList. Since we only have one palindrome so far, only the “Add another” button is enabled. This is reflected in the blueprint (Figure 5.20(b)). Recursion is simply yet another task-application node (Figure 5.20(c)). Entering the recursion creates a new blueprint instance for the add1by1 task in which another palindrome task is executed (Fig-
When the user submits another valid palindrome, we encounter `viewInformation` again. This time, however, the “Done” button is enabled, because the `hasManyElems` predicate holds. (Figure 5.21(e)). Pressing “Done” finishes the `add1by1` task and returns the list of palindromes (Figure 5.21(f)).

```latex
\begin{figure}
\centering
\begin{tikzpicture}
\node (task) [Task] {Task \{a\}};
\node (listSoFar) [Task a] at (0,0) {listSoFar \{a\}};
\node (palindrome) [Task] at (-3,0) {palindrome \{1-35\}};
\node (elem) [Task] at (-4,-1) {elem};
\node (newList) [Task] at (-5,-2) {newList \{elem : listSoFar\}};
\node (viewInformation) [Task] at (-6,-3) {\texttt{viewInformation}};
\node (add1by1) [Task] at (-7,-4) {\texttt{add1by1}};
\node (hasManyElems) [Task] at (-8,-5) {\texttt{hasManyElems}};
\node (done) [Task] at (-9,-6) {\texttt{Done}};
\node (xs) [Task] at (-10,-6) {xs};
\draw (task) -- (listSoFar);
\draw (palindrome) -- node[above] {\texttt{viewInformation}} (newList);
\draw (newList) -- node[above] {\texttt{Has Many Elements}} (done);
\draw (done) -- (xs);
\end{tikzpicture}
\caption{Dynamic blueprints of \texttt{add1by1}}
\end{figure}
```

### 5.4.1.3 Parallel Chat Tasks

In the `parallelChat` example we saw that the function application of `chatTasks` can only be pretty printed. There are two reasons for this: 1) we don’t have a `InstanceOf` instance for lists (for the sake of this example), and 2) `chatTasks` is a function application. We cannot compute any kind of function statically. At runtime, however, we would like to know which tasks are being executed in parallel, so we need to replace the pretty-printed expression with a list of task-application nodes dynamically. We can see how Tonic deals with this situation in Figure 5.22.

Figure 5.22(a) shows that we select two friends to chat with: Bob and Carol. Next, the `parallelChat` task delegates three chat tasks: one to the current user, Alice, and one to each of her friends. Since the `chatTasks` function application is now evaluated, we can substitute a list of task application nodes for the pretty-printed expression. Each of the nodes contain the parallel task’s name and task
CHAPTER 5. STATIC AND DYNAMIC VISUALISATION OF MONADIC PROGRAMS

Figure 5.21: Dynamic blueprints of `add1by1`, recursive call.

Figure 5.22: Dynamic blueprints for the parallel chat example
5.4. DYNAMIC BLUEPRINTS

ID. For each of these nodes a corresponding blueprint instance is created, which can be inspected as well (Figure 5.22(a)).

5.4.2 **Tonic Architecture**

To enable such dynamic features, we need to make a connection between the static blueprints and the program’s run-time. With this connection, we can pass additional information from the original program to the Tonic run-time system. This is similar to standard tracing and debugging tools. Connecting blueprints and a program’s run-time is done by extending the `Contained` and `Blueprint` classes with *wrapper functions* that we apply to the original program at compile-time. These wrapper functions are executed at the same time as the program’s original functions. It is up to the programmer to provide sensible instances for these classes. We have already provided instances for both classes for the `Task` type that can be used in any iTasks program. Section 5.4.4 shows how these classes are defined for iTasks.

Figure 5.23 shows the architecture of a Tonic-enabled iTasks application. Tonic maintains a central SDS with run-time information. When a wrapper function is applied, it writes additional information to this SDS, allowing us to track the program’s progress and inspect its values. The specifics of what data the wrappers contain are discussed later in this section. Writing to the share triggers an update that refreshes the dynamic blueprint in the Tonic viewer.

![Architecture of integrated Tonic viewer](image)

The Tonic viewer is written in iTasks itself and is therefore yet another task. Using the Tonic viewer in an iTasks application requires the programmer to make sure the viewer task is reachable by the program’s end-user. Having the viewer built into the application that is going to be visualized has certain advantages. In iTasks’ particular case, this allows us to easily inspect nearly all function arguments and task values using iTasks’ own generic editors. This even works for complex types. Section 5.4.5 shows how this integrated viewer is used by an end-user. Section 5.5.1 talks about a solution that does not require the viewer to be integrated with the original application.
When applying the viewer-task, the programmer can optionally provide additional render functions with which the rendering of individual function-application nodes can be customized. The programmer can use our fully declarative SVG library to define alternative visualizations. This library is a general-purpose tool to draw arbitrary vector images. As such, the programmer is not constrained in what the custom rendering looks like.

5.4.2.1 Contained Monads

The Contained class is what identifies interesting function applications. It is therefore the right place to gather more information about the functions being applied. For example, which function is being applied? To which blueprint node does this function application correspond? How does the value of the underlying function application influence the program’s workflow? We extend the Contained class with only one function: wrapFunApp, as shown in Figure 5.24.

```
class Contained m | Monad m where
    wrapFunApp :: ModuleName FuncName ExprId [(ExprId, a -> Int)]
                    (m a) -> m a | iTask a
```

Figure 5.24: Complete definition of the Contained type class.

It is here that the Tonic system is notified of the execution of individual computations, where the current value of these computations is inspected, where blueprints are updated dynamically, etcetera. wrapFunApp takes five arguments, the first two of which are the module and function name of the function being applied. The third argument is an ExprId, which together with the module and function name of the function application’s context (obtained via the Blueprint class and passed through by iTasks; see Section 5.4.2.2), uniquely identifies this function application. The same ExprId is also found in the blueprint of the parent function, allowing us to relate run-time execution to the static blueprint. The fourth argument allows us to do dynamic branch prediction. It is a list of pairs, the first element of which is the ExprId that refers to the case block of which we want to predict its future. The second element is a function that, given the value of type a of the wrapped task (Task a), gives the index number of the branch that will be chosen, should that value be used. Before discussing how the Contained class is used we need to understand how dynamic branch prediction works.

Tonic’s dynamic branch prediction feature utilizes the fact that Tonic is implemented as a compiler pass in the Clean compiler. During the Tonic pass, we copy case blocks and lift them to a newly generated function. We transform the right-hand side of the individual cases and return the index of the branch as integer. We call this entire procedure case lifting. By applying this fresh function, we known, using the original case expression, the index of the branch that will be taken, should that expression be evaluated with an identical context. Definition 5.4.1 formalizes this process.
Definition 5.4.1. Case lifting transformation. Given a case expression

\[
\text{case } f \ x_1 \ldots x_i \text{ of }
\]

\[
p_1 \to e_1 \\
\ldots \\
p_j \to e_j
\]

Generate a fresh function

\[
dbpf :: a_1 \ldots a_i \to \text{Int} \\
dbpf \ x_1 \ldots x_i = \text{case } f \ x_1 \ldots x_i \text{ of }
\]

\[
p_1 \to 1 \\
\ldots \\
p_j \to j
\]

As mentioned earlier, wrappers are not always applied. In particular, it might be necessary to forego wrapping certain expressions when they are an argument to another function. Consider again the addby1 example. Should we wrap the recursive call as well as the task variable, the recursive instance would effectively have two wrappers around task due to laziness. When task is evaluated, both wrappers would be evaluated as well, polluting Tonic’s run-time state with wrong data. Still, in some cases we do want to wrap higher-order arguments. The most prominent case for this is the bind combinator. An iTasks-specific case are the parallel combinators. They are rendered as a container within which we want to keep following the workflow’s progress. We need the wrappers to do so. To support this case, we only wrap function arguments when the function itself comes from a module that does not enable Tonic. In addition, a function-level pragma, either TONIC\_CONTEXT or TONIC\_NO\_CONTEXT, can be provided. When the former pragma is used, the function’s arguments are wrapped. With the latter, they are not. The pragmas override the default module-based wrapping behaviour and allow custom domain-specific behaviour to be specified instead. Definition 5.4.2 formalizes the transformations the Tonic compiler applies to utilize the Contained class.

Definition 5.4.2. Contained transformation. For all functions \( f :: \alpha_1 \ldots \alpha_i \to m \alpha_n \) in module \( M \), applied as \( fe_1 \ldots e_i \), and for which we there is an instance of class Contained \( m \):

\[
[f e_1 \ldots e_i >>= \lambda v \to e_j]
\]

iff module \( m \) does not enable Tonic or \( f \) has TONIC\_CONTEXT

\[\Rightarrow\]

wrapFunApp "M" "f" exprId(f) dbpC(v, e_j) (f[e_1]...[e_i])

\[\Rightarrow\]

\[
[f e_1 \ldots e_i >>= \lambda v \to e_j]
\]

otherwise

\[\Rightarrow\]

wrapFunApp "M" "f" exprId(f) dbpC(v, e_j) (fe_1...e_i)

\[\Rightarrow\]

\[
\lambda x \to [e_j]
\]
[\[ eg_1 ... eg_i \] ]

\[
\Rightarrow \text{wrapFunApp "M" "f" exprId(f) [\[ (f[eg_1] ... [eg_i]) \] ]}
\]

[\[ eg_1 ... eg_i \] ]
otherwise
\[
\Rightarrow \text{wrapFunApp "M" "f" exprId(f) [\[ (eg_1 ... eg_i) \] ]}
\]
\[
\Rightarrow [\[ e \] ]
\]
\[
\Rightarrow e
\]

Two additional functions are used during this transformation: \textit{exprId} and \textit{dbpC}. \textit{exprId}(f) returns a unique identifier for the application of \textit{f} to its arguments. \textit{dbpC} enables dynamic branch prediction for contained monads as follows. For all lifted case functions \textit{dbpf}_k x_1 ... x_i, x_{i+1} from \textit{e}_j, if \textit{v} = x_{i+1} and \textit{x}_1 ... \textit{x}_i are bound, then \[(\text{caseExprId(dbpf}_1, \text{dbpf}_1, x_1 ... x_i), ..., (\text{caseExprId(dbpf}_n, \text{dbpf}_n, x_1 ... x_i)]. Here, \textit{caseExprId} returns the unique identifier for the original case expression that was used to create \textit{dbpf}. Implementing dynamic branch prediction in a bind is possible because the monad right-identity law guarantees that for a bind expression \textit{e}_1 \texttt{>>= \lambda x \rightarrow e}_2, \textit{x} will always bind \textit{e}_1’s result value.

5.4.2.2 Blueprint Monads

The \textbf{Blueprint} class already allows us to identify functions for which to generate a blueprint. This class is therefore well suited to capture some meta data for blueprint functions that would otherwise be lost at run-time. We extend the \textbf{Blueprint} class with two functions: \texttt{wrapFunBody} and \texttt{wrapFunArg}, as shown in Figure 5.25.

\begin{verbatim}
class Blueprint m | Contained m where
  wrapFunBody :: ModuleName FuncName [(VarName, m ())] [(ExprId, Int)]
    (m a) -> m a | iTask a
  wrapFunArg :: VarName a -> m () | iTask a

Figure 5.25: Complete definition of the Blueprint type class.
\end{verbatim}

The \texttt{wrapFunBody} function is statically applied to the body of a blueprint function. It has several goals: to make the blueprint function’s module and function name available at run-time, to provide a way to inspect the blueprint function’s arguments, and to do future branch prediction based on the function’s arguments. The \texttt{wrapFunArg} function is used in the third argument of \texttt{wrapFunBody}. It is statically applied to all function arguments to enable their inspection at run-time. In general, the compiler applies the following transformation rule:
Definition 5.4.3. Blueprint transformation. For all function definitions \( f :: \alpha_1 \ldots \alpha_i \rightarrow m \alpha_m \) in module \( M \), for which we have an instance of class Blueprint \( m \):

\[
[f x_1 \ldots x_i = e] \\
\Rightarrow f x_1 \ldots x_i = \text{wrapFunBody} \quad "M" \quad "f" \\
\quad [\text{"}x_1\text{"}, \text{wrapFunArg} \; x_1] \\
\quad \ldots \\
\quad [\text{"}x_i\text{"}, \text{wrapFunArg} \; x_i] \\
\quad \text{dbpB}(x_1 \ldots x_i, e) \\
\quad [e]
\]

\( \text{dbpB} \) works subtly different from \( \text{dbpC} \). Rather than being associated with a variable bound by a lambda in a bind, it works on the function’s arguments, which are all bound as soon as the blueprint is instantiated.

The iTask constraint on the Contained and Blueprint class members is used extensively in iTasks. Unfortunately, due to limitations in Clean’s type system, we are currently forced to include this context restriction in our two classes, even though they might be instantiated for monads that have nothing to do with iTasks. We will come back to this limitation in Section 5.7.

5.4.3 Tonic Wrappers in Action

Applying all transformations to the primeCheck example transforms it to the code in Figure 5.26 (manually simplified for readability). Module names passed to the wrappers are fully qualified. The lists of numbers are the unique expression identifiers from the exprId function. The \_f\_case function is an instance of the \text{dbpf} function.

```haskell
primeCheck :: Task String
primeCheck = wrapFunBody "itasks_tonic_examples" "primeCheck" [] []
    wrapFunApp "itasks_tonic_examples" "enterNumber" [0, 0]
        [([0, 1, 0, 1, 0], \_f\_case\_4566316320)] enterNumber
    >>= \num -> let numStr = toString num in
        wrapFunApp "iTasks.API.Common.InteractionTasks" "viewInformation"
            [0, 1, 0, 0] [] (viewInformation "Entered:" [] numStr)
    >>= \| if (isPrime num)
        (wrapFunApp "iTasks.API.Common.InteractionTasks"
            "viewInformation" [0, 1, 0, 1, 0, 0] []
            (viewInformation "Is_prime:" [] numStr)
        (wrapFunApp "iTasks.API.Common.InteractionTasks"
            "viewInformation" [0, 1, 0, 1, 0, 1] []
            (viewInformation "Isn’t_prime:" [] numStr)
```

Figure 5.26: Example of the transformed primeCheck program.
5.4.4 Dynamic Blueprints in iTasks

To demonstrate how these wrappers can be used, we show their concrete implementation for iTasks. A task in iTasks is represented by the Task type (Figure 5.27).

```
:: Task a = Task (TaskAdministration TonicAdministration *IWorld
  -> *(TaskResult a, *IWorld))
```

Figure 5.27: The Task type.

A task is implemented as a continuation which takes some internal task-administration and some Tonic administration and passes it down the continuation. It chains a unique IWorld through the continuation, which allows interaction with SDSs and provides general IO capabilities, amongst other things. The continuation produces a TaskResult, which, amongst other things, contains the task’s result value.

The wrappers we place in the code unpack the continuation from a Task constructor and use it to define a new task, as shown in Figure 5.28. In the case of the Blueprint class, the wrapper’s job is to create a new blueprint instance for the task that is being started (line 7), while in the case of Contained, the wrapper’s job is to update the blueprint instance in which the task-application takes place. In that case, a blueprint instance already exists and just needs to be loaded from Tonic’s internal administration (line 23). Both wrapper classes perform similar operations: the relevant blueprint instance is loaded and updated, after which it is stored again, triggering a redraw event. One of the differences between the two classes is in when the original continuation is executed. In wrapFunBody, it is the last thing the wrapper does (line 12). In wrapFunApp, the original continuation is executed half-way in the wrapper (line 27). After executing the original continuation, the blueprint instance is loaded again, since it may have been updated by other wrappers in the mean time. Another thing both class instances have in common is that they both do future branch prediction (lines 9 and 29).

5.4.5 The Integrated Tonic Viewer

The integrated Tonic viewer is written in iTasks, for iTasks. To use the viewer for viewing dynamic blueprints, the programmer has to import it in the application that needs to be inspected, thereby including it as part of the original program. Implementing the viewer in iTasks is advantageous, because it allows us to develop it quickly and to leverage our Graphics.Scalable library for drawing the blueprints. Another advantage is that we can easily integrate SDSs in our iTasks programs and refresh the correct tasks when the SDSs are changed. Tonic uses SDSs to store its blueprint instances and run-time meta-data. Any time an instance or its meta-data is updated, the Tonic viewer gets a signal and is able to redraw the corresponding blueprint. Yet another advantage of implementing Tonic in iTasks, for iTasks, is that we have iTasks’ generic machinery at our disposal with
which we can easily inspect the data that is being passed around in the program. With the generic instances derived for the data, inspecting the data has become equivalent to applying a `viewInformation` editor.

Figure 5.29 shows a screenshot of the integrated dynamic Tonic viewer. Both the original application and the Tonic viewer run in the browser. The latter has its own URL. The viewer offers two modes, represented by two tabs: a mode with which one can view static blueprints and a mode with which one can view dynamic blueprints. When viewing static blueprints, one can browse through all static blueprints for that particular application. Viewing static blueprints is useful when using Tonic as a means of communication with stakeholders. In this mode, Tonic is akin to static UML or BPMN viewers. When viewing dynamic blueprints,
the user is presented with a list of active tasks, i.e. a list of blueprint instances. Any task can be selected in order to view its instance. Meta-data such as a task’s unique identifier, start time, modification time, optional end time, and which is working on it is also presented.

![Figure 5.29: A screenshot of the integrated Tonic viewer](image)

Below the list of blueprint instances is a large space for rendering the dynamic blueprints. Exactly what is visualized can be customized in a settings panel on the right side of the screen. For example, the “Unfold depth” slider determines how many levels of child tasks are shown together with the selected task. In the screenshot in Figure 5.29 we have selected the “Show all child tasks” option, so all child tasks to the currently selected task are shown recursively. Another option is “Show task value”. This opens a floating window in which one can see the task value of a selected task. Tasks can be selected for this purpose by clicking the small square on the right-hand side of the task-application node. Other features of the viewer include viewing the doc-block comments associated with a particular function, showing all finished blueprints, and a compact-mode, in which task-application arguments are not rendered.

One of the challenges in making a viewer for dynamic blueprints is designing a way to navigate through all active blueprints. Even in a small application such as this `add1by1` example, the number of blueprint instances quickly rises. To manage a potentially large number of blueprint instances, the Tonic viewer offers a means to filter the list of dynamic blueprints. This is done in the “Filter query” panel on the right side of the screen. Active blueprints can be filtered by substring matching.
on any of the columns in the blueprint list. Complex filters can be constructed using conjunction and disjunction operators.

5.5 Blueprints for All

In the previous sections we have looked in great detail at the way Tonic is implemented for iTasks programs. One of the claims we have made earlier is that we now support blueprints for any monad. To solidify this claim, we shall look at an example of dynamic blueprints of a program in the \texttt{IO} monad. Showing a dynamic blueprint for non-iTasks programs requires a new Tonic viewer, which we will discuss as well.

5.5.1 Dynamic Blueprints of the I/O Monad

Let’s look at how Tonic handles the \texttt{IO} variant of the \texttt{primeCheck} example. Figure 5.30 shows the dynamic blueprints for \texttt{primeCheck}. This dynamic blueprint is produced by an experimental stand-alone Tonic viewer, which serves as a proof-of-concept that such a stand-alone viewer can be constructed. As such, we are currently limited in the kind of information that we can dynamically show. The next section will elaborate on the implementation of the stand-alone viewer and talk about how the Tonic classes are implemented for the \texttt{IO} monad. We will also discuss some of the challenges we have encountered.

Creating a general (i.e. iTasks-agnostic) stand-alone viewer largely requires solving the same problems as for writing an embedded viewer: how does one load and draw the blueprints? How does one receive and process dynamic updates? How does one inspect dynamic data? It turns out that these questions become significantly more challenging when answering them for a general and stand-alone Tonic viewer. We will look at these aspects next.
5.5.2 Stand-alone Viewer Architecture

Instead of including the Tonic viewer as part of an iTasks program, the stand-alone viewer communicates with the to-be-inspected program via TCP. Figure 5.31 shows its architecture. There is a two-way communication channel between the original application (the server) and the Tonic viewer (the client).

![Architecture of the stand-alone Tonic viewer](image)

Blueprints are stored on disk in the same directory as the application for which they are generated. This allows the embedded Tonic viewer to locate them. The stand-alone viewer is not necessarily located in the same directory as the program that needs to be inspected, however. As a result, it cannot access the blueprints directly. Instead, it requests blueprints from the server and caches them, after which they can be drawn. The stand-alone viewer uses the same drawing mechanism as the built-in viewer.

Dynamic updates are provided by the Tonic wrappers. In the iTasks implementation, these wrappers write directly to the Tonic SDS. Wrappers for the stand-alone viewer write to a TCP connection instead. On the client-side, this data is stored again in an SDS.

5.5.3 Drawing Dynamic Blueprints

Figure 5.32 shows the protocol the Tonic viewer uses to instantiate blueprints and update them. When starting the client, it connects to exactly one server. The server registers the client, so it knows it can send updates to it when the program is executed. These updates are received by the client. If a given blueprint instance does not exist yet, the client tries to instantiate it. If the blueprint is not available on the client yet, it requests it from the server. Finally, the blueprint instance is updated and the client waits for the next update.

In the integrated Tonic viewer, blueprints are identified by a task’s unique identifier. In the stand-alone viewer, we abstract over this identifier by allowing it to be anything for which equality is defined. It is up to the implementation of the Blueprint and Contained classes to determine what the identifier is.

Inspecting values at run-time is another challenge in the stand-alone Tonic viewer. In the integrated viewer, we simply imposed the iTask constraint on any-
thing that could be inspected, allowing rich visualizations. In general, we cannot rely on this constraint being fulfilled. In the stand-alone viewer, we therefore currently disallow inspection of run-time values. One could take a first step towards dynamic value inspection by, for example, impose JSON (de)serialization constraints. Inspecting raw JSON data structures quickly becomes unwieldy for complex data structures, however.

5.5.4 Discussion

A clear downside to the approach presented above is that for each monad for which one wants to have dynamic blueprints, one needs to implement a blueprint server. The current implementation of the stand-alone Tonic viewer also has several limitations. It is currently not possible to inspect values or do dynamic branch prediction, nor is it possible to select which blueprint instance you are interested in; the viewer only ever shows the blueprint instance for which the latest update arrived. Still, we feel like this is an important step towards positioning Tonic as a general tool.

5.6 Related work

Tonic can be seen as a graphical tracer/debugger. Several attempts at tracer/debuggers for lazy functional languages have already been made. Some examples include Freja [74, 73], Hat [95, 96], and Hood [10], the latter of which also has a graphical front-end called GHood [90]. All of these systems are general-purpose and in principle allow debugging of any functional program at a fine-grained level. Tonic only allows tracing on a monadic abstraction level. Due to our focus on
monads, Tonic does support any monad, including the IO monad. All of the aforementioned systems only have limited support for the IO monad. Freja is implemented as its own compiler which supports a subset of Haskell 98. Previous Hat versions were implemented as a part of the nhc98 compiler. Modern Hat versions are implemented as stand-alone programs and support only Haskell 98 and some bits of Haskell 2010. Tonic is implemented in the Clean compiler and supports the full Clean language, which is more expressive than even Haskell 2010. Hood, on the other hand, requires manually annotating your program with trace functions.

GHood is a graphical system on top of Hood that visualizes Hood’s output. Its visualizations are mostly aimed at technical users. Graphical programming language, such as VisaVis and Visual Haskell, suffer from similar problems. Tonic explicitly aims at understandability by laymen by choosing a higher level of abstraction, hiding details that do not contribute significantly to understanding the program, and by utilizing coding conventions.

Another way to look at Tonic is as a graphical communication tool. In a way, it is similar to docblock-like technologies, such as Javadoc, in which documentation is included in the comments in the code. Docblocks are typically used to generate textual API documentation, rather than comprehensive graphical representations. In another way, Tonic is similar to UML and BPMN. Both of these technologies also offer a means to specify programs and workflows. This is something Tonic is not designed to do. Previous work from our group, GiN – Graphical iTasks Notation can be used for that.

5.7 Discussion and Conclusion

In this paper we generalised and expanded our original Tonic idea. Any monadic program can now be statically visualized by Tonic. While dynamic visualization is currently mostly limited to iTasks, we have laid the foundation for dynamically visualizing any monadic program.

So far, we have extensively experimented with using Tonic for iTasks. Our approach of using type classes for defining how dynamic behaviour should be captured allows for an almost completely orthogonal implementation for iTasks; the core system only required very minimal changes. The biggest change was made to the way iTasks handles task IDs. These IDs are not generated deterministically, so we had to implement a form of stack-tracing in iTasks to capture which tasks had already been executed. Systems with deterministic identifiers will not have to resort to such measures.

Section shows the results of experiments aimed at supporting dynamic blueprints for the IO monad. The fact that we can successfully generate these blueprints suggests that Tonic can be used in contexts other than iTasks as well. While this experimental Tonic viewer works reasonably well for simple IO programs, it lacks many of the features shown in Section and is not very user-friendly. In the future we want to expand this stand-alone viewer to the point where it can replace the built-in iTasks Tonic viewer.

1http://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javadoc.html
Complete iTasks Agnostisism

Even though we have made the Tonic compiler completely iTasks-agnostic, Tonic itself still is tied to iTasks by means of the iTask context restriction in the Blueprint and Contained classes. The iTask class is used to be able to generically inspect values. Its presence in the classes means that, even when using Tonic for non-iTasks programs, we require an iTasks-specific class to be instantiated for all types that we want to inspect. Clean’s type-system, however, offers no elegant solution to this problem. GHC in particular could solve this problem elegantly using its ConstraintKinds and TypeFamilies extensions, as shown in the code snippet in Figure 5.33. Here, the context restriction depends on the type of the Blueprint monad.

```haskell
class Monad m => Contained m where
  type CCtxt m a :: Constraint
  type CCtxt m a = ()
  wrapFunApp :: CCtxt m a => (ModuleName, FuncName) -> ExprId
              -> m a -> m a

class Contained m => Blueprint m where
  type BpCtxt m a :: Constraint
  type BpCtxt m a = ()
  wrapFunBody :: BpCtxt m a => ModuleName -> FuncName -> [(VarName, m ())]
              -> m a -> m a
  wrapFunArg :: BpCtxt m a => String -> a -> m ()

instance Contained Task where
  type CCtxt Task a = ITask a
  wrapFunApp = ..

instance Blueprint Task where
  type BpCtxt Task a = ITask a
  wrapFunBody = ..
  wrapFunArg = ..

instance Contained Maybe where
  wrapFunApp = ..
```

Figure 5.33: GHC definition of Tonic type classes.

For Clean, we could require values to be serializable to JSON so we can display data as a set of key-value pairs. While this approach would generalise the Tonic classes in the short term, it limits the ways in which we can present the inspected values. For example, we can currently render interactive graphics in the Tonic inspector. A true solution would be to implement variable context restrictions in type classes in Clean, similar to GHC.
Portability

By generalising Tonic it becomes clear that it could be implemented in a context different from Clean as well. Acknowledging that GHC in particular offers elegant solutions to improve Tonic’s type classes, it would be interesting to explore porting Tonic to GHC.

Dynamic Blueprint Modification

Tonic’s blueprints, whether static or dynamic, are currently read-only. We cannot influence the execution of programs or change a program’s implementation. In the future we would like to explore such possibilities.

Wrapping up

Tonic lays the foundation for a plethora of distinct but related tools. On the one hand, blueprints can be seen as automatic program documentation. Each time a program is compiled, its blueprints are generated as well, giving the programmer up-to-date documentation for free. Furthermore, dynamic instances of these blueprints document the program’s dynamic behaviour. Due to the blueprint’s high level of abstraction, this free documentation can serve as the basis of communication between various project stakeholders as well, enabling rapid software development cycles. Whether Tonic succeeds in being a suitable communication tool is a subject for future work.

Another way to look at Tonic is as a graphical tracer and debugger. Dynamic blueprints trace the execution of the program, while Tonic’s inspection and future branch prediction capabilities add features desirable in a debugger. Even for programmers, having such information visualized may aid in understanding the behaviour of the programs they have written better. It may also aid in constructing the required program faster or with less effort.

Yet another avenue worth exploring is education. We are currently including blueprints in the lecture slides of functional programming courses. In our experience, students struggle with the concept of monads, so we want to see if and how Tonic can reduce these problems.
Chapter 6
Towards the Layout of Things

When writing a user interface (UI), the layout of its elements play an important role. Programmers should be able to specify the layout of UIs in an intuitive way, while being able to separate the concern of laying out the UI from the rest of the software implementation. Ideally, the same layout language can be used in multiple application domains, so the programmer only has to learn one set of layout concepts. In this chapter we introduce such a general-purpose layout language. We obtain this language by abstracting from a layout language we have introduced in previous work for declaratively defining Scalable Vector Graphics (SVG). We show that this abstract layout language can be instantiated for multiple domains: the SVG library by which the language is inspired, ncurses-based text-based user interfaces, and iTasks. In all of these cases, a separation of concerns is maintained.

6.1 Introduction

Every user interface (UI) consists of a collection of possibly interactive UI elements. The layout of the UI can significantly influence the user experience. Being able to lay out appealing user interfaces is therefore important for user-facing software. Ideally, defining such UIs is easy to do for a programmer, and can be done while maintaining a separation of concerns from the business logic of a program.

In previous work \[6\], we have introduced a Scalable Vector Graphics \[27\] (SVG) library called Graphics.Scalable. With this library, one can create SVG images in a purely compositional way by combining basic SVG elements using a small set of layout combinators. Only three basic layouts were specified: collages, overlays, and grids. In a collage, each SVG element, which in turn may be a layout of SVG elements, is given an absolute position, while in an overlay the individual elements can be aligned relative to a parent container. A grid layout can be used to place SVG elements in rows and columns. The SVG library also features two derived layout combinators above en beside, which are defined as grids of 1 column and row, respectively.

Defining SVG images with these layout combinators turned out to be very practical. In fact, we also wanted to be able to express the layouts of our iTasks framework \[83\] \[68\] in the same terms. Rather than implementing a new layout language, we opted to abstract from our original layout language and use this abstract language to implement the layout combinators for both our SVG library and iTasks. At the same time, the new abstract layout language must also be powerful enough to capture other domains. This gave rise to the following questions. How should one specify the spatial layout of visual, possibly interactive, components in your program? How should one separate the concern of maintaining the life-cycle
of UI components from their chosen layout? How do you identify UI components? In this chapter we propose a general purpose solution to these challenges.

For application domains that provide direct access to the UI components, it suffices to instantiate the overloaded layout language. Many application domains do not provide direct access to their UI components, however. For instance, in most widget-based APIs the program must first call handle-returning actions and use the obtained handle values to control the life-cycle and layout of the created UI components. Thus, the concerns of creating UI components on one hand versus arranging their layout on the other hand is not well separated.

To still provide a separation of concerns, we introduce a way to perform pattern matching on a specific part of the structure of the program. An annotation function is applied to this part, which takes an abstract representation of the program’s structure over which layout can be specified. Pattern matching on this abstract representation can then be used to identify individual user-interface components. The annotation function returns a layout definition containing some or all of the UI components. With this annotation approach, the specification of layout can be decoupled from the identification of UI components. Naturally, if the annotated program fragment is changed, the identification code must also be re-considered. However, provided that the collection of identified UI components remains identical, this does not affect the layout specification. This also holds the other way around: changing the layout specification does not affect the identification code.

The proposed combination of overloaded layout language and function annotation works for completely different application domains. We demonstrate this with the following case studies:

1. The Graphics.Scalable library of iTasks is the obvious first candidate to consider because the overloaded layout language was derived from it. Because it provides direct access to its UI components, scalable images, it suffices to instantiate the overloaded layout language.

2. The ncurses library-available as a Haskell package-is the second case. With ncurses terminal-style “GUI” applications can be created. As with the previous case, ncurses provides direct access to the UI components, so it suffices to instantiate the overloaded layout language. However, the application domain is quite different as the programmer needs to divide the available screen estate to the appropriate UI components.

3. The third case is iTasks, but now we wish to arrange the layout of the automatically generated UIs of entire tasks and task compositions. This is an illustrative case of a domain in which the code annotation is required to identify the task UIs that need to be provided with a layout.

We have implemented the overloaded layout language in both Clean and Haskell. Both iTasks-related case studies are implemented in Clean, while the ncurses case is implemented in Haskell.

---

6.2. THE GENERALIZED LAYOUT LANGUAGE

triptych (cs, rs₁, rs₂) (a, b, c, d, e) = beside [] cs [] [ above (repeat AtRight) rs₁ [] [a, b] Nothing , c , above (repeat AtLeft) rs₂ [] [d, e] Nothing ] Nothing

Figure 6.1: The triptych layout specification

The remainder of this chapter is structured as follows. The overloaded layout language is introduced in Section 6.2, and the code annotation in Section 6.3. Section 6.4 up to Section 6.6 contain the three above-mentioned case studies. In Section 6.7 we analyse the properties of this approach and characterize for what kind of systems this is a viable solution. Related work is described in Section 6.8. The conclusions are found in Section 6.9.

6.2 The generalized layout language

In this section we introduce the generalized layout language. To get an intuitive understanding of the kind of layouts we wish to be able to specify, Section 6.2.1 starts with two examples. The layouts in these examples will be used throughout the rest of the chapter in various layout domains, without modifications. Important to note here is that different domains typically require arguments of different types for the layout functions. As such, reusable layout specifications need to be parametrised by these arguments as well, rather than just the individual elements that are to be positioned in the layout.

6.2.1 Layout language examples

Example triptych (Figure 6.1) illustrates a grid-based layout of a fixed set of items. A triptych consists of a large centre pane c, and two side panes. The centre pane and the side panes are placed beside one another using the beside layout. The side panes are usually, but not always, half the width of the centre pane. We divide each side pane into two sub-panes a, b and d, e. Sub-panes a and b are placed above one another using the above layout, and so are sub-panes d and e. Sub-panes a and b are right-aligned, while sub-panes d and e are left-aligned. It is the responsibility of the application domain to express the correct placements of the elements within the beside and above layouts. This is done via the parameters cs, rs₁, rs₂. The type of these parameters and the way in which they influence layout can differ per domain.

The second example, rolodex (Figure 6.2), illustrates an overlay-based layout of an arbitrary number of elements. In an overlay layout, elements at the front of the list are place underneath the elements later in the list. The middle item,
rolodex _ _ []
    = overlay [] [] Nothing
rolodex x y things
    = overlay
      ( repeatn na (AtLeft, AtTop)
        ++ repeatn nb (AtLeft, AtBottom)
      )
      ( reverse (take na (zip2 x' y'))
        ++ reverse (take nb (zip2 x' y'))
      )
      ( as ++ bs ++ [c ] )
Nothing
where
n   = length things
na  = n / 2
nb  = n - na - 1
(as, [c : bs']) = splitAt na things
bs  = reverse bs'
x'  = tl (scan (+) zero x)
y'  = tl (scan (+) zero y)
y'' = tl (scan (-) zero y)

\[ \begin{array}{cccc}
  x_0 & x_1 & \ldots & \cdot \\
  y_0 & y_1 & \ldots & \cdot \\
\end{array} \]
\[ \begin{array}{cccc}
  \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot \\
\end{array} \]

Figure 6.2: The rolodex layout specification

c, must be displayed closest towards the viewer, so it must be placed at the very end of the list of elements of the layout function. The preceding items, \( as \), are displayed above \( c \) on the \( y \)-axis and at increasing horizontal offset \( (x_0, x_0 + x_1, x_0 + x_1 + x_2, \ldots) \) and increasing upward offset \( (y_0, y_0 + y_1, y_0 + y_1 + y_2, \ldots) \). Similarly, the subsequent items, \( bs' \), are displayed below \( c \) on the \( y \)-axis and at increasing horizontal and downward offset. In order to create the correct \( z \)-axis ordering, \( bs' \) is in reversed order \( (bs) \) in the list of elements of rolodex.

### 6.2.2 Implementing the generalized layout language

We generalize the original layout language from **Graphics.Scalable** by means of several type classes, which we will introduce in the rest of this section. Every layout is a function that arranges a finite list of elements, or *things*, into a new, composite, *thing*. The arrangement is specified by means of a list of *offsets*, that correspond one-by-one with the list of *things*. If the list of offsets happens to be shorter, then it is padded with default values (*zero* in case of offsets), and if it happens to be longer, then it is truncated. This is a general design guideline throughout the layout language. The meaning of the offsets is determined by a *host*. If a host is present, its coordinate system is used. If it is absent, then the coordinate system is found by taking the bounding box of the dimensions of all things. This amounts to the following type of every layout function:

\[
:: \text{LayoutFun} \text{ offset thing host m}
\]

114
6.2. THE GENERALIZED LAYOUT LANGUAGE

\[\text{offset} \rightarrow \text{thing } m \rightarrow \text{Maybe (host } m) \rightarrow \text{thing } m\]

The types of \textit{host} and \textit{thing} are parameterized with type variable \textit{m} to accommodate the expected instances. The coordinate system is conventional: the \textit{x}-axis increases towards the right of a display area, and the \textit{y}-axis increases towards the bottom of a display area. There is an implicit \textit{z}-axis that increases towards the viewer. By convention, elements that occur at a higher index in the list of things have a higher \textit{z}-value, and can thus obscure elements at a lower index position.

The core of the layout language is formed by two multi-parameter type classes: \textbf{Overlay} and \textbf{Grid}. We introduce two separate type classes because not every application domain handles the two key concerns: ordering things along the \textit{z}-axis and ordering things within the \textit{x}- and \textit{y}-plane. Typically, application domains that do not handle overlapping elements will not support the \textbf{Overlay} language. An application domain instantiates the classes by choosing types for the \textit{things}, \textit{dimensions}, \textit{offsets}, and \textit{host}. The \textit{thing} type determines the other types, which is denoted in Clean by means of a functional dependency by the prefix \textit{~}.

The main concern of the \textbf{Overlay} class member functions is to control the layout of things in the \textit{z}-axis, using the implicit ordering in its list of things:

\begin{verbatim}
class Overlay thing ~offset ~host where
  overlay :: [(XAlign, YAlign)] -> LayoutFun offset thing host m
  collage :: LayoutFun offset thing host m

:: XAlign = AtLeft | AtMiddleX | AtRight
:: YAlign = AtTop | AtMiddleY | AtBottom
\end{verbatim}

Here, the functional dependency reads as \textit{thing uniquely identifies offset} and \textit{host}. In other words, once the type system knows the type of \textit{thing}, it knows from the functional dependency what the types of \textit{offset} and \textit{host} are. An (\textit{overlay aligns offsets things host}) first aligns every \textit{thingsi} according to \textit{alignsi} with respect to \textit{host}. The default value for \textit{aligns} is \textit{(AtLeft, AtTop)}, and the list is either padded or truncated to match the length of \textit{things}. Second, the position of \textit{thingsi} is tuned with \textit{offsetsi}. The \textit{collage} class member function is a convenience function that has default implementation \textit{collage = overlay []}: the placement of its elements is dictated by the implicit \textit{z}-axis and their offsets.

The main concern of the \textbf{Grid} class member functions is to control the \textit{x}- and \textit{y}-axis.

\begin{verbatim}
class Grid thing ~dim ~offset ~host where
  grid :: GridDimension GridLayout
        [ (XAlign, YAlign)] [dim] [dim] -> LayoutFun offset thing host m

  beside :: [YAlign] [dim] -> LayoutFun offset thing host m
  above :: [XAlign] [dim] -> LayoutFun offset thing host m

:: GridDimension = Columns Int | Rows Int
:: GridMajor = ColumnMajor | RowMajor
:: GridXLayout = LeftToRight | RightToLeft
:: GridYLayout = TopToBottom | BottomToTop
\end{verbatim}
CHAPTER 6. TOWARDS THE LAYOUT OF THINGS

:: GridLayout ::= (GridMajor, GridXLayout, GridYLayout)

A (grid dim layout aligns cols rows offsets things host) places things in a grid structure. Its number of columns and rows is determined by dim. The order of grid-cells that are selected is determined by layout: the GridMajor value dictates whether this occurs column-by-column or row-by-row; the GridXLayout value determines if the grid is subsequently filled from left-to-right or right-to-left, and finally, the GridYLayout value determines if the grid is subsequently filled from top-to-bottom or bottom-to-top. Every things_i is aligned within its grid-cell according to aligns_i (default value is (AtLeft,AtTop)). The cols (and rows) lists add additional constraints on the widths (heights) of the columns (rows). The application domain determines what the default value is. The final position of things_i is obtained by tuning with offsets_i.

Grid also has convenience layout functions with default implementations:

beside as cs = grid (Rows 1) (RowMajor, LeftToRight, TopToBottom)
\[ [(\text{AtLeft}, a) \\ \ \ \ \ | \ \ \ \ \ \ \ \ \ \ \ \ a \leftarrow as] \ \ cs [] \]

above as rs = grid (Columns 1) (ColumnMajor, LeftToRight, TopToBottom)
\[ [(a, \text{AtTop}) \\ \ \ \ \ | \ \ \ \ \ \ \ \ \ \ \ \ a \leftarrow as] \ \ [] \ \ rs \]

In general, a layout language needs a means to refer to the size of its components and perform computations on them (add, subtract, maximum, minimum) in order to construct a correct composition. This requires a tag type to identify the component, and a dim type that represents the size of the component. This is captured with the multi-parameter type classes TagOf and DimRef:

class TagOf thing ~tag where
  tagOf :: thing -> tag

class DimRef tag ~dim where
  xdim :: tag -> dim
  ydim :: tag -> dim
  cdim :: tag Int -> dim
  rdim :: tag Int -> dim

The expression (tagOf thing) retrieves the tag of thing. It is the responsibility of the application domain to assign to each thing one unambiguous tag (the system tag). The application domain can optionally offer a tag function to add custom tags to things. Hence, in general, a thing is associated with a non-empty collection of tags. The expressions (xdim t) and (ydim t) refer to the x-width and y-height of the thing that is tagged with t. The expressions (cdim t i) and (rdim t j) refer to the x-width of the i-th column or the y-height of the j-th row of the thing that is tagged with t. Tag-expressions are symbolic references. It is the concern of the implementation to take this into account when computing with such values. Unmatched tag expressions always have value zero.

Computations with dimensions come with instances for common overloaded arithmetic functions +, -, abs, and ~ (negation). Slightly less usual are the overloaded functions zero, *, /, min, and max. zero produces a zero-like value, such as 0 or 0.0 for integers and real numbers, respectively. Expressions \((d *\ \ k)\) and
(\(d / k\)) multiply and divide a dimension \(d\) with a scalar value \(k\). Expressions \((\min ds)\) and \((\max ds)\) take the minimum and maximum of a list \(ds\).

6.3 Relating UI components to layout

Not every application domain is suited to instantiate the overloaded layout language of things. There are several issues that make it complicated to do this:

1. In most widget-based API’s the program must first call \(handle\)-returning actions and use the obtained \(handle\) values to control the life-cycle and layout of the created UI components. As a result, the layout expression language becomes a layout action language that is interleaved with the UI action language.

2. One cannot use the UI creation code to identify the resulting UIs because each call of the same code results in a newly created UI instance whose appearance, content, and size diverges over time when compared with the other instances.

3. Most applications require a UI that evolves dynamically over time: the number of windows, panels, items keep changing to best reflect the current application state and needs of the users. Hence, the layout specification needs to be dynamic as well.

4. To counter the above limitation, many UI toolkits offer an API to inspect the widget-structure or DOM-structure at run-time. However, these APIs can break the abstraction barrier that is intended by the implementer of UI components.

5. Some UI approaches implement an automatic layout algorithm while other approaches leave the specification of layout entirely to the programmer. In the first case, the application developer might want to overrule the layout of a particular piece of code and leave other pieces as-is. The solution should work for all of these cases.

We counter the above issues in the following way. First, to deal with issue 5, we introduce a code-annotation. Code without an annotation behaves as dictated by the application domain. Annotated code gets overruled by the specification within the annotation. The annotation specification is a function. This deals with issues 1 and 5. The function is provided with information of the current collection of UI components. The application domain is responsible for providing the information, and can thus protect the program against breaking the abstraction barrier. To resolve issues 2 and 4, we observe that UI components are always organized in a hierarchical way (for instance, windows containing child elements, some of which can be panes that contain further child elements, and so on). The argument of the function-annotation is a rose tree parameterized with the type of UI components of the application domain. The application domain defines the relation between the annotated code and the rose tree.
The rose tree structure of UI components is defined as follows:

\[
\text{:: UITree tag} = \text{UILeaf tag} \mid \text{UINode tag [UITree]}
\]

\[
\text{tagOfUI :: (UITree tag) -> tag}
\]

The `tag` type parameter uniquely identifies the nodes and leaves of the rose tree. It is the first type parameter of every `DimRef` type class instantiation. The trivial access function `tagOfUI` simply returns the tag found at the root of its argument. The application domain decides which compositions of UI components can be decomposed (`UINode`) or are considered to be atomic (`UILeaf`).

The rose tree structure is the domain of the function annotation. The range type depends on the application domain: types need to be defined to instantiate the type classes `Overlay`, `Grid`, `TagOf`, and `DimRef` of the overloaded layout language. If type `T` is the type of `things` of a particular application domain, then the layout function has type:

\[
\text{my_layout :: (UITree tag) -> T m}
\]

In this way, the layout language is open ended to allow an application domain to add further constructor functions to create UI elements or transformations specific for that domain. For instance, in SVG based approaches you wish to support rotation and skewing, and in widget based GUIs, you wish to support panels that can be scrolled or resized.

The missing link connects the UI rose tree with the domain of things via the overloaded function `uiOf`:

\[
\text{class UIOf thing ~tag where}
\]
\[
\text{uiOf :: (UITree tag) -> thing m}
\]

### 6.3.1 UI pattern examples

The first example applies the `triptych` layout to a piece of code. Hence, the annotation needs to identify five UI items. If we know that the program is structured as shown in the image to the right, then the corresponding layout definition can be defined as follows:

```haskell
example_1 (cs, rs_1, rs_2) (UINode _ [ UINode _ [a, b]
, UINode _ [d, e]
, c ])
= triptych (cs, rs_1, rs_2) (a', b', c', d', e')
where
[a', b', c', d', e'] = map uiOf [a, b, c, d, e]
```

In this example we pattern match on a `UITree` value to identify sub-components in the original UI tree. Important note is that sub-layouts `a`, `b`, `c`, `d`, and `e` can be arbitrarily complex user interfaces themselves. We maintain manageability by not pattern matching to the leaves of the tree.

The second example collects all leaf UI items of an annotated piece of code and applies the `rolodex` layout to them.

---

118
6.4. THE LAYOUT OF GRAPHICS.SCALABLE

6.4 The layout of Graphics.Scalable

The overloaded layout language defined in Section 6.2 is a generalization of the original layout language of Graphics.Scalable. As a consequence, this section is brief, and serves mainly as an overview of the two tasks that have to be performed to apply the overloaded layout language to a new domain: identify the domains and the constructor functions.

6.4.1 Graphics.Scalable domains

First we define the domains that the type classes Overlay, Grid, TagOf, and DimRef are instantiated with. These are: the domain of things, dimensions, offsets, hosts, and tags. Their type definitions are:

```haskell
type Image m = ()  -- domain of things
newtype Span = Span  -- dimensions
data Offset = Offset (Span, Span)  -- offsets
data Host m = Host m  -- hosts
data ImageTag = ImageTag  -- tags
```

The domain of things in Graphics.Scalable is captured with the opaque Image m type. Every image is infinitely large and perfectly transparent. There is no global coordinate system. With each image a span box is associated relative to which visual content is rendered. The dimensions are captured with the opaque Span type. Although span values are defined most of the time with Real values, they get a ‘physical’ pixel-based interpretation only when an image gets rendered at a client device. Offsets are a pair of a horizontal and vertical span values. The host is an image that serves as the ‘background’ image, and its span box is used to deal with the alignments and offsets. Finally, tags are captured by the opaque ImageTag type.

With these domains, we obtain the layout language of images:

```haskell
instance Overlay Image Offset Host
instance Grid Image Span Offset Host
instance TagOf Image ImageTag
instance DimRef ImageTag Span
```
Their implementations map to the existing implementation of `Graphics.Scalable`.

### 6.4.2 Graphics.Scalable constructor functions

The next step is to define the application domain dependent constructor functions of the domain types. For `Image` these are the common shapes: rectangles, circles, ellipses, lines, and text. Except for text, these shapes are defined via their span box (where circles require only the diameter). The image of a text is defined with:

```hs
text :: FontDef String -> Image m
```

`FontDef` captures the standard SVG attributes to define a font. The $y$-span of the text is defined by the `fontysize` field, using a real number. However, the $x$-span of a text, rendered with a given font, is determined by the client device. This is a major complication when dealing with the layout of text. In `Graphics.Scalable`, the function `textxspan` is a symbolic span-expression that represents the $x$-span of the given text, when rendered with the given font, on the current client device.

```hs
textxspan :: FontDef String -> Span
```

It should be noted that the SVG image transformation functions (rotate, skew, flip) and image rendering attributes (stroke, opacity) do not alter the span box of the transformed image, but only their rendering. Hence regarding layout, they are irrelevant. However, this is not the case for the scaling functions:

```hs
fit :: Span Span (Image m) -> Image m
fitx :: Span (Image m) -> Image m
fity :: Span (Image m) -> Image m
```

`fit x y img)` guarantees that the result has precisely width $x$ and height $y$. `fitx x img)` guarantees that the result has precisely width $x$, and derives height $y$ proportionally to the current size of `img`. Similarly, `fity y img)` guarantees that the result has precisely height $y$, and derives width $x$ proportionally to the current size of `img`. These `Spans` can be constructed with, amongst others, the `px` function:

```hs
px :: Real -> Span
```

The final application domain dependent constructor functions concern the opaque `ImageTag` type. The top level function to create an image of a server-side value of type `s` and client-side value of type `m` has type:

```hs
s -> m -> *[(ImageTag, *ImageTag)] -> Image m
```
6.4. THE LAYOUT OF GRAPHICS.SCALABLE

Figure 6.3: Rendering of the card image in Graphics.Scalable

Figure 6.4: Screenshot of the triptych example in Graphics.Scalable

The list of image tag values is infinitely long and is generated by the image library implementation. The two tags in the tuple are the same tag, but they make different use of Clean’s uniqueness types \( \text{[15]} \). With this extension of the type system, types can be annotated with a uniqueness attribute \( \ast \). The attribute guarantees that there is ever only exactly one reference to a given unique value. The uniquely attributed version is used in the function \( \text{tag} \):

\[
\text{tag} :: \ast \text{ImageTag} \ (\text{Image} \ m) \rightarrow \text{Image} \ m
\]

to guarantee that it adds the tag to the non-empty tag set of at least one particular image. The shared version of \( \text{tag} \) can be used arbitrarily many times using the type class \( \text{DimRef} \) member functions.

6.4.3 Graphics.Scalable examples

Let \( \text{card} :: \text{Span} \ \text{Span} \ \text{Int} \rightarrow \text{Image} \ \text{Int} \) create an image of size \( w \times h \) that renders a steelblue ‘card’ on which the number is printed in white (Figure 6.3).

We use this function to create a triptych of five cards, using the unmodified triptych layout as defined in Section 6.2.1 (Figure 6.4) for some given card size \( w \) and \( h \). In the triptych, we proportionally scale the height of the side-panel cards to half the height of the central card.

\[
\text{a} = \text{triptych} (\square, \square, \square) (\text{fity} (h / .2) (\text{card} \ w \ h \ 1))
\]
\[
\quad , \text{fity} (h / .2) (\text{card} \ w \ h \ 7)
\]
\[
\quad , \text{card} \ w \ h \ 42
\]
\[
\quad , \text{fity} (h / .2) (\text{card} \ w \ h \ 4)
\]
\[
\quad , \text{fity} (h / .2) (\text{card} \ w \ h \ 2)
\]
\[
\)

As another example, we create a rolodex of cards 1 through 16 (Figure 6.5), using the unmodified rolodex layout as defined in Section 6.2.1. The card widths decrease by steps of 0.08\( w \), so the horizontal offsets increase by steps of 0.04\( w \). The vertical offsets increase by \( \frac{h}{2}, \frac{h}{3}, \ldots \).

\[
\text{b} = \text{rolodex} (\text{repeat} (w * .04)) [h / . k \ \backslash \ k <= [2..]]
\]
\[
\quad (\text{zipWith} \text{fitx} (\text{reverse} \ (\text{take} \ 7 \ ws)) \ ) \text{as}
\]

121
CHAPTER 6. TOWARDS THE LAYOUT OF THINGS

Figure 6.5: Screenshot of the rolodex example in Graphics.Scalable

```haskell
++ [c]
++ zipWith fitx ws bs
)
where
cards = map (card w h) [1 .. 16]
(as, [c:bs]) = splitAt 8 cards
ws = [0.92, 0.84 ..]
```

6.5 The layout of ncurses

The ncurses library is a well-known library that supports command-line interface (CLI) based user interfaces. It allows rendering and placement of text or glyphs, and allows the programmer to specify how keyboard or mouse interaction should be dealt with. To demonstrate the applicability of the overloaded layout language outside the browser, we have implemented it for ncurses as well. In particular, it allows the programmer to layout the glyphs. It does not deal with the interaction. Since Clean does not currently have bindings for ncurses, we ported the layout library to Haskell. This allows us to use the ncurses package from Hackage. Porting the layout language to Haskell is straightforward and involves only minor syntactical changes. Therefore, we do not show the Haskell definition of the type classes.

6.5.1 The ncurses domains

The type definitions of the domain of things, dimensions, offsets, hosts, and tags in ncurses are:

```haskell
data CursesElem m -- domain of things
data CursesSpan -- dimensions
type CursesOffset = (Int, Int) -- offsets
data CursesHost a -- no hosts for NCurses
data CursesTag -- ...
```

In ncurses, a terminal is divided in a grid of mono-spaced characters and glyphs. Each character and glyph takes up a single cell in the grid. A string is simply
6.5. THE LAYOUT OF NCURSES

a sequence of (potentially multi-byte) mono-spaced characters. These properties make doing layout for a terminal easy, since the width and height of each character and glyph is a fixed $1 \times 1$. A string is then simply $1 \times n$ in size, where $n$ is the number of characters in the string. In other words, we always know the size of the things that we want to layout. Hence, offsets are integers, each of which represents an on-screen cell. There is no implementation of the DimRef class and the host concept for ncurses. With the ncurses domain types defined, we create the following class instances:

\begin{verbatim}
instance Overlay CursesElem CursesOffset CursesHost
instance Grid CursesElem () CursesOffset CursesHost
instance TagOf CursesElem CursesTag
instance DimRef CursesTag CursesSpan
\end{verbatim}

The main building block is the CursesElem. CursesElem values can be created with instances of the ToCurses class:

\begin{verbatim}
class ToCurses a where
c :: a -> CursesElem()

instance ToCurses String
instance ToCurses Glyph
instance ToCurses Text
instance ToCurses (CursesElem ())
\end{verbatim}

6.5.2 Examples of ncurses

Using the exact same layout specifications as shown in Section 6.2.1, we can also lay out ncurses elements. Here we will lay out rectangles with a border using the rect function.

\begin{verbatim}
rect :: Int -> Int -> CursesElem () -> CursesElem ()
\end{verbatim}

rect takes a width and a height, as well as a curses element that is rendered in the centre of the rectangle. Using this function, we can now visualize the triptych example. The triptych code is shown below, with its output in Figure 6.6.

\begin{verbatim}
a = triptych([[]], [[]], [[]]) (rect 7 3 $ c "A", rect 7 3 $ c "B"
, rect 7 8 $ c "C"
, rect 7 3 $ c "D", rect 7 3 $ c "E")
\end{verbatim}

The rolodex example is equally simple. Its code is shown below, with a screenshot of its rendering in Figure 6.7.

\begin{verbatim}
b = rolodex (repeat (px 1)) [px 3, px 2, px 1]
[ rect 5 3 $ c "1", rect 7 3 $ c "2"
, rect 9 3 $ c "3", rect 7 3 $ c "4"
, rect 5 3 $ c "5"]
\end{verbatim}
6.6 The layout of Task UIs

From the perspective of reasoning about layout, the iTasks application domain belongs to the ‘problem’ category discussed in Section 6.3: tasks are actions for which a UI is generated automatically, the UI is carefully hidden from the application programmer, and the same task gives rise to distinguishable UI instances. Hence, we need to use the function annotation of Section 6.3 to identify and layout task UI items. The current implementation of iTasks does not support overlapping UI items, so it cannot support the Overlay layout language. It does support the Grid layout language.

6.6.1 Task UI domains

As before, we start to define the domain of things, dimensions, offsets, hosts, and tags for the iTasks application domain. The type definitions are:

```haskell
:: TaskUILayout a  -- domain of things
:: TaskUISize = { minSize :: UISize, maxSize :: UISize, hasSplitter :: Bool }  -- domain of things
```
:: UISize = FlexSize | WrapSize | ExactSize Int | PctSize Real
  // dimensions
:: Offset :== (Int, Int)  // offsets
:: UIHost m = InHost    // hosts
:: TaskUITag            // tags

class  toUISize a :: a -> UISize
instance toUISize Int           // convert to ExactSize
instance toUISize Real          // convert to PctSize

exact p = ExactSize p
pct p = PctSize (fromInt p)
mkTaskUISize a b c = {minSize = a, maxSize = b, hasSplitter = c}

gDefault{|TaskUISize|} = mkTaskUISize FlexSize FlexSize False
fixUISize a = mkTaskUISize (toUISize a) (toUISize a) False
splitUISize a b = mkTaskUISize (toUISize a) (toUISize b) True

The opaque type TaskUILayout is only used to introduce the member functions of the Grid layout language to iTasks.

The language of dimensions is much richer than in the previous case studies. As with ncurses, layout of task UIs is concerned with dividing screen estate to UI items but without limiting it to the very simple cell-based approach of terminal-style UIs. By default, the host is divided equally. This can be altered by means of the TaskUISize parameter. It controls the widths of the columns in case of beside, the height of the rows in case of above, and both in case of grid. The minimum and maximum sizes (minSize and maxSize) are specified by means of a UISize value. The default value FlexSize imposes no restrictions, WrapSize is the bounding size of the elements, ExactSize p (p ≥ 0) is exactly p pixels, and PctSize p (0 ≤ p ≤ 100.0) is \( \frac{p}{100} \) of the host size, after rounding to integer pixels. The task layout algorithm computes (column and row) sizes within these constraints. In case the specified minSize < maxSize, the user can be allowed to manually choose a valid size between these values. This is indicated by means of the hasSplitter field. We call a TaskUISize rubber if minSize < maxSize and no splitter has been requested, and we call it splitter if minSize < maxSize and a splitter has been requested. A splitter user interface element (depending on the client device) is created between column i and i + 1 if either the width of column i is splitter or if the width of column i is rubber and the width of column i + 1 is splitter (analogous for rows).

Offsets are expressed as pairs of pixels. The UIHost type reflects the twofold purpose of defining task UI layout to assign to each identified task UI a part of the available screen estate (Just InHost), and to arrange the relative positioning of the task user interfaces (Nothing). For instance, one can first define the layout of a collection of UI items using Nothing, obtaining a composite UI of a certain size, and then place and align it in a smaller part of the screen using (Just InHost).

The TaskUITag type connects the function annotation with the layout. The standard way in iTasks to annotate a piece of code is by means of the prefix / postfix tune combinators:

class tune b :: b (Task a) -> Task a
CHAPTER 6. TOWARDS THE LAYOUT OF THINGS

(@>) infixr 2 :: b (Task a) -> Task a \| tune b
(<@>) infixl 2 :: (Task a) b -> Task a \| tune b

We wrap the function annotation in a new type and turn it into an instance of the `tune` type class:

:: TaskLayout = E.a: TaskLayout ((UITree TaskUITag) -> TaskUILayout a)

instance tune TaskLayout where ...

Here, the task is annotated with a function that is given a `UITree` (as defined in Section [6.3]) and produces a layout. With these domains, we obtain the layout language of task UIs:

instance Grid TaskUILayout Int Offset UIHost
instance TagOf TaskUILayout TaskUITag
instance DimRef TaskUITag UISize

6.6.2 User interfaces in iTasks

In this section we look at the internal implementation of user interfaces in iTasks in order to identify some of the challenges faced when implementing a layout language for such a sophisticated framework. In iTasks, user interfaces and their sub-elements are represented as a rose tree of type `UI` (shown below). Each node in the tree has a label indicating the type of user interface element, and a map of attributes for that node.

:: UI = UI UINodeType UIAttributes [UI]

:: UINodeType
  = UIParallel
  | UIStep
  | UIPanel
  | UIInteract
  | UIButton
  | ..

:: UIAttributes ::= Map String JSONNode

To minimize network traffic and computation time, iTasks updates its user interfaces incrementally, communicating only that what has changed. This incremental communication is represented by the `UIChange` type, shown below. Changes are applied to the user interface by the `applyUIChange` function. Its definition is omitted, but its type is included to show which types play a role in user interface updates.

:: UIChange
  = NoChange
  | ReplaceUI UI
  | ChangeUI [UIAttributeChange] [(Int, UIChildChange)]

applyUIChange :: UIChange UI -> UI
Applying \texttt{NoChange} acts as identity operation, while \texttt{ReplaceUI} simply replaces the entire user interface with a new one. \texttt{ChangeUI} is responsible for updating individual user interface components and is most frequently used.

To apply layout to tasks, we need to integrate the layout language with this user interface update mechanism. Layout is always specified on the static layout defined by the static task composition. At runtime, the layout may change dynamically via the \texttt{ChangeUI} change. These same changes will need to be reflected in the layout, so that the layout rules can also be applied after the user interface has been updated.

The dynamic behavior of user interfaces in iTasks complicates the application of layouts. To be able to layout, the components that are being layed out have to exist for the layout to be meaningful. In a fully dynamic setting this can not be guaranteed. We therefore consider the layout language only under the following conditions:

- A layout has to be explicitly applied to a part of a UI.
- When a UI is replaced completely with a \texttt{ReplaceUI} change, a layout can rearrange arbitrary sub-UI's into a new UI.
- Subsequent UI changes are only allowed if they modify the content of a sub-UI. If they affect the structure of the UI as transformed by the layout, a run-time error is produced.

With these restrictions the layouting can be achieved as follows. We first consider the \texttt{ReplaceUI} changes. When those occur we uniquely label every node of the UI that is being replaced. Then, using the layout language, we create a UI to UI transformation and apply it to the UI. In the transformed UI we can inspect the labels to build a relocation map that records which parts of the original UI were used and their position in the new UI. On subsequent \texttt{ChangeUI} events, we use the relocation map to detect if the change targets the content of the relocated parts of the UI or the structure of the layed out UI. If only the content is affected, we rewrite the \texttt{ChangeUI} to target the relocated UI parts.

The current implementation of the task layouts only implements horizontal, vertical and grid layouts, without support for offsets. In other words, we only implement the \texttt{Grid} class for iTasks. This is not a fundamental limitation, but rather a limitation of the iTasks client implementation, which was not designed to work with arbitrary collages. Future versions of the iTasks client may add support for free-form layouts.

### 6.6.3 Task UI constructor functions

An iTasks specification defines the work that needs to be done by end-users and computer systems, each of whom and which have different locations and use different client-devices to perform their work. With each end-user, a collection of tasks is associated. The iTasks run-time system collects these tasks, determines the corresponding UI items to be rendered for the particular client device that is used by the end-user at that time, and subsequently assembles a suitable UI for
the end-user using a default layout algorithm. There are two classes of tasks that generate UI items: the interactive tasks, called editors, and task combinators that introduce control items for an end-user to interact with.

### 6.6.3.1 Editor tasks

An editor is a generic task with which an end-user can enter, view, or update a value of any first-order (custom or pre-defined) type.

```haskell
enterInformation :: String [EnterOption m] -> Task m | iTask m
viewInformation :: String [ViewOption m] m -> Task m | iTask m
updateInformation :: String [UpdateOption m m] m -> Task m | iTask m
```

A descriptor string is used to inform the end-user of the purpose of this task. The EnterOption, ViewOption, and UpdateOption parameters can be used to provide a custom rendering function for the value of the editor. The use of these options does not influence the way editors are placed in a layout, so we do not discuss them further. With enterInformation, the end-user creates a new value of type m. In case of the other two editors, an initial value is provided. The UI of viewInformation only displays the value, and the UI of updateInformation allows the user to alter it. Figure 6.8 shows typical renderings of these elements (using the descriptors "enter", "view", "update" and the initial value 42):

The above interaction tasks are naturally applied in cases where the value to be interacted with is carried along with the control flow. However, data sources also exist outside of the control flow, and require interaction as well. The following sibling interaction tasks do this:

```haskell
viewSharedInformation :: String [ViewOption r] (ReadWriteShared r w) -> Task r | iTask r
updateSharedInformation :: String [UpdateOption r w] (ReadWriteShared r w) -> Task w | iTask r & iTask w
```

Instead of an initial value to work on, they manipulate a shared data source. Changes to the shared data source are propagated to all interaction tasks that are connected thusly.

Editors can be customized in case the default rendering is inadequate. This is done via the (Enter/View/Update)Option parameter. For any editor one can choose to map the value to another first-order domain that is automatically rendered. A UI can also be defined from scratch for update(Shared)Information using editlets. In particular, it is possible to use an Image (Section 6.4) to render the content of an editor. Here, we customize an Int editor to show its content as a card:
6.6. THE LAYOUT OF TASK UIs

Figure 6.9: Graphical rendering of cards.

```
edit_card = updateInformation "card" [asImage o card w h]
edit_card' = updateSharedInformation "card" [asImage o card w h]

asImage :: (Image a) -> UpdateOption Int Int
```

The result of either of these editors is shown in Figure 6.9. How \texttt{asImage} can be implemented is explained in Chapter 4, which introduces our compositional vector graphics library. In all of the above cases, the UI of an editor is accessed as a leaf in the UI rose tree.

6.6.3.2 Task combinators

All possible ways of collaboration boil down to two core task combinators: \textit{sequential} and \textit{parallel} composition.

Sequential composition, denoted with \texttt{\textbullet\textbullet\textbullet} and pronounced as \textit{step}, is basically a generalized, guarded version of the standard monadic \texttt{\textbullet\textbullet\textbullet}, \textit{bind}, operator in the presence of task values that evolve over time.

```
(\textbullet\textbullet\textbullet) \textbf{infixl} 1 :: (Task a) [TaskCont a (Task b)] -> Task b \| iTask a \& iTask b
```

```
:: TaskCont a b = O nValue ((TaskValue a) -> Maybe b)
| O nAction Action ((TaskValue a) -> Maybe b)
| E . e: O nException (e -> b) \& iTask e
| O nAllExceptions (String -> b)
:: TaskValue a = NoValue | Value a Stability
:: Stability ::= Bool
```

The UI control elements originate from the guarded \texttt{OnAction} task continuations. The \texttt{Action} parameter indicates that a user can interact with the application via a clickable user interface element, such as a button or menu item. These actions are co-located with the UI of the left-hand side task argument of \texttt{\textbullet\textbullet\textbullet}. For the purpose of the layout proposal in this chapter, we consider them to be an integral part of the UI that is matched on. The UI belongs to the leaf constructor of the UI rose tree.

Parallel composition, denoted with \texttt{parallel}, captures the collaboration of a (possibly dynamic) number of tasks. The progress between these tasks is accessible to both the participating tasks as well as any external context (such as a guarded task continuation of the \texttt{\textbullet\textbullet\textbullet} combinator). Without going into too much detail, we briefly walk through the signature of \texttt{parallel}:
parallel :: [(ParallelTaskType, ParallelTask a)]
    [TaskCont [(TaskTime, TaskValue a)] (ParallelTaskType, ParallelTask a)]
-> Task [(TaskTime, TaskValue a)] \ iTask a

The ParallelTaskType governs the end-user-ownership of the task. A task can be embedded or detached, thus enabling task distribution between co-workers. The ParallelTask is a task function that is provided with access to the current status (task values and meta-information) of the collaborating tasks via a shared data source. The task continuation (TaskCont, see \textgreater\ast\above) can add new tasks to the collection of tasks (tasks can also be removed). For the purpose of the layout proposal in this chapter, parallel composition is an ordered sequence of UI items from the perspective of the current end-user. Each UI is a sibling node within the node constructor of the UI rose tree. A UI belonging to another end-user is empty, having dimensions of zero size.

Many task compositions have a simple static structure and can do without the rather elaborate signature and interface of parallel. For instance, some frequently occurring combinations are:

\((-\&-\&-)\) infixr 4 :: (Task a) (Task b) \rightarrow Task (a,b) \mid iTask a \& iTask b
\((-\mid\mid-)\) infixr 3 :: (Task a) (Task a) \rightarrow Task a \mid iTask a
allTasks :: [Task a] \rightarrow Task a \mid iTask a
anyTask :: [Task a] \rightarrow Task a \mid iTask a

\((t_1 -\&-\&- t_2)\) evaluates two tasks in parallel until both have a stable task value, and allTasks generalizes this to a list of tasks. \((t_1 -\mid\mid- t_2)\) evaluates two tasks in parallel until either one has a stable task value, and anyTask generalizes this to a list of tasks. In these cases, each of the task UIs are retrieved via the node constructor of the UI rose tree.

For any derived task combinator, the corresponding UI rose tree structure must be documented.

### 6.6.4 Task UI examples

We use the triptych layout specification to place four interactive tasks with which the end-user can edit integer values, around a task that displays the sum of these values as a blue card, using edit\_card\textquoteleft. Analogous to customizing the card tasks, we introduce interactive tasks that edit a particular element of a list of values at some index location \(i\):

\[
\begin{align*}
\text{edit\_elt } i &= \text{updateInformation } ("edit_i", ++\ toString i) [upd\_elt i] \\
\text{edit\_elt'} i &= \text{updateSharedInformation } ("edit_i", ++\ toString i) [upd\_elt i] \\
\text{upd\_elt } i &= \text{UpdateWith } (\text{flip } (!!) i) (\text{flip } (\text{updateAt } i))
\end{align*}
\]

We define the task structure as follows:

\[
\text{task\_triptych :: (ReadWriteShared } [\text{Int}] [\text{Int}]) \rightarrow \text{Task } [\text{Int}]
\]

\[
\begin{align*}
\text{task\_triptych } sds &= \text{edit\_card'} \text{ sum } sds \\
&\mid [\text{anyTask } \text{edit\_elt'} i sds \backslash \backslash \ i \leftarrow [0 .. 3] ]
\end{align*}
\]

The desired task layout is obtained by adding the following layout annotation to the above expression:
6.6. THE LAYOUT OF TASK UIS

Figure 6.10: Screenshot of the triptych example in iTasks

withShared [1, 7, 4, 2] task_triptych <<@ TaskLayout my_layout

my_layout :: (UITree TaskUITag) -> TaskUILayout a
my_layout (UINode _ [c, UINode _ [a, b, d, e]])
  = triptych ([], [], []) (a', b', c', d', e')
where
  [a', b', c', d', e' : _] = map uiOf [a, b, c, d, e]

Figure 6.10 shows the resulting task UI layout.

To illustrate the flexibility of the approach, suppose that somebody wishes to exploit the following equivalence:

t -||- anyTask ts = anyTask [t:ts]

and thus alters task_triptych as follows:

task_triptych' :: (ReadWriteShared [Int] [Int]) -> Task [Int]
task_triptych' sds = anyTask [edit_card' sum sds : edit_elt' i sds \ i <- [0 .. 3]]

Although the programs are equivalent, their structure is different. We only need to alter the pattern match accordingly:

withShared [1,7,4,2] task_triptych' <<@ TaskLayout my_layout'

my_layout' :: (UITree TaskUITag) -> TaskUILayout a
my_layout' (UINode _ [c, a, b, d, e])
  = triptych ([], [], []) (a', b', c', d', e')
where
  [a', b', c', d', e' : _] = map uiOf [a, b, c, d, e]

to obtain the same desired layout of task UIs.

To illustrate a layout of task UIs that deploys the much richer language of dimensions, here is an example of an irregular layout that occurs often in the iTasks system itself (in Figure 6.11 a dashed line indicates a splitter).

my_layout (a,b,c,d,e)
  = beside [] [splitUISize (pct 15) (pct 85)] []
    [ above [] [defaultSize, splitUISize (pct 10) (pct 30)] []
      [a, b] (Just InHost)
    , above [] [fixUISize (pct 10), splitUISize (pct 20) (pct 40)] []
      [c, d, e] (Just InHost)
  ] (Just InHost)
UI element $b$ can be resized by the user between 10% and 30% of the height of the left column. The top bar, $c$ has a fixed height of 10% of the height of the right column. UI element $d$ can be resized by the user between 20% and 40% of the height of the right column.

Figure 6.11: An irregular layout of tasks $a \ldots e$

6.7 Analysis

In this section we justify the claim that we have proposed a “general purpose solution” for the challenges of specifying spatial layout of UI items, separating the concern of maintaining the life-cycle of UI items from their chosen layout, and identifying UI items.

In application domains in which every constructor function has the property that identical calls yield indistinguishable UI items, it is sufficient to instantiate the overloaded layout language. The application domains `Graphics.Scalable` (Section 6.4) and `ncurses` (Section 6.5) satisfy this property. In [6] we explain how `Graphics.Scalable` has been inspired by the mature Racket image API [50, 13]. For this application domain the overloaded layout language can also be instantiated. There is an interesting, and deliberate difference, between the ways image dimensions are handled. In the overloaded layout language, the class `DimRef` introduces a tagging system to identify images of which dimensions need to be found. In Racket, this is done more directly. Paraphrasing its `image-width` function signature in `Graphics.Scalable`:

\[
\text{image-width} :: \text{(Image m)} \rightarrow \text{Int}
\]

Such a function only makes sense in a context where image-generating functions yield indistinguishable images. An implementation of the `DimRef` member functions can use the images themselves for the tags.

In application domains that rely on action-based constructor functions, more effort is required to integrate it with the overloaded layout language. Virtually every widget-based library is action-based: the programmer is required to call handle-creating actions that have as immediate side-effect that the corresponding widget object is created. The returned handle is used later on in the program to alter its properties, such as dimensions, position, stacking order, visibility, accessibility, and finally, to delete it (manually, or via a finalizer mechanism). Instead
of immediately having a side-effect and create a widget, the implementation of such approaches should be altered to create an intermediate representation of the widget, or, if the back-end allows it, create an invisible widget. The intermediate representation can then be used afterwards to apply the layout language to. Once the layout has been computed, either the widgets can be created at the correct positions, sizes, and stacking order using the intermediate representation or made visible after settings its other properties.

In this context, Clean Object IO \[5, 7\] takes position between these two extremes: it offers both a declarative GUI representation language that can serve as an intermediate representation language and it offers an action-based API to create any of these GUI elements. The GUI representation language of Object IO uses a rather complicated layout language and can be replaced entirely by the overloaded layout language described in this chapter. Its action based functions can be altered as described above, allowing the code annotation to be introduced. The UI rose tree that is required is a rather straightforward projection of the hierarchical structure of the intermediate representation.

Finally, the iTasks case study demonstrates that the approach is applicable also for systems in which no handles, or similar values, are created.

The code annotation works for any application domain, regardless whether it deploys an existing layout strategy or none at all, leaving layout at the discretion of the UI constructor functions. When the implementation has been altered to a two-phase process as described above, it is clear which elements get influenced by the code annotation. For these elements, the new layout can be computed and protected against further manipulation by passing it to other code annotations or layout strategy as a UI rose leaf.

The example in Section 6.6.4 shows how separation of concerns is achieved in the solution. The code annotation works as a pivot. If the task structure is altered, only the UI rose tree pattern-match changes along, and the layout specification is untouched. If the layout specification is altered, only its call within the code annotation changes along, and the task structure is untouched. From this point of view, the approach shares the same advantages and disadvantages as standard pattern-matching in functional programming languages.

As shown in this chapter, layout specifications can be reused across multiple domains. This implies that the layouts that specify the relative position of their elements are portable. However, the effort required in porting an application to a different user interface back-end remains largely the same, since the reusable layout specifications need to be parametrised by domain-specific hints.

### 6.8 Related Work

In traditional, widget based, GUI libraries, the application code uses actions to generate one GUI component at a time, together with some kind of identification value that must be used to control the life cycle of that GUI component. Examples of such approaches are Haggis \[44\], TkGofer \[22\], wxHaskell \[67\]. In these approaches, GUI component creation and identification is not separated at all.
The identification values are required to control the layout of the elements. The layout language is the familiar set of structured layout, i.e., placing GUI components horizontally, vertically, and in a grid. In Clean Object IO, the creation of GUI components and their identification is turned around: pre-conceived identification values are used to identify GUI elements within a shallowly embedded DSL that describes entire, composite, GUI structures. This improves the separation of concerns. In all of these approaches, rules need to be defined when an identification value is used that does not (temporarily or forever) correspond with a GUI element. The relation between identification value and GUI component is fragile. The code annotation that we propose in this chapter does not suffer from this fragility, but it comes at the expense of introducing a dependency between the concrete task structure and the UI rose tree.

The seminal Fudgets [21] system used a purely combinator based approach to specify GUI applications in order to move away from the traditional widget abstraction towards a functional style of programming. A GUI component, fudget, is a stream processor that can be glued together with other GUI components to form a more complex stream processor, using combinators. A default layout algorithm takes care of placing the GUI components. The layout can be tuned at the fudget combinators when the default is not appropriate. Thus, this approach has the similar disadvantage as the old layout mechanism of iTasks, viz. that tuning the layout clutters the original fudget structure. The school of functional reactive programming, FRP, suffers from the same issue. FRP examples are Fran [39], FranTk [72], Fruit [24], and Yampa [25]. In FRP, a GUI component processes a signal, which is a continuous, time-varying value. GUI components are glued together to process more complex signals. Just like Fudgets, layout is specified at the combinator level, and therefore interferes with the original combinator structure.

iTasks is one of many systems that utilize the web infrastructure to create distributed, interactive applications. In Wash/CGI [102], the application developer uses a monadic abstraction to create forms and an identification value to access their content. Therefore, as with traditional widget based approaches, GUI component creation and identification are tightly coupled.

At a higher level of abstraction, we find Hop [93] and Links [23]. Hop uses a stratified approach, and offers two separate, cooperating, languages, one for the web server and one for the web clients. Links, as iTasks, uses a single language approach, but unlike iTasks, the application developer needs to use the keywords server and client to coordinate where which part of the application should be executed. Both approaches differ from iTasks, in which the entire program is compiled to target the server, using the ‘ordinary’ Clean code generator, and one to target the client, using a JavaScript compiler [33]. The designers of Hop and Links have seamlessly blended HTML with functional features to define behavior in a callback style, making it look familiar to developers who know their HTML. In addition, both approaches offer access to the HTML structure via DOM-manipulating functions, providing the developer with low-level access to the created GUI. Low-level access to the GUI can break the final user interface, is sensitive to changes of implementation, and should concern only the interactive task that creates the GUI. In the approach proposed in this chapter, the application domain determines which
abstraction barriers should be kept intact and can protect against this simply by offering a composite GUI as a leaf value in the UI rose tree.

6.9 Conclusions

In this chapter we have introduced an overloaded, general purpose, language of the layout of things. Instead of inventing yet another layout language, we have generalized the Graphics.Scalable image layout language. For some application domains, this suffices to specify the layout of its inhabitants. We have shown this for scalable vector graphics (Graphics.Scalable) and a terminal style GUI toolkit (ncurses). For application domains in which the inhabitants can only be found indirectly, or even do not exist, we have introduced a code annotation with which the application developer can pattern match the structure of UI that is generated, and define an appropriate layout. We have shown this for handling layout of arbitrarily complicated UI components (iTasks).

Separation of concerns is achieved in this way in iTasks. The task engineer can concentrate on defining the appropriate task structure, knowing that a default task layout is always available. She can replace ‘equals by equals’, as illustrated in Section 6.6.4 knowing that it is always possible to define a custom task layout via the code annotation. Tuning the layout of a task involves defining a pattern matching code to find the task UIs and creating a custom task UI layout. If the task structure is changed, then it is likely that the task tree pattern must be changed as well, so there is a price to pay. However, in this way fragile task references do not exist (see Section 6.8) and the task layout definition does not have to be changed if the collection of task UIs is the same.

In the current proposal we have not dealt with the possibility to introduce ‘harmless’ UI content such as frames to visually delimit parts of the user interface, or expressive labels to guide the application user, and so on. We conjecture that this can be dealt with via domain specific constructor functions in combination with the constructor functions for the host.

The long term vision is that in the specification of software the way to specify things and their layout are completely orthogonal issues. We think that this chapter is a first step towards this goal.
Chapter 7
Task-Oriented Software Development

When developing software, it is considered good practice to maintain a separation of concerns, e.g., separate control flow from business logic and presentation. Doing so is not always straight-forward, however. How separation of concerns can be achieved depends on the language and libraries used, as well as the type of application that needs to be built.

We are particularly interested in how to build Command & Control (C2) prototype applications for the Royal Netherlands Navy. C2 systems are multi-user systems that coordinate information and work between a multitude of people and machines. In the naval domain, the work that needs to be done can be complex and time-critical. The tools used to write such software must be sophisticated enough to support this complex domain. Such tools are more likely to do so if they support a separation of concerns.

Our programming paradigm of choice is Task-Oriented Programming (TOP), which is implemented by the iTasks framework in the purely functional programming language Clean. With TOP, one defines programs in terms of the tasks people and machines need to perform. Given this high-level program description, a TOP framework generates a working multi-user web-based application with which humans and machine can perform the tasks described in the system.

In this chapter we investigate whether a prototype C2 system for fire-fighting and damage control on board of a navy vessel can successfully be implemented in a structured way using TOP, while maintaining separation of concerns. Previous work by Carlson et al. [20] already shows how functional programming can be successfully applied to program single-user C2 applications. Additionally, the Incidone application [69] shows how TOP can be used to implement an incident response application using TOP. Neither work specifically promote separation of concerns, nor do they attempt to structure the way in which the applications are written.

With this work we identify the relationships between traditional functional programming and TOP. We also identify four distinct concerns that can be developed orthogonally when applying TOP. We combine these insights into a structured software development approach called Task-Oriented Software Development (TOSD). In a sense, TOSD can be seen as the art of developing TOP applications.

Our approach to developing TOSD has been mostly experimental. While developing the C2 prototype, we paid careful attention to which components we used, how they fit together, and whether they could evolve orthogonally. If certain aspects of the application could not be developed orthogonally, we modified iTasks to correct this. In addition to a working prototype C2 application, of which the source code is available, this resulted in a substantial amount of experience with structurally developing TOP applications.

This work shows that TOP can be successfully used for implementing applica-
tions for complex socio technical domains, and that the software is able to evolve, due to a separation of concerns. Additionally, this work also shows that software engineering practices have a place in functional programming as well.

7.1 Introduction

When developing complex software systems, developers strive for separation of concerns, allowing them to work on part of the system in isolation as much as possible. Separating individual concerns reduces the complexity of the development effort. How easy it is to maintain a separation of concerns depends on the complexity of the application one wants to develop, and the APIs and abstraction methods offered by the programming languages, libraries, and tools being used.

We are interested in developing software for the rather complex application domain of Command and Control (C2) systems. C2 systems are tools that enable gathering information from distributedly operating people, sensors, and other information sources. Their task is to inform and coordinate all participating parties in an optimal way such that they can accomplish their shared goals. For example, the Dutch Coast Guard uses C2 systems to coordinate Search and Rescue efforts on the North Sea. Here, Dutch and Belgian coast guards and navies may need to work together on the water, joined by medical personnel on shore. Another example is the Royal Netherlands Navy, which deploys C2 systems to support their missions anywhere on earth.

Since the application domain itself is complex, the software that supports the work people in the domain need to do is necessarily complex as well. To successfully develop C2 software, it is important that the various aspects of the applications can be developed and maintained mostly orthogonally, so that the development effort does not become too complex for the application programmers. The tools used to develop the C2 systems will need to meet two requirements: they must allow the most important concepts from the domain to be implemented naturally in the chosen language, libraries and frameworks, and they must enable a separation of concerns, enabling orthogonal development of various application aspects.

To fulfil the first requirement, we choose to apply the Task-Oriented Programming (TOP) paradigm. In TOP, tasks are the central notion with which one constructs programs. This concept corresponds naturally with the tasks people and machines need to perform in a distributed C2 setting. Everybody who uses the system, i.e. every user, needs to be constantly informed with up-to-date information about what is going on (situational awareness), such that, at each time, everybody can decide what is best to do next. Commonly, the actions of one user has consequences for what other users can see and do. Additionally, each user may be physically located in a different location, may be performing various tasks, using multiple systems, sensors, and platforms. At the same time users need to communicate and synchronize with each other, offering every user a suitable user-interface, and so on.

The TOP implementation we use is called iTasks, which is a shallowly embedded domain-specific language (eDSL) in the lazy, purely functional
programming language Clean. Clean is a strongly and statically type language that uses a Damas-Hindley-Milner based type system with full type inference. Functional programming is shown to be very well suited for modular programming, due to features such as higher-order functions. Using iTasks, one can quickly construct multi-user, distributed, web-based applications. As we shall see later, iTasks offers a separation of concerns, fulfilling the second requirement. We identify various concerns using a development approach called Task-Oriented Software Development (TOSD), which we introduce in this paper. In other words, we capture the art of constructing TOP programs into a structured format. By mindfully applying TOSD, separation of concerns follows.

We demonstrate how the C2 domain is naturally mapped to TOP and how applying TOSD enables a separation of concerns by presenting a prototype C2 system, which we have implemented in iTasks. This prototype serves as an illustration how multi-user, distributed, web-based applications can be defined incrementally in iTasks, while maintaining separation of concerns. The C2 prototype application we have made has two major features. Firstly, to facilitate quick experimentation with alternative ship lay-outs, it allows the user, e.g. a designer at a shipyard, to model the lay-out of a ship and location of its main systems in a graphical editor. Secondly, the application can be used to simulate fire-fighting and damage-control (FFDC) scenarios, both fully simulated and using human-in-the-loop testing. These scenarios provide insights in how the lay-out of a ship and the location of its systems influence the way calamities can be resolved.

The rest of this paper is structured as follows. First, Section 7.2 explains what TOP is, how it extends Traditional Functional Programming (TFP), and how applications are developed using TOSD. Next, Section 7.3 shows how TOSD is applied to develop our prototype C2 application. It shows the architecture of a TOP application and it highlights the high level of abstraction and the separation of concerns iTasks offers. Section 7.4 discusses related work. Finally, Section 7.5 discusses our results so far and concludes with some future work.

7.2 Task-Oriented Programming and Software Development

TOP is implemented as an eDSL in the purely functional programming language Clean. In Clean, the core concepts with which one programs are (pure) functions and algebraic data types. A pure function is a function that does not have any observable side-effects and which, given the same inputs, always produces the same result. In a purely functional programming language, functions are composed by applying them to values or other functions. Functions are first-class citizens and can be used as argument to another function, or be given as a result of a function.

TOP programs are specified in terms of tasks. Therefore TOP and C2 systems share the same conceptual level: which tasks need to be performed to achieve a goal, and by whom. Here and in the rest of the paper we use iTasks as TOP implementation of choice. We do so, because iTasks takes away a lot of manual work commonly involved in the development of web-applications, such as the handling
A major technical role of a C2 system is to coordinate I/O actions between its users. As such, one might not immediately consider a pure functional language as ideal candidate for defining C2 systems. However, iTasks appears to be well suited to implement a C2 (prototype) system, partly because of the fact that many technical details are hidden and large parts of an iTasks program is generated, and partly because of the high-level concepts TOP offers, which will be discussed below.

TOP with iTasks does not replace Traditional Functional Programming (TFP). We use Clean’s powerful language facilities to enhance TFP with TOP concepts. These concepts have a significant impact on the way and ease with which multi-user web-based applications can be developed. The table below relates the main concepts of TFP to the additional concepts offered by TOP. The concept in the right column augments the concept in the left column. Each of the concepts in the right column will be discussed shortly.

<table>
<thead>
<tr>
<th>TFP</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic Data Types</td>
<td>Shared Data Sources (SDSs)</td>
</tr>
<tr>
<td>Pure Functions</td>
<td>Tasks</td>
</tr>
<tr>
<td>Function Applications</td>
<td>Task Combinators</td>
</tr>
<tr>
<td>I/O (Monads)</td>
<td>Task Editors</td>
</tr>
</tbody>
</table>

The approach for developing TOP applications is described by a process we call Task-Oriented Software Development (TOSD). In TOSD we distinguish the following phases: (1) Domain Modelling using TFP, (2) Shared Data Modelling, (3) Task Modelling, and (4) User-Interface Customization. TOSD does not impose or assume a particular software engineering discipline, such as waterfall, agile, or iterative software development. It can therefore be used in any of these software development approaches. TOSD phases can be implemented in any order, although the rest of this paper will maintain the order presented above.

The TOSD phases can be graphically represented as shown in Figure 7.1. The foundation of TOP with TOSD is TFP. Without TFP, no TOP application can be built. Tasks and Shared Data build on the models and relations defined in TFP. Finally, when all other parts of the application are implemented, can we apply optional UI customization.

The rest of this section describes the individual TOSD steps in more detail.

### 7.2.1 Domain Modelling Using TFP

Developing a TOP program typically starts off the same way as any functional program. Algebraic data types capture the entities from the application domain, while pure functions over these types capture the relations between these entities.
However, for defining side-effects, I/O, interaction, user interfaces, communication, synchronization, and so on, TOP concepts are used instead of familiar TFP techniques.

The absence of side-effects in the resulting domain model ensures that this part of the code is easily testable and maintainable. Conventional functional programming best-practices, such as Test-Driven Development and Type-Driven Development, may be applied here at the programmer’s discretion.

### 7.2.2 Shared Data Modelling

In a C2 system, all participants need to be informed about the latest state of affairs. iTasks offers Shared Data Sources (SDSs, or shares) that share arbitrary information between the different distributively working parties. Shares use a publish-subscribe approach: whenever shared information is changed by someone, those parties who need to be informed are automatically updated. Shares are commonly defined globally, though they can be defined in any scope. Typically, once domain modelling has taken place, one decides which data should be made available as an SDS. Common Algebraic Data Types can be used to define the type of an SDS.

As will be explained below, tasks can inspect each others progress while their work takes place and can react based on their respective task values. However, tasks do not only depend on the status of other tasks, they are very likely to also depend on shared global data, such as the current time, the current value of sensors on some other system, information stored in data bases, files on file systems, and so on. SDSs are compositional in two directions: some SDS combinators aggregate SDSs, while others allow for projections of information. Not all changes might be relevant for a task, so the share projections can be used to specify which notifications are relevant and which are not.
CHAPTER 7. TASK-ORIENTED SOFTWARE DEVELOPMENT

7.2.3 Task Modelling

A task is a description of work someone or some device has to do. A TOP application is a task, which can in turn be composed of sub-tasks. The description of a task is declarative and focuses on what has to be done, hiding how it is all technically realized in the often complicated distributed architectural setting with many systems and many users.

iTasks Tasks

More formally, an iTasks task is a reactive, monadic function of type Task a. iTasks tasks can have arguments like any other function, and may be higher order: they can have functions and tasks as argument or result.

Task functions differ from ordinary functions because they are reactive, which means that a task may yield a value of type a that changes over time. A task value can be in one of three states: there can be no value, or it can have a value of type a which is either stable or unstable. When there is no value, the task does not have sensible information about its progress at that time. When a value is unstable, the task has sensible information about its progress, but the value may still change. Finally, when a value is stable, the work is either done, or a deadline has passed, and the task value no longer changes.

Task values can change over time due to the fact that some work is done by someone somewhere. Change can be caused by interactions performed by the end user to whom the task is assigned, new values produced by other tasks, or changes in some SDS which is relevant for the task being executed.

Task Combinators

Tasks can be described in terms of sub-tasks, and finally in basic tasks. Basic tasks are tasks which can e.g. interact with the user (called editors), access a database, read out a sensor, or execute a task on some remote computer. Tasks can be combined with task combinators. Most frequently used task combinators are defined in terms of two basic combinators: the sequential step combinator and the parallel combinator. They obtain their expressive power because one can observe and inspect the current values of the involved sub-tasks. One can then decide what the effect for the combined task will be. From these basic combinators, arbitrarily complex derived combinators can be constructed.

One can assign tasks to anyone with a particular role, to a specific user, or to some specific computer system. Commonly there are many tasks someone has to do, but one can decide to work on any of them, in any order.

After the tasks have been modelled, one obtains an executable iTasks application which can coordinate the tasks thus described.

Task Model Verification

During development, it is important that domain experts frequently verify that the correct software is being built. One way of doing that is by letting domain experts
use the application being developed in so-called human-in-the-loop testing. While this is an effective solution, it often takes up a lot of time and costs a lot of money.

Another approach is to use Tonic \[98, 99\], which stands for “Task-Oriented Notation Inferred from Code”. Tonic generates graphical representations, called blueprints, of the monadic structure of tasks. It utilizes the fact that TOP programs are specified on a high-level of abstraction, in terms of the tasks that need to be performed. The high level of abstraction and the task-based vocabulary used to define the tasks in the program result in blueprints that, with a little help from a programmer, are understandable even to people without a technical background. Verification can now be sped up by having domain experts review blueprints, rather than perform lengthy human-in-the-loop tests. Section 7.3.2 shows an example of a blueprint.

### 7.2.4 User-Interface Customization

For any first order type \(a\), a web-based interactive editor can be generated with which the user can construct a proper value of the demanded type. Hence, in iTasks, one gets user-interfaces for free. Generating user interfaces aids rapid prototyping tremendously. The web-based user interfaces work in any HTML5 compatible browser.

Automatically generated user interfaces may not always have the right look and feel, however. Any user interface can therefore be customized. In the simplest case, a type is simply transformed to another type, after which the generic machinery generates another generic user interface. A more sophisticated customization can be achieved defining custom client-side editors using editlets \[^3^4\]. iTasks offers a pre-made editlet which can render a value using scalable vector graphics (SVG) \[^2^7\] using the Graphics.Scalable library \[^6\]. The use of this editlet is shown in Section 7.3.1.

iTasks uses a default layout algorithm to place the generated user interfaces of sub-tasks in order to produce the assembled user interface. The programmer can customize the layout of these task user interfaces to obtain the demanded lay-out of the interface components.

### 7.3 Implementing a C2 System Using TOSD

In this section we apply TOSD to implement a part of a C2 system and highlight some of its more interesting implementation aspects. We base the requirements for this C2 system on a case study that we have defined together with the Netherlands Defence Academy (NLDA).

Our focus for this demonstration is on fire fighting and damage control (FFDC) on board of a navy vessel. Particularly in a combat situation, a ship and its crew can take damage. A grenade may impact the ship, causing a fire in one of the compartments. Damage may impact a ship’s ability to achieve its mission. This damage may need to be mitigated before a ship can work towards achieving its mission again. To reason about these things, we need to be able to model a ship,
CHAPTER 7. TASK-ORIENTED SOFTWARE DEVELOPMENT

its crew, and its systems. Additionally, we need to define what a mission is and how the ship and its crew are related to it.

First, Section 7.3.1 shows how we represent a ship in terms of data types and functions, and how we have developed a graphical user interface to construct a ship model. In presenting the functions we often omit their implementation and only present their types. We explicitly follow the four TOSD phases in order. By doing so, we can clearly see how we maintain a separation of concerns, since each TOSD phase represents a different aspect of the application. Section 7.3.2 shows how the ship designed with the design tool can be used to simulate a fire fighting and damage control (FFDC) exercise. The full application contains many more features than we are able to cover in this paper. Section 7.3.3 summarizes the application’s remaining features.

7.3.1 Modelling a Ship

In this subsection we model the ship. The name of each TOSD phase is printed in bold letters at the start of that particular phase.

Domain Modelling

On board of a navy vessel, people and computers need to work together to solve time-critical problems. These processes rely on information sources, such as the ship’s mission, physical ship properties such as the layout of the ship and systems placement, and the state of the ship’s sensors. In order to explore these interactions, we need to be able to model a ship in which they take place. To do so, we introduce several new algebraic data types and record types:

```haskell
:: Decks ::= [Deck]
:: Deck = { deckId :: String
           , sections :: [[Section]]
           , deckSize :: (Real, Real) }
:: Section = { description :: String
             , borders :: Borders
             , hops :: [Coord3D] }
:: Borders = { north :: Border
              , east :: Border
              , south :: Border
              , west :: Border }
:: Border = Open | Door | Wall
:: Coord2D = { col :: Int
              , row :: Int }
:: DecksLevel ::= Int
:: Coord3D ::= (DecksLevel, Coord2D)
:: Distance ::= Int

decksToGraph :: Decks -> Graph
shortestPath :: Coord3D -> Coord3D -> Graph -> Maybe ([Coord3D], Distance)
```
7.3. IMPLEMENTING A C2 SYSTEM USING TOSD

The top level type `Decks` is a type synonym for a list of `Decks`. It represents an ordered list of two-dimensional maps of type `Deck`. Decks placed towards the beginning of the list are assumed to be physically situated above the decks placed towards the end of the list. A `Deck` is implemented as a record type. A record can have one or more record fields, each of which have a specified type. At its conceptual core, a `Deck` is a list of lists (essentially a grid) of `Sections`, represented by a list of lists of `Sections`. Additionally, a `Deck` has a unique identifier `deckId` and virtual dimensions `deckSize`.

A `Section` is a rectangular area with borders on each side. A border, represented by the algebraic data type `Border`, can be either an open border, allowing passage to another section, a door, which can potentially be locked, and a wall, preventing passage to the adjacent section. A `Section` has an optional description. From each section, there is a possibility to go up or down a staircase to another deck. This is represented by the `hops` field, which contains a list of coordinates, pointing towards the target sections. A `Section` is identified by a `Coord3D`. A `Coord3D` is a tuple of an index (pointing to a `Deck` in the `Decks` list) and a column and row index, pointing to a `Section` in the `sections` grid. This way, each `Section` can be uniquely identified.

One of the features the C2 application has is a shortest path algorithm with which one can calculate the shortest route between any two `Coord3Ds` on board. This is useful for, amongst other things, determining where fire fighting equipment should be placed. The algorithm is implemented in plain Clean. The function `decksToGraph` converts the `Decks` to a `Graph`, after which the function `shortestPath` calculates a shortest path between two coordinates.

Shared Data Modelling

The `Decks` structure is central to the application, as it represents the ship in which all action takes place. With the ship editor application, a user can create and edit a `Decks` structure. The C2 application reads the structure. Any change made with the ship editor should immediately be reflected in the rest of the application. To enable this, and to store the ship model in a central place, we create an SDS for the map structure. Shares in iTasks are represented by the following types:

```clean
c:: ReadWriteShared r w = ... // rest of implementation omitted for brevity
c:: Shared a := ReadWriteShared a a

sharedStore :: String -> a -> Shared a | iTask a
```

The `ReadWriteShared` type takes two type parameters: a read type and a write type. These types need not be the same, although in practice they often are. To simplify using the same read and write types, iTasks offers the `Shared` type synonym. The `sharedStore` function creates a new SDS of type `Shared a`. It has two parameters: a unique identifier of type `String` and a default value for the share of type `a`. This default value cannot be just any value. Values of the type `a` can only be stored in a share if an instance of `type class iTask` exists for them, which is denoted in the type signature of `sharedStore` with `| iTask a`. 
Instances of this class can be derived automatically for all first-order Clean types. Now we can define a custom share to share the Decks data structure:

```haskell
decksShare :: Shared Decks
decksShare = sharedStore "decksShare" []
```

Decks is a list of Deck, so a sensible default value is an empty list. Now we can read and write a Decks structure from anywhere in the application. By default, sharedStore stores data on-disk as serialized JSON. This can be changed to in-memory storage or storage in some external system, such as a relational database.

**Task Modelling**

Modelling a ship by manually creating instances of data types is a tedious and error-prone job. A shared editor task can generically generate a graphical user interface that allows us to edit the Decks data structure directly in the decksShare share. iTasks provides several editors, among which the updateSharedInformation editor:

```haskell
updateSharedInformation :: String -> [UpdateOption r w] -> ReadWriteShared r w
                         -> Task r | iTask r & iTask w
```

This editor enables the user to directly update the information in the share specified in the third parameter by means of a generically generated graphical user interface. Its first argument is a title that can be displayed on top of the generated UI. Its second argument allows the programmer to customize the way the data from the share is displayed. More on this later. We can now define an editor for updating the decks in decksShare.

```haskell
editDecks :: Task Decks
editDecks = updateSharedInformation "Edit Map" [] decksShare
```

Running the editDecks task, we get user interface as shown in Figure 7.2.

![Figure 7.2: Generic editor user interface.](image)
Customization

While this generic editor makes it easier for a human to construct a value of type `Decks`, it is not a convenient way to construct an entire ship model. With the second argument to `updateSharedInformation` we can customize the generic editor in order to make it more user-friendly.

```hs
:: UpdateOption r v = E.v: UpdateAs (r -> v) (r -> v -> w) & iTask v
  | E.v: UpdateUsing (r -> v) (r -> v -> w) (Editor v) & iTask v
```

Both `UpdateOption` constructors take functions to convert from the read type `r` to a view type `v` and back to a write type `w`. The type specification shows that the view type `v` can be any type for which the `iTask` type class is instantiated. Additionally, the `UpdateUsing` constructor allows the user to specify a custom editor representation using the `Editor` type, of which we omit the implementation. For the ship editor, we want to render an interactive graphical representation of the ship. For this, iTasks supports drawing with SVG using the `SVGEdition` type.

```hs
:: SVGEdition v c = { renderImage :: v -> c -> Image c,
  ... // rest of implementation omitted for brevity
}
fromSVGEdition :: SVGEdition v c -> Editor v | iTask v
```

The `SVGEdition` record contains, amongst other fields, the `renderImage` field. This field contains a function that, given a view `v` and model value `c` creates an SVG image of type `Image c`, using the `Graphics.Scalable` library. The `fromSVGEdition` function then takes the `SVGEdition` and converts it into a general `Editor` type, which we can use in the `UpdateUsing` constructor. Putting this all together, we arrive at a new `editDecks` function:

```hs
id :: a -> a
const :: a -> b -> a
editDecks :: Task Decks
editDecks = updateSharedInformation "Edit_map"
  [UpdateUsing id (const id) imageEditor] decksShare
where
  imageEditor = fromSVGEdition { renderImage = \_ maps2D -> shipImage maps2D
  , ... // rest of implementation omitted for brevity
}
shipImage :: Decks -> Image Decks
```

In this particular case, the types of `v` and `c` to `SVGEdition` are the same. Two auxiliary functions are used: `id` and `const`. The former takes one argument and returns it as-is, while the latter takes two arguments, ignores the second, and returns the first. We can now render the ship as shown in Figure 7.3.

Using this approach, we can create a graphical ship modelling tool that allows everyone to construct a crude ship model. The screenshots shown in Figure 7.4 and Figure 7.5 show this tool in action. In the bar on the left side of the screen...
a user can specify the number of decks and the ship’s dimensions. Each deck is divided into a grid. Each grid cell wall can be absent, solid, or solid with a door. A user need only click on a wall to cycle between those three states. Any change made in this editor view is immediately propagated to the rest of the application via the relevant SDSs, including the possibly already running FFDC simulation.

Sections can contain items, devices, and pipes or cables. A fire extinguisher (Ex) is an example of an item. A radar (Ra), a power generator (Po), and a cooling pump (Co) are examples of devices. Finally, there are cooling water pipes (−Co−) and power cables (−Po−). The jagged triangles are stairs to another deck.

All of these objects play a role in the FFDC simulation. Actors that have been tasked with extinguishing a fire will need to find a fire extinguisher first, pick it up, and then find a way to a fire before extinguishing it. Devices in the real world generally need power to operate. On a ship, many devices, such as radar or a power generator, also need an active cooling mechanism. Together, all of these dependencies form a network of devices with possibly cyclic dependencies. For example, the power generator requires cooling, while the cooling system requires power. Power or cooling water is transported via power cables and cooling pipes. The ship editor allows the user to make the systems and the connections between the systems explicit, creating a graph in which devices are nodes and cables and pipes edges. With this graph of devices, we can reason about the effects of devices or cables/pipes being disabled, for example when they are destroyed by a fire. In turn, we can reason about the effects this has on the ship’s mission.
Discussion

In this subsection we applied the TOSD process to develop a prototype for an interactive ship design tool. Each TOSD phase could be implemented separately due to iTasks’ separation of concerns. This separation allows developers to focus on one aspect of the application at a time, reducing the cognitive load of developing an application.

7.3.2 FFDC Simulation

Now that we have a ship model, we want to be able to use it to simulate a fire fighting and damage control (FFDC) scenario. In case a ship is hit by a grenade, the ship may be damaged and there may be fires on board. Should the automatic fire suppression system be out of service, fires need to be put out manually. Doing so requires personnel, which may be already occupied with other tasks. The officer in charge of these matters, the Damage Control Officer (DC-Off), needs to decide whether to allocate resources, such as personnel and fire fighting equipment, towards putting out a fire, or to let that fire burn out. In case of multiple fires, the DC-Off also needs to decide which of these to put out first. These decisions can be complicated by a lack of resources. For example, all fire fighting personnel (a resource to the DC-Off) may already be busy fighting other fires, or doing other important tasks. Additionally, the contents of the rooms that are on fire or the room that are near the fire may influence the decisions the DC-Off makes. For example, there may be explosives, such as ammunition or fuel, or critical systems, such as such as a radar or cooling station, or important pipes and cables in the room. A defect in one of those systems or cables may impact the mission.

Conventionally, reasoning about these problems is done based on a DC-Off’s experience alone. No automated reasoning tools of any kind are available to support the DC-Off in the decision making process. The C2 application presented in
this paper aims to support the DC-Off in this process. To achieve this, we need the
ship model developed in the ship editor, and live information about where people
are on board and where damage has been detected. Most of this information is
currently not available in an automated way.

In this subsection, we see part of an application that supports the DC-Off
in dealing with damage on board. Again we will apply the TOSD approach by
focusing on a single aspect of the application at a time. However, instead of
customizing the rendering of a single editor, we show how to customize the layout
of a composed set of tasks.

In our scenario, actors walk around inside the ship until they are given a task
by the DC-Off. While walking around, they may encounter items, such as fire
extinguishers, which they may pick up. Items that have been picked up can be
dropped at any time.

Domain Modelling

An actor is uniquely identified by a value of type User, which is a type defined by
iTasks. Each section of the ship may contain zero or more users. We represent this
as the SectionUsersMap, which is a Map key-value data structure from a unique
section identifier Coord3D to a list of users [User].

:: SectionUsersMap :== Map Coord3D [User]

Shared Data Modelling

Users can walk around inside the ship, so their location may change at any time.
Such a change needs to be reflected in the rest of the application, so that the user
interface may be updated accordingly. We use an SDS sectionUsersShare to
model this mutability. By default, it contains an empty map:

sectionUsersShare :: Shared SectionUsersMap
sectionUsersShare = sharedStore "sectionUsersShare" newMap

Task Modelling

Now that we can keep track of the users on board, we can define a task that
allows a user to walk around in the ship. Once a user is logged in, the walkAround
task is started for that user. The walkAround task presents the user with a user
interface containing a graphical rendering of the current deck the user is on and
four buttons allowing the user to walk around on this deck.

walkAround :: User -> Task ()
walkAround user = watch (sectionUsersShare |*| decksShare)
   -|| viewDeck user
   >>> [ OnAction (Action "Go_west") (hasValue (moveTo West))
        , OnAction (Action "Go_north") (hasValue (moveTo North))
        , OnAction (Action "Go_south") (hasValue (moveTo South))
        , OnAction (Action "Go_east") (hasValue (moveTo East)) ]
7.3. IMPLEMENTING A C2 SYSTEM USING TOSD

where
moveTo :: Direction -> (SectionUsersMap, Decks) -> Task ()
viewDeck :: User -> Task ()

For brevity, we only provide the types for `moveTo` and `viewDeck`. Several new task and share combinators are used in the `walkAround` task. Starting on line 2, we see the task combinator `watch` and the infix share product combinator `|*|`, both provided by iTasks.

Given a share, `watch` creates a task of which the task value is the latest value in this share. Whenever the share is updated, the task value is updated as well. In this case, `watch` observes a composition of two shares, `sectionUsersShare` and `decksShare`. The `|*|` combinator combines two shares into a new read-only share, of which the read type is a product of the read types of the individual shares. Now, whenever either share is updated, the product of the share is updated, triggering an update in the task value of the `watch` task.

On line 3 we see the `viewDeck` task, which, given a user, renders the deck on which this user is located. The `watch` task is composed in parallel with the `viewDeck` task, using the parallel `-||` combinator, also provided by iTasks:

This combinator takes two tasks and executes them in parallel. The task value of the left-hand side task, the `watch` task in this case, is returned as task value of the composition. This value is then observed by iTasks’ step combinator `(>>*)`:

The step combinator observes the task in its left-hand side argument and reacts to either its value, an action that has taken place, or an exception that has been thrown. How the program reacts to these events is defined in the list of task continuations of type `TaskCont`. A `TaskCont` can react in two ways. Firstly, a continuation may be triggered by the `task value` of the left-hand side task. This case is covered by the `OnValue` constructor, which takes a function as argument that takes a `TaskValue` and optionally produces a new task. Whenever a new task is returned in the `OnValue` case, the step combinator as a whole is replaced by that task. Secondly, a continuation may be triggered by an `action` like clicking a button or menu item. Each `OnAction` creates a button in the user interface. Whether it is enabled or not depends on whether the corresponding continuation returns a `Just` or a `Nothing` value. When the button is clicked, the task in the continuation replaces the entire step again and is executed.
In the example, the left-hand side argument to the step combinator is the composition of the watch and viewDeck tasks. Due to the use of the (||-) combinator, the step continuously observes the value of the watch task only. It therefore is the task value of watch that is observed in the step’s continuations. This task value includes the most recent SectionUsersMap and the most recent Decks. We have added four buttons, one for each compass direction, with the OnAction combinator. The moveTo function determines whether a button is enabled or not by checking the map whether there is a wall in the way or not. If the button is clicked, moveTo updates the sectionUsersShare, causing the user to move.

We can graphically verify the task definition using Tonic. The blueprint for walkAround is shown in Figure 7.6.

Each rectangle with round corners represents a task. The name of the task is printed in bold in the upper part of the rectangle. Any arguments that a task may have are listed in the lower half. Blueprints are read from left to right. In this example, the first task is a parallel task that performs two tasks in parallel. It is a parallel task with a left bias, meaning that only the value of the left task is returned. This is graphically represented by a line that continues to the right side of the parallel box for the watch task, while there is no such line for the viewDeck task. The diamonds can be seen as guard for a particular execution branch. In this case, all branches become accessible when the parallel task has either a stable or an unstable value. When the parallel task has a value, the buttons are enabled. The dashed rectangles with user icon and text describe actions the user of the application can perform. Once a button is pressed, the corresponding moveTo task is executed.

In a similar fashion we can define tasks for changing decks by walking up or down stairs, or picking up or dropping items, such as fire extinguishers. We can combine all of these tasks into one task moveUser, which combines all the aforementioned tasks onto one screen:

```haskell
moveUser :: User -> Task ()
moveUser user = forever ( walkAround -||- changeDecks
                           -||- pickUpItems -||- dropItems)
```
Here we again see two new task combinators. The parallel \((-||-)\) is a variant of the \(-|\-\) task, which returns either the left or the right task’s value, depending on which task has finished first. We use this combinator to combine four tasks into one. The \texttt{forever} task combinator, also provided by iTasks, restarts this combination of tasks whenever either of the inner tasks has finished. This way, we get a continuously present user interface, shown in Figure 7.7. The green square with an A represents user Alice walking around in the ship.

![Figure 7.7: Alice walking around on the ship. Without proper task layout.](image)

### Customization

Despite the custom SVG rendering, the layout of the user interface is still very generic. The individual tasks in the parallel composition are rendered underneath each other. What we want instead is to apply a custom layout. In this custom layout, the user interfaces of the \texttt{changeDecks}, \texttt{pickUpItems}, and \texttt{dropItems} tasks are placed next to each other. The user interface of the \texttt{walkAround} task is placed above the other task interfaces. Task layout is a separate concern from the logical composition of tasks, so we want to be able to define it in a way that does not require us to modify the task structure at all. iTasks gives us this possibility with the \texttt{tune combinator} (\texttt{<<<@}), as shown below:

```haskell
instance tune Layout where
  (<<<@) infixl 2 :: Task a -> Layout -> Task a

modifyUI :: (UIRef -> UILayout) -> Layout

moveUser :: User -> Task ()
moveUser user
  = forever (walkAround -||- changeDecks -||- pickUpItems -||- dropItems)
    <<<@ modifyUI moveUserUI

where
moveUserUI :: UIRef -> UILayout
moveUserUI (Par _ [ walkAroundUI
                                 , Par _ [changeDecksUI, Par _ [pickUpUI, dropUI]]])
```

153
Using the tune combinator and the modifyUI function, we provide a layout via the moveUserUI function. This function receives a rose tree mimicking the task structure of the task the tune combinator is applied to. We can now pattern match on this structure to identify references the user interfaces of the individual task in the parallel composition. These individual references, which have type UIRef, can now be restructured using a couple of pre-made combinators:

\[
\begin{align*}
\text{uiBeside} & \colon [\text{UILayout}] \to \text{UILayout} \\
\text{uiAbove} & \colon [\text{UILayout}] \to \text{UILayout} \\
\text{uiOf} & \colon \text{UIRef} \to \text{UILayout}
\end{align*}
\]

Given a list of layouts, the uiBeside and uiAbove combinators arrange the layouts horizontally or vertically, respectively. The uiOf function dereferences a UIRef and turns it into a concrete part of the layout. Any UIRef not used is not included in the final user interface. The final result of this layout is shown in Figure 7.8.

![Figure 7.8: Alice walking around on the ship. With proper task layout.](image)

**Discussion**

In this subsection we applied the TOSD process to develop a user interface with which one can walk around on the ship designed in the ship editor tool. Again we show that each TOSD phase can be implemented separately. We showed that the customization phase is not only limited to editors’ user interfaces, but extends to the composition of parallel tasks as well, all while maintaining the separation of concerns.

### 7.3.3 Other Application Features

So far we have only shown a small part of the C2 application. The full application has many more features, which are summarized here.

A ship operates based on what its mission is. For the ship’s commander, the C2 application has an interface in which the commander can manage the ship’s mission. A mission is modelled as a descriptive text and a set of capabilities that are required to execute that mission. For example, the mission “patrol coast” may require the capabilities “surface sensors” and “radio communication”. We have provided a pre-made list with such capabilities. Each of these capabilities have a set of devices associated with them.
that can be used to support that capability. For example, the capability “surface sensors” can be executed by both a radar and a camera system.

As mentioned in Section 7.3.1, the design tool has the ability to add devices to the ship and connect them with cables. When a ship is hit with a grenade, fire can break out. The application features a “kitchen” mode in which a person can start fires anywhere in a ship. When a fire detector device is present in a room with a fire, the DC-Off is automatically notified of the presence and location of that fire.

In order to aid the DC-Off in deciding which fires to fight and in which order, we have included a damage prediction mode. With this interface, the DC-Off can reason about the consequences of a room being completely disabled by e.g. a fire. The system uses the network of cables and devices to determine which devices can no longer operate if the cables or devices in the selected room are destroyed. In turn, this information is used to determine which mission elements are imperilled by the disabled devices.

After selecting a fire that needs to be put out, the DC-Off needs to choose someone to actually put out the fire. The application shows an overview of the people on board. It also shows how close they are to a fire, considering they need to fetch an extinguisher first. Next, the DC-Off needs to select someone and give that person the order to go and fight the fire.

One way to do so is to assign that task to a human. This person can use a graphical user interface to walk around in the ship using the moveUser function from Section 7.3.2. The user needs to manually find a fire extinguisher, then walk to a fire, and finally use the extinguisher to put out the fire.

A second way is to simulate a person using a rudimentary agent. The agent can ask the ship’s computer to automatically determine the shortest over-all route to the fire that goes via an extinguisher. It then proceeds to automatically grabbing that extinguisher and putting out the fire.

The third and last way is to use a script to direct an agent. Using a script editor, a sequence of commands can be constructed which an agent executes in the specified order. For example, “go to nearest (object)”, “take (object)”, “go to location (x)”, “use (object)”.

In real life, testing an application with all human actors is time and resource consuming and should be reserved for a relatively stable and well-designed product. The ability to use agents instead of human actors increases the developer’s ability to explore the domain’s design space quickly and cheaply.

7.4 Related Work

Commercial C2 systems

Various commercial C2 systems are offered by well-known companies, such as Lockheed Martin and Saab. In contrast to these systems, we cannot support real-life military operations. Our strength lies in our ability to quickly prototype new ideas in the C2 domain, such that they may be included in mature C2 systems in the future.

\[http://www.lockheedmartin.co.uk/us/products/air-defense-C2.html\]
\[http://saab.com/region/saab-australia/land/c4i1/tactical-c2-systems2/\]
CHAPTER 7. TASK-ORIENTED SOFTWARE DEVELOPMENT

Using Functional Programming for C2 Systems

Carlson et al. [20] have used Haskell to implement geometric region servers for navy command and control. They conclude that Haskell is well-suited for developing navy C2 systems, because prototypes cost significantly less time to develop. Their results are particularly encouraging, because they draw these conclusions even without the use of generic programming techniques to generate large parts of their application, which TOP does employ. Another significant difference between their prototypes and ours is that they mainly do domain modelling. They do not provide a graphical user interface, let alone a multi-user web-based one.

Coast Guard Crisis Management System

In previous work [69] we have implemented a crisis management system for the Dutch Coast Guard in iTasks. This application, called Incidone, allows for managing incidents that occur in Dutch coastal waters. It allows for an operator to coordinate the efforts of search-and-rescue parties and medical services, among other things.

While this application did successfully demonstrate that Task-Oriented Programming can be used for developing complex multi-user applications, it left much room for improvement in the Customization Phase. Layout could only be specified at a parallel task combinator. Layout instructions were therefore mixed with the program’s task structure, violating the idea of separation of concerns. These limitations were an important driving force behind the development of TOSD.

Functional Reactive Programming

Functional Reactive Programming (FRP) was first introduced by Elliott and Hudak [40]. FRP has two central notions: behaviours and events. Behaviours are values that change continuously over time. Events are discrete. They can be real-world events, like a mouse click, or something more abstract like a predicate.

While semantically behaviours are defined over time, existing practical FRP libraries do not always make time explicit. With this practical relaxation of what it means for something to be a behaviour, an SDS can also be seen as a behaviour. Reacting to changes in the behaviour can be defined in the step combinator. The left-hand side of the step would need to include a watch, updateSharedInformation, or a similar task of which the task value is updated when the SDS’s value changes. An OnAction rule in the right-hand side of the step combinator can then be seen as a reactive event. Once one of the step’s branches is entered, the observed task is be discarded and the reactive behaviour ceases.

Elm [26], a well-known web application framework, was initially promoted as a practical implementation of FRP. Recently however, they abandoned FRP in favour of a concept called subscriptions. WebSharper [16, 46] is another example of a reactive web framework. Its offers a more traditional MVC-style framework for web application programming, whereas TOP operates at the higher abstraction level of tasks.

Model-View-Controller

Model-View-Controller [65] (MVC) is a popular approach for implementing separation of concerns. Models represent the business logic in an application, views commonly

http://elm-lang.org/blog/farewell-to-frp
represent a graphical user interface with which the user interacts, while the controller mediates between those two to keep them separated. In TOP and TOSD, both model and view can be readily recognised. The result of the Domain Modelling phase can be seen as the application’s model, while the optionally customized GUI can be seen as the application’s view. Identifying the controller is less obvious. In a sense, editors can be seen as generic controllers that automatically mediate between model and view. We choose to not describe TOP as an MVC framework, because one does not implement controllers whilst doing Task-Oriented Programming.

7.5 Conclusion and Future Work

We have shown a systematic approach, called TOSD, to developing a complex, multi-user TOP program. TOSD is enabled by the fact that we can neatly separate various aspects of an application. This same separation of concerns also enables a flexible approach to applying TOSD. The individual phases do not have to be applied in a fixed order. As a consequence, the TOSD approach, and by extension TOP, can be used in any software-engineering approach, be it agile or a more traditional waterfall.

Section 7.3.1 and Section 7.3.3 discuss the ability to place and connect devices, and then reason about the impact of damage on the availability of these devices. This ability is still in an early form. More sophisticated reasoning that includes support for redundant systems and reasoning about amounts of resources is required to make this feature more realistic and usable in a real-life situation. Additionally, a fire in a room may have consequences outside of that room as well. An adjacent room may contain ammunition or other explosives, for example, increasing the priority of putting out the fire quickly. We want to add this kind of information to the application as well.

In collaboration with the Royal Netherlands Navy, we aim to interface the prototype C2 application with real ship’s systems. Doing so allows us to access real sensor data and reason about real-life FFDC scenarios. This can make the application usable in real-life FFDC drills.

TOSD can currently only be applied by manually ensuring that the various concerns are separated. We are currently studying context-aware IDEs, which could aid in structurally applying TOSD during task-oriented programming.
Chapter 8
Conclusions

For this thesis we set out to create a prototype of a part of a C2 system using a wide array of functional programming tools. At the centre of our tool set sits iTasks, an implementation of the TOP paradigm in Clean. Along the way towards the C2 prototype, we have significantly expanded and improved our collection of tools.

Tonic, which implements the concept of blueprints, is the most significant new tool in our tool set. Blueprints are graphical representations of the monadic structure of code. We have seen how to generate both static and dynamic blueprints for a large class of programs and which challenges are encountered in the process.

In the process of learning how to draw the blueprints we generate from code, we developed a novel way to draw Scalable Vector Graphics in a web browser. From this graphics language we later extracted a powerful layout language that can be used in multiple different domains, including the layout of tasks. With this new layout language, we were able to maintain separation of concerns in the implementation of iTasks programs.

In this chapter we first evaluate several aspects of blueprints in Section 8.1. We define more clearly what blueprints are not, after which we make a first attempt at validating the usefulness of blueprints. Next, we give an overview of the potential future applications of blueprints in Section 8.2. We then discuss graphics and layout in Section 8.3, after which we discuss separation of concerns in Section 8.4 and applications in C2 in Section 8.5. Finally, we wrap up in Section 8.6.

8.1 Blueprints

In this section we evaluate the concept of blueprints. It is important to realise what blueprints are not, namely a way to do visual programming. Next, we reflect on to whether blueprints succeed in their aim to improve understanding of monadic code. Lastly, we look at how blueprints can be used in Haskell.

8.1.1 Generating Blueprints

Chapter 5 showed how blueprints can be generalized to visualize any monadic program. To do so for static blueprints, a notion of sequential composability of computations is all that is needed, which is what is provided by the monadic bind combinator. We can generate static blueprints for all monadic programs. This is even true for types that have been implemented as a monad, but do not adhere to the monad laws.

Dynamic blueprints are not so easily generalized in a lazy language, however. The order in which blueprint nodes are highlighted, must match the intuitive notion of “the right order”. I.e., a bind’s left-hand computation must intuitively be highlighted before its right-hand side computation. Highlighting only happens when a computation is fully evaluated, so for successful highlighting, the left-hand side must be evaluated before the right-hand side. This is not always the case in a lazy language. In fact, a bind’s arguments do not necessarily need to be evaluated completely at all. After the evaluation of a bind,
the only thing we know for certain is that the bind’s left-hand side argument is reduced to weak head normal form (WHNF). We also know that if a bind’s right-hand side argument is (partially) evaluated, the left-hand side argument has to have been reduced to WHNF first. Hence, the blueprint arrows signify the order in which monadic computations are reduced to WHNF.

Due to lazy evaluation, the order of evaluation is in general hard to predict. Compiler transformations may influence the evaluation order as well, complicating the situation further. To ensure that monadic computations are reduced in a particular order, it is necessary to thread a state through the monad. In other words, state monads can successfully be visualized dynamically. This is because a computation in a sequence of state-monadic computations requires the state from the previous computation as input, forcing its evaluation first before being evaluated itself. Indeed, both the Task and the IO monads are state monads and we have seen that both can successfully be dynamically visualized.

8.1.2 What Blueprints are Not: Visual Programming

Visual programming is programming by using graphical elements as building blocks. One way to implement visual programming is by offering a completely graphical language (e.g. Scratch). Another way is to offer a graphical modelling formalism from which high-level code is generated (e.g. UML).

Visual programming is often proposed as a way to enable people without programming experience to code. A common sentiment seems to be that one need only find the right graphical syntax for this to succeed. This notion disregards the fact that programming is an inherently creative process. Even with the help of graphical formalisms, programming still requires one to be able to think computationally and algorithmically. No graphical formalism will instantaneously teach someone to do so.

Tonic is explicitly not aimed at being a visual programming language. With Tonic, code is written first, and only then are blueprints generated. With this approach, textual programming and visualization are used in synergy, as also proposed by e.g. Petre. We do hope that graphical formalisms such as Tonic will lower the barrier for people to take up programming, and possibly enable them to learn faster by being given visual feedback.

8.1.3 Validating Blueprints

In the Tonic papers, we suggest that Tonic’s blueprints may make it easier to communicate with domain experts, because the images are more intuitive and less intimidating to them than Clean code. While we have not performed a formal study with domain experts to verify whether this is actually true, we did perform a case study to find out how Tonic is received by bachelor students in the second part of an introductory functional programming course. Like domain experts, these students have little to no experience with monadic programming, although they have been exposed to basic functional programming concepts in the first part of the course. During this course, the students are introduced to the concept of monads for the first time. Traditionally, the students find monads one of the hardest concepts in this course. We wanted to find out whether Tonic could help with learning about monads, because blueprints could potentially aid visual learners in understanding the course contents better.

Blueprints had a mostly positive impact. In general, students indicated that blueprints helped them to understand both the slides and the exercises. The students found
blueprints mostly easy to understand and they indicated that blueprints can help them to create a mental model about the execution of monadic code. This is reflected in the exam results, which for the monad-related questions were higher than the previous years results, but not significantly so.

From these results we can conclude that it is worthwhile to experiment with using Tonic in education again. The results of this study do not directly translate to the case of the domain expert that needs to be involved in the development process of a real software product. Still, since a group of bachelor students and non-technical domain experts have at least in common that most of them do not know monads, these results do suggest a reasonable chance for a favourable review for Tonic when used with a domain expert.

Future research can study how blueprints are received by domain experts when applied in the context of software development projects. Additionally, such research could identify ways to improve blueprints to the extent that they measurably add value to the process of understanding programs for non-programmers.

8.1.4 Blueprints for Haskell

With the cautiously optimistic reception of blueprints we received in the study described in Section 8.1.3, it may be worth while to make blueprints available to a larger audience. One way to do so would be to implement blueprint generation in GHC. Its compiler architecture supports plugins that act on its core language, minimizing the compiler modification we would need to implement. Additionally, GHC’s more powerful type system would enable more fine-grained control over Tonic’s value inspection feature, as discussed in Chapter 5, Section 5.7.

8.2 Potential Future Applications of Blueprints

This thesis is mostly focused on the technical aspects of generating and rendering blueprints. Accomplishing this took a great deal of work, which did not leave much time to experiment with the application of blueprints, with the exception of the experiment described in Section 8.1. In this section, we suggest future work on how blueprints could potentially be used to touch every aspect of the software development process. None of these aspects of blueprints have been studied formally, so we cannot conclude whether blueprints will be successful in these situations.

8.2.1 Software Requirements

Software development always starts with finding out what kind of software needs to be built. This is an important step in the process and is often done together with domain-experts. In iterative development settings, requirements gathering may even be done multiple times. Requirements engineers therefore require good tools to facilitate in communication with the experts. Blueprints may be able to aid in this process. Requirements engineers can sit down with domain experts and draw blueprints with pen and paper in order to capture the user requirements. In iterative development processes, domain experts can also look at the generated blueprints to understand what has been implemented and base any new requirements on those.
8.2.2 Software Verification and Validation

An important topic during software development is to make sure that the right software is being built. This is called software validation, which CMMI \[1\] defines as an effort “[..] to demonstrate that a product or product component fulfils its intended use when placed in its intended environment.” In other words, whether the software actually offers value to its end-users. One of the simplest ways to perform validation is to ask the intended end-users to use the product and provide feedback. While this form of validation is essential to any software development process, it is a time consuming and expensive process.

Another important aspect of software development is to make sure that the software is being built according to its specification. This is called software verification, which CMMI defines as an effort “[..] to ensure that selected work products meet their specified requirements.” In other words, does the resulting product conform to the requirements specification that was created before development started? Verifying software is a lot of work, because the original specification, often written in natural language, needs to be compared with a piece of software, which are two completely different things. A person will first need to read and understand the specification before a comparison can be made, which again is a time consuming and costly activity.

If verification and validation can be done more efficiently, it can save a lot of time in the development process. Speeding up software verification relies on making it easier to compare the specification to the implementation. If the specification is given in terms of blueprints, it becomes easy to compare it to the generated blueprint.

Speeding up software validation relies on more rapidly gathering feedback from the domain experts. Blueprints may allow domain experts to directly judge the implementation of the program, rather than being restricted to using the program as the only means of validation.

8.2.3 Documentation

Software documentation is important to an end-user who wants to use the program, as well as to programmers who need to understand the implementation, before they make changes to the code. Writing documentation is a labour-intensive process, because it is usually done in natural language and only semi-formal models and because documentation needs to keep up with the latest version of the software.

In practice, documentation commonly lags behind the actual implementation \[9\]. One way to mitigate this problem is to generate documentation from the program’s source code. This way, the documentation is always up to date with the implementation. Blueprints are exactly this: generated, high-level descriptions of the actual program implementation. They have the additional advantage that they document the implementation for programmers while remaining accessible to non-programmers.

8.2.4 Development

Integrating blueprints in an IDE may provide benefits to a programmer as well. A blueprint-enabled IDE can show the blueprint for the functions being edited, giving the programmer an alternative, graphical view on the same code. This graphical view may allow for a more intuitive notion of whether the code is structured in the right way. Such a view may even enable multi-disciplinary teams to work on the same code base, each with their own view on the program.
8.2.5 Debugging

In practice, software is rarely truly done and features need to be added, and programming mistakes need to be corrected. Particularly the latter chore is not always easy, evidenced by the large number of tracing and debugging tools that are available to programmers. Proper tool support for tracing and debugging can save a programmer a lot of valuable time and energy. By adding support for augmenting blueprints with *dynamic* information, they can be used as a tracer or debugging tool as well.

Tracers and debuggers are commonplace in programming. Particularly for imperative programming languages, where execution order is generally fixed, powerful debuggers have been developed. Debugging and tracing *lazy* functional language is harder to do, however, because execution order is *not* specified.

Tools for lazy functional languages do exist [74, 95, 10]. They tend to offer fine-grained debugging for any functional program, whereas blueprints focus on the abstraction level of monads. These tools and blueprints are therefore complementary. They could even be used in conjunction. Blueprints may provide a means to identify monadic computations that require debugging, after which one of the other debuggers can be used to perform fine-grained debugging of that code.

8.3 Graphics and Layout

One of the challenges associated with blueprints is drawing them. Since blueprints can be generated from any valid monadic program, it is in general unknown what they look like. The fully declarative graphics library `Graphics.Scalable` has greatly simplified drawing the blueprints. Thanks to its fully declarative API, drawing large and complex blueprints is not inherently harder than drawing small and simple blueprints. Its most apparent downside is related to the fact that the entire library is currently executed in the browser, which has a significant performance penalty. Solving these performance-related problems is the most important future work for this library.

When defining graphics declaratively, one also needs fully declarative layout combinators to position individual elements relative to one another. The layout language in the `Graphics.Scalable` library provides these as well. They also play an important role in making it easier to render arbitrary blueprints. We successfully generalized these combinators into a general-purpose layout language. It is now possible to write general layouts once and use them in different domains. What remains now is applying the layout language to more domains to gain more experience with it. Based on this experience, we can fine-tune the layout API, find patterns in identifying the components that need to be laid-out, and discover design patterns for general user interfaces.

8.4 Separation of Concerns

Blueprints should contain as much detail about the program as necessary to understand what a program does, but no more than that. In practice, this means that blueprints should mostly show control flow and data flow, but little else. Achieving this becomes easier when the target program is implemented with a strict separation of concerns in mind. For example, layout code must not clutter blueprints, which should mainly present control and data flow.
Achieving separation of concerns in iTasks can be achieved by applying Task-Oriented Software Development (TOSD) practices. TOSD separates traditional functional programming (TFP), like the definitions of algebraic data types and pure functions, from TOP-concerns such as tasks, shares, and layout. The separation of tasks, shares, and layout are valuable for rendering blueprints, so they do not get cluttered. The separation of TFP and the other concerns is valuable, because pure functions are easy to test and reason about, improving the reliability of the code. How the explicit awareness of TOSD can impact the development process remains a subject for future work. Still, one can already imagine human-machine-interaction experts working together jointly but orthogonally with task programmers to create the best possible software, both from a technical and usability standpoint.

8.5 Applications in C2

One of the goals of this thesis was to implement a prototype of a part of a C2 application. Since the C2 domain is far from trivial, a structured development approach was required. This resulted in the development of the TOSD approach.

The use of TOSD in the development of this C2 prototype is clearly visible in the program code of the C2 application. Significant parts of the application, such as shortest-path calculations and network flow calculations, are implemented using nothing but pure functions and algebraic data types. This resulted in code that was easily debugged and tested.

iTasks' ability to distribute tasks and automatically synchronize state between multiple clients plays a crucial role in the prototype application. In the application, multiple users need to work together to perform damage control on board of a ship. Task delegation and shared situational awareness are key to doing so successfully. TOP supports these requirements well. Task delegation is a built-in task, whereas shares provide a shared situational awareness for free.

One aspect of the application is that it contains a model of a ship in which various fire fighting and damage control (FFDC) scenarios can be performed. Currently, the ship models are created using a crude built-in editor. Future work should focus on taking the design of a real ship and use it as a realistic environment for the FFDC scenarios. In addition to the creation of a more realistic environment, another desirable next step would be to link the prototype to an OPV’s on-board systems so that we can read out real internal sensor data and utilize the on-board communication systems.

8.6 Wrapping Up

The new and improved tools presented in this thesis provide TOP developers with more means to implement software for complex domains. Additionally, the tools suggest new ways of developing functional programs in general. It is our hope that this work, combined with the work upon which it is built, will make TOP more accessible to a wider audience. We hope that particularly blueprints will eventually find applications in the wider field of functional programming as well.
My Publications

Published Articles


Technical Reports


Unpublished Articles

BIBLIOGRAPHY

Bibliography


[60] image.rkt, Dec 2014.


B. Yorgey. Monoids: Theme and variations (functional pearl). In *Proceedings of The Haskell Symposium*, Copenhagen, Denmark, September 13 2012. ACM.
Summary

Command & Control (C2) systems are complex socio-technical systems with which distributed groups of people and machines can be coordinated, such that they can work together towards a common goal. C2 systems are generally deployed to support complex cooperative tasks, and should be able to respond to dynamically changing situations and unexpected events.

The work that is supported by C2 systems is very complex, and the software that powers C2 systems is therefore necessarily complex as well. Because of this, developing such systems is far from trivial, and programmers require powerful tools to do so. This thesis is motivated by the challenge of further developing the set of tools needed to create C2 systems by developing a prototype C2 system for the Royal Netherlands Navy.

This work builds on top of an existing general-purpose application framework called iTasks, which is an implementation of a novel functional programming paradigm called Task-Oriented Programming (TOP). With TOP, programs are specified in terms of the tasks that a human or machine needs to do. Conceptually, the C2 domain is a natural match for Task-Oriented Programming, because the work that is supported by a C2 system can be seen as the tasks that humans and machines have to perform to achieve their goals.

Despite the fact that TOP leverages the common notion of tasks, communication with non-technical domain experts remains a challenge. There always exists a conceptual gap between the ideas that people have in mind and the formal implementation of these ideas in software, which hinders mutual understanding between programmers and domain experts. For successful communication, this gap needs to be bridged somehow, but there are no existing tools that do so successfully.

By building new tools that leverage the concept of tasks, we aim to facilitate successful communication between programmers and domain experts. The naval domain provides ample opportunity to identify where such tools are needed, because it is highly complex, enabling us to push the existing toolset to its limits.

We have developed a new tool that generates a graphical representation of an iTasks task specification that programmers have created. We call these graphical representations blueprints. Blueprints aim to leverage a person’s intuitive ability to understand pictures, combined with an intuitive idea of what a task is, to help them understand the programs that have been written by programmers.

Blueprints can be either static or dynamic. Static blueprints are a direct graphical representation of the original program code. Dynamic blueprints are instances of static blueprints that are graphically augmented with run time information. This information includes the tasks that are currently being worked on, who is working on which task, and the current value of the active tasks.

iTasks is implemented as a shallowly embedded domain-specific language (eDSL) in the lazy, purely functional programming language Clean. Blueprints can be generated for concepts other than tasks as well by leveraging the functional programming concept of monads. Monads enable sequential composition of computations in lazy functional programming languages. Static blueprints can be generated for any monadic program. In a lazy functional programming language, the order of computation cannot be guaranteed. Therefore, dynamic blueprints can only be generated for state monads, which enforce an
order of computation via the state they thread through internally.

Because iTasks is shallowly embedded in Clean, language constructs from the Clean language can be used to construct iTasks programs as well. This has a profound impact on our ability to generate blueprints. Due to the shallow embedding, host language constructs play an important role in an iTasks program. Some host language constructs therefore need to be included in the generated blueprints to make the blueprints understandable. However, these constructs are commonly only tangible at compile time. To deal with this, we need to involve the Clean compiler in the blueprint generation process. Particularly for dynamic blueprints, this means that we need to connect the static world at compile time with the dynamic world at run time. We make this connection by inserting trace functions at key points in the original program at compile time.

In generating a blueprint, one needs to make careful considerations on what to show and what not to show in it. Blueprints should contain as much detail about the program as necessary to understand what a program does, but no more than that. Too little detail and the blueprints are uninformative; too much and they become overwhelming. iTasks lends itself well to visualisation. Its monadic level of abstraction and strong separation of concerns allows for balanced blueprints to be generated.

Blueprints are not aimed at being a visual programming language, but at being a communication tool. While this thesis only focusses on the technical aspects of blueprints, there may be many areas in which they may be useful. Such areas may include software requirement gathering and specification, verification and validation, documentation, debugging, and education.

To generate blueprints and to bring all technologies together in a prototype C2 application, we have overcome several challenges along the way. Firstly, we developed a DSL in Clean to create Scalable Vector Graphics (SVG) in the browser, allowing us to draw and interact with the blueprints. Secondly, in order to make complex software like a C2 system easily adaptable to new insights and requirements, we created a systematic way to architect TOP applications, keeping a separation of various program concerns, like data modelling and graphical layout, in mind.

The work in this thesis truly is a cocktail of tools. What connects these tools is that they are all related to the concept of eDSLs. It is our hope that this work, combined with the work upon which it is built, will make TOP more accessible to a wider audience. We hope that particularly blueprints will eventually find applications in the wider field of functional programming as well, such as in Haskell.
Samenvatting

Command & Control (C2) systemen zijn complexe sociotechnologische systemen waarmee gedistribueerde groepen mensen en machines aangestuurd kunnen worden om een gezamenlijk doel te bereiken. C2 systemen worden gebruikt om complexe coöperatieve taken te ondersteunen en dienen dynamisch te kunnen reageren op veranderende situaties en onverwachte gebeurtenissen.

Het werk dat door C2 systemen ondersteund wordt is erg complex, met als gevolg dat de software voor C2 systemen ook noodzakelijk complex is. Hieruit volgt dat de ontwikkeling van dergelijke systemen ook complex is en programmeurs krachtig gereedschap nodig hebben om de systemen te implementeren. Dit proefschrift is gemotiveerd door de uitdaging om de verzameling gereedschap dat nodig is om C2 systemen verder te ontwikkelen door een prototype van een C2 systeem te ontwikkelen voor de Koninklijke Nederlandse Marine.

Dit werk bouwt voort op een bestaand algemeen toepasbaar applicatie raamwerk genaamd iTasks; een implementatie van een nieuw functioneel programmeren paradigma genaamd Task-Oriented Programming (TOP): taak-georiënteerd programmeren. Met TOP kunnen systemen gespecificeerd worden in termen van de taken die door een mens of een machine uitgevoerd dienen te worden. Conceptueel zijn het C2 domein en Task-Oriented Programming een geschikte combinatie, omdat het werk dat ondersteund wordt door een C2 systeem gezien kan worden als de taken die een mens of machine uit moet voeren om hun doel te bereiken.

Ondanks het feit dat TOP gebruik maakt van het concept van taken blijft communicatie tussen domeindeskundigen en programmeurs een uitdaging. Er bestaat altijd een conceptueel gat tussen de ideeën die mensen in hun hoofd hebben en de implementatie daarvan in software. Dit gat belemmert de communicatie tussen programmeurs en domeindeskundigen en moet gedicht zien te worden. Er bestaan echter geen geschikte tools om dat te doen.

Door nieuwe tools te ontwikkelen die het taak concept gebruiken trachten we de communicatie tussen programmeurs en domeindeskundigen te verbeteren. Het marine domein biedt veel kansen om dit doel te bereiken, omdat het een zeer complex domein is. Dit stelt ons in staat de bestaande tools tot hun uiterste te drijven.

Voor dit werk hebben we een nieuwe tool ontwikkeld dat een grafische representatie van een iTasks taakbeschrijving, zoals gespecificeerd door programmeurs, genereert. We noemen deze grafische representaties blueprints. Blueprints trachten het menselijke intuitionistieke begrip voor plaatjes en het taakconcept te gebruiken om de programma’s die programmeurs schrijven inzichtelijk te maken.

Blueprints kunnen zowel statisch als dynamisch zijn. Statische blueprints zijn een directe grafische weergave van de oorspronkelijke programmacode. Dynamische blueprints zijn instanties van statische blueprints die tevens informatie bevatten die tijdens de uitvoer van het programma beschikbaar is gekomen. Deze informatie bevat onder andere de taken waaraan op dat moment gewerkt wordt, wie aan welke taak werkt en de waarde van de actieve taken.

iTasks is geïmplementeerd als een ondiep ingebedde domein-specifieke taal in de hui, puur functionele programmeertaal Clean. Blueprints kunnen ook gegenereerd worden voor andere concepten dan taken alleen door gebruik te maken van het functioneel pro-
grammeren idee van monads. Monads maken sequentiële compositie van berekeningen mogelijk in een luie functionele programmeertaal. Statische blueprints kunnen gegenereerd worden voor elk monadisch programma. In een luie functionele programmeertaal kan de volgorde van berekeningen echter niet gegarandeerd worden. Dit heeft als gevolg dat dynamische blueprints enkel gegeneerd kunnen worden voor state monads, welke een evaluatievolgorde afdwingen doordat ze intern een staat doorgeven.

Doordat iTasks ondiep ingebd is kunnen taalconstructen uit Clean ook gebruikt worden om iTasks programma’s te specificeren. Dit heeft een sterke invloed op de manier waarop we blueprints genereren. Door de ondiepe inbedding spelen elementen uit Clean een belangrijke rol in iTasks programma’s. Om blueprints begrijpelijk te maken zullen sommige van deze elementen ook bevat moeten worden in de blueprints. Deze elementen zijn echter niet beschikbaar tijdens de uitvoer van een programma. Om deze elementen toch te kunnen bevatten in de blueprints moeten we de Clean compiler betrekken in het blueprint generatie proces. Voor dynamische blueprints betekent dit dat we de statische wereld tijdens het compilatieproces moeten zien te verbinden met de dynamische wereld tijdens de uitvoer van het programma. We maken deze verbinding door middel van het toevoegen van traceerfuncties in het originele programma tijdens de compilatie.

Tijdens het genereren van een blueprint moet men een goede overweging maken van de informatie die al of niet in de blueprint bevat wordt. Blueprints moeten genoeg detail bevatten om inzicht te geven in wat een programma doet, maar niet meer dan dat. Te weinig detail en de blueprints zijn niet informatief. Te veel en ze zijn niet langer begrijpbaar. iTasks leent zichzelf wel voor visualisatie. Het monadische抽象nieveel en sterke scheiding van verschillende belangen maken het mogelijk om gebalanceerde blueprints te genereren.

Blueprints zijn geen grafische programmeertaal. Ze zijn enkel een communicatiemiddel. Alhoewel dit proefschrift zich voornamelijk richt op de technische aspecten van blueprints, zijn er meerdere mogelijke toepassingsgebieden voor blueprints. Dit kan zijn in het vergaren en specificeren van software requirements, verificatie en validatie, documentatie, debugging en educatie.

Om blueprints te kunnen genereren en alle technieken samen te brengen in een C2 prototype hebben we verschillende hindernissen moeten overwinnen. We hebben een domein-specifieke taal ontwikkeld om Scalable Vector Graphics (SVG) te kunnen weergeven in de browser, zodat we blueprints kunnen tekenen en met ze kunnen interacteren. Ook hebben we nieuwe manieren ontwikkeld om op een systematische wijze en met een strikte scheiding van belangen TOP programma’s te ontwerpen.

Dit werk is een ware cocktail van tools. Wat deze tools gemeen hebben is dat ze allen gerelateerd zijn aan het concept van ingebedde domein-specifieke talen. Het is onze hoop dat dit werk TOP toegankelijk maakt voor een groter publiek. We hopen tevens dat blueprints in het specifiek ook toepassingen vinden in andere gebieden van het functioneel programmeren veld, bijvoorbeeld in Haskell.
Acknowledgements

I have had the privilege to enjoy a long and productive student life, mainly because my parents kept supporting me unconditionally. Without them, I would not have gotten to where I am today, and I would have never had the opportunity to write this thesis. Thank you mom and dad!

All the work in this thesis is related to functional programming. I was first introduced to functional programming at Utrecht University, where it changed both my personal and professional life in profound ways. My thanks go out in particular to Doaitse, Andres, Atze, Wouter and Johan, and in general to the entire Software Technology group at Utrecht University.

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. Rinus, Peter, and Jan Martin: thank you for your guidance and help over the years as my supervisors. My thanks also go out to László, Bas, John, Tim, Markus, Steffen, Pieter, Ingrid, Giso, Manxia, Maarten, Martijn, and everyone else in iCIS for creating a great environment to work in and by being there for me when I had questions to ask or a bottle of wine to share.

This research has been made possible by the Netherlands Defence Academy and TNO. Aside from these organisations in general, I would like to thank Fok, Jos and Guido in particular for their support, feedback, and ideas.

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, Doaitse, and Sven-Bodo.

There are countless other people that have helped and supported me, or that I have enjoyed working, travelling, and socialising with. Thank you Manon, Jorrit, Nico, and anyone I may have unintentionally forgotten to name here.
Jurriën Stutterheim

1985     Born in Middelburg, the Netherlands
1998 – 2003  HAVO
            Meridiaan College, Vestiging Het Nieuwe Eemland
2005 – 2007  Programmer
            University Medical Center Utrecht and
            Utrecht University of Applied Sciences
2006 – 2010  Bachelor Computer Science
            Windesheim University of Applied Sciences
2007 – 2009  Entrepreneur
            Intermentis
2010 – 2013  Master Computing Science
            Utrecht University
2013 – 2017  PhD Candidate
            Software Science
            Institute for Computing and Information Science
            Radboud University Nijmegen