Radboud University

Institute for Computing and Information Sciences
Software Science (SwS)

Task-Oriented Programming for developing non-distributed interruptible embedded systems

Master's Thesis

Jasper Piers

Supervisors:

Dr. Pieter Koopman
Ing. Niek Maarse

Second reader:

Dr. Peter Achten

Final version

Nijmegen, August 2016

Abstract

Task-Oriented Programming (TOP) has proven itself effective for the implementation of interactive, distributed, multi-user applications through the use of the *iTasks* framework. In this thesis we show that TOP is also well-suited for developing non-distributed embedded systems whose processes can be interrupted by events that can occur at any time. A general-purpose TOP framework titled μ*Tasks* is created for use in a case study. Through this case study we show that a task-oriented solution results in code with a higher maintainability that is able to more effectively deal with the aforementioned events compared to a modern-day object-oriented one.

Contents

Contents iii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Problem statement . . . 2

1.3 Research method . . . 2

1.4 Thesis outline . . . 3

2 Task-Oriented Programming 4

2.1 History . . . 4

2.2 Principles . . . 5

2.3 Functional languages as a host language . . . 6

2.3.1 Purely functional languages . . . 7

2.3.2 Static type systems . . . 7

2.3.3 Lazy evaluation . . . 7

2.3.4 Function currying and higher-order functions . . . 8

2.3.5 Algebraic data types and generics . . . 8

2.3.6 The state monad . . . 8

2.4 iTasks . . . 11

2.4.1 Tasks and task evaluation . . . 11

2.4.2 Shared data sources . . . 12

2.4.3 Editors . . . 12

2.4.4 The step combinator . . . 14

2.4.5 The parallel combinator . . . 14

3 µTasks: a general-purpose Task-Oriented Programming framework 16

3.1 Task-related types and definitions . . . 17

3.2 Basic task functions 17

3.2.1 List-based tasks 19

3.3 The step combinator 20

3.4 Step combinator instances 22

3.4.1 Sequential combinators 22

3.4.2 Repetition combinators 23

3.4.3 Miscellaneous combinators 24

3.5 Exception handling 25

3.6 Parallelization 27

3.6.1 Timeout 29

3.6.2 Order maintenance 30

3.7 Shared data sources 32

3.8 Atomic tasks 36

3.9 I/O-related actions 40

3.10 Miscellaneous components 41

3.10.1 Tasks as a sequence of steps 41

3.10.2 Task equivalents of existing functions 42

3.10.3 Collections of tasks 43

4 Case study: a task-oriented payment terminal implementation 45

4.1 Background: payment terminals and transactions 45

4.1.1 Transaction simplifications 51

4.2 System characteristics and candidate solutions 51

4.2.1 Candidate solution: implicit data relay by use of state 52

4.2.2 Candidate solution: state as a shared resource 52

4.3 Proof of concept 55

4.3.1 Example: cardholder verification 56

4.3.2 Discussion 59

5 Related Work 61

5.1 Payment terminal implementation at CCV 61

5.1.1 Architectural landscape 61

5.1.2 Top-level payment application design 62

5.1.3 Problem areas 63

6 Conclusion **70**

6.1 Future work 71

Bibliography **72**

Chapter 1

Introduction

1.1 Motivation

Embedded systems are everywhere nowadays, being present in devices like cell phones, satellites, traffic lights and even pacemakers; day-to-day life would be very different without them. Embedded systems facilitate people's lives. As such, it is not surprising that the embedded system market has experienced accelerated growth in the past decades [EJ09] and that market analysts expect this market to continue growing [Hex14]. The growth of the amount of embedded systems came with an increase of embedded system software complexity [EJ09]. This increase in complexity poses a software design challenge, especially for systems that need to be robust [HS06]. Examples of this are things like payment terminals, where integrity of the account balance needs to be maintained, and systems used in avionics, where system failure could result in casualties.

Traditionally, programming languages like C or even assembly languages were used for implementing embedded systems due to the limited capabilities of these systems (e.g. a low amount of memory) [Bar98]. Embedded systems nowadays are much less restricted by hardware, opening up more possibilities implementation-wise. Some embedded systems are even equipped with an embedded operating system [SGG08]. One of the techniques available for implementing modern-day embedded systems is Object-Oriented Programming (OOP), which is one of the most prominent programming paradigms today [Sta16]. Object-oriented programming promises higher modularity and less complexity than procedural languages by providing the ability to refine software into objects. This is not a language guarantee however.

Task-Oriented Programming (TOP) is a relatively new programming paradigm [PAK07]. When developing an application using a Task-Oriented Programming language, a programmer is specifying *what* must be done, and not *how*. The used Task-Oriented Programming framework translates this specification into a fully functional application, automatically generating the *how*-aspects of the application (e.g. its user interface). This results in code that reads closely like the specification it is based on. In a task-oriented application, work is represented by *tasks*. The internals of tasks are hidden by design: an observer of a task is only able to observe its *task value* [AKLP13]. This gives tasks a high level of modularity. With *task combinators* tasks can be composed into new tasks. These task combinators represent common programming patterns, ranging from simple ones like performing two tasks in sequence to more complex ones like performing multiple tasks in parallel. Because of its facilities for parallelization, Task-Oriented Programming has proven to be effective for the implementation of distributed multi-user systems [Lij13b].

A subset of embedded systems are those that are *non-distributed* and that can be *interrupted*. A system is *distributed* when its hardware or software components are located at networked comput-

ers communicating only by passing messages [CDKB11]. With an *interruptable* system, we mean a system whose execution is allowed to be interrupted by unexpected events, without resulting in state corruption of the system or systems connected to it. An example of such a system is a payment terminal, where an unexpected event like removal of a payment card should maintain integrity of the card holder's account balance.

The properties of Task-Oriented Programming suggest that it may be well-suited for the implementation of this category of embedded systems. The parallel facilities of TOP seem well-suited for the detection of events that are not expected at any particular time. Additionally, the task abstraction mechanism of TOP and the way it coordinates tasks through combinators is expected to result in maintainable code that corresponds closely to the specification it is based on. As mentioned earlier, Task-Oriented Programming is well-suited for distributed applications. The only reason for not taking these applications into consideration is for scope limiting reasons.

1.2 Problem statement

The hypothesis of this thesis is that using Task-Oriented Programming for implementing *interruptible, non-distributed* embedded systems results in code with a higher maintainability and in an application that is able to more effectively deal with unexpected events, compared to modern-day object-oriented approaches.

1.3 Research method

A case study serves as the main method of study. In preparation of this case study, a general-purpose Task-Oriented Programming framework named *µTasks* was developed. This is done because the only existing TOP framework, *iTasks*, aims to facilitate the development of a different category of applications, namely "interactive, distributed, multi-user applications" [AKP13]. As such, the semantics and components of *iTasks* are closely tied to this problem domain. While the systems considered in this thesis are interactive, this interactivity involves only other systems; user interactivity is not one of the properties we wish to facilitate. Additionally, the applications under consideration are not distributed; they run on a single system.

Despite it targeting a different group of applications, *µTasks* adheres to the task-oriented principles and resembles *iTasks* in aspects like naming for similar task combinators. For the implementation of *µTasks*, the purely functional programming language *Haskell* is used. This means all code snippets provided in this thesis are written in the *Haskell* programming language, unless explicitly stated otherwise. There was no particular reason for choosing this language. Any other pure functional language, for example *Clean* [Pla01] which is used for implementing *iTasks*, could have been used.

The case study is performed at a company named CCV[1]. CCV offers a comprehensive range of payment solutions for both small and large businesses in various markets. These solutions include everything needed to facilitate and validate payments, including the processing of payments and other transactions. "CCV systems" is the software development department of CCV, which consists of several teams tasked with developing and maintaining software for the aforementioned payment solutions. One of the main activities of these teams is the development of software for various payment terminals. One line of terminals, named "CCV#", is implemented using the Object-Oriented Programming language C# [AA15].

[1] https://www.ccv.nl/

The goal of the case study is to develop a task-oriented implementation for one of the processes performed on a payment terminal: performing a financial transaction. The first step performed in this case study is analyzing what a payment terminal transaction entails. Based on this analysis, several core characteristics are identified. Because a financial transaction is very complex and involves many different variables, these characteristics have been identified in the context of *interruptability*. All other details are either simplified and omitted. As such, the resulting task-oriented solution is a proof of concept, not a fully functional application. From the identified characteristics, task-oriented candidate solutions are defined. One of these solutions is determined best-suited and used for development of the proof of concept using μ*Tasks*. The findings of the case study are compared against the results of an analysis performed on the aforementioned "CCV`#`" terminals.

1.4 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 introduces Task-Oriented Programming. In this introduction, its history and principles are discussed, as well as the *iTasks* framework and functional programming on which it relies. Chapter 3 presents μ*Tasks*, a general-purpose Task-Oriented Programming framework. Next, chapter 4 presents the results of the case study. Chapter 5, contains an analysis of the software running on CCV`#` payment terminals. Finally, chapter 6 concludes this thesis.

Chapter 2

Task-Oriented Programming

This chapter introduces Task-Oriented Programming. When developing an application using a Task-Oriented Programming language, a programmer is specifying *what* must be done, and not *how*. The used Task-Oriented Programming framework translates this specification into a fully functional application, automatically generating the *how*-aspects of the application (e.g. its user interface). In a task-oriented application, work is represented by *tasks*. Tasks have a high level of modularity because observers of a task only observe a value emitted through a typed interface. With *task combinators*, tasks can be composed into new tasks. These task combinators represent common programming patterns, ranging from simple ones like performing two tasks in sequence to more complex ones like performing multiple tasks in parallel. Task-Oriented Programming has proven to be an effective method for developing "interactive, distributed, multi-user applications" [AKP13, Lij13b]

Section 2.1 discusses the historical background of Task-Oriented Programming. Next, section 2.2 enumerates the underlying principles and design philosophy. Section 2.3 briefly discusses some properties and concepts of functional programming, which lie at the heart of Task-Oriented Programming. *iTasks*, from which the Task-Oriented Programming paradigm originated, is discussed last in section 2.4.

2.1 History

Task-Oriented Programming, abbreviated as TOP, is a relatively new programming paradigm. It is the next step up from a toolkit named *iData* [PAK07], which is short for *i*nteractive *Data*. This toolkit enabled programmers to program forms in server-side web applications. At the time, forms were typically viewed at a low level: a collection of primitive elements like input fields, check boxes, buttons, etc. The goal of the *iData* toolkit was to take away the necessity of viewing forms on a low level by allowing programmers to view them only on a high level. On a high level, forms can be seen as editors of structured values having a certain type [PA05]. The *iData* toolkit took care of generating the low level realization from the high level type definitions. This meant that programmers were now tasked with creating data types when creating forms instead of having to perform a lot of low level coding (e.g. HTML). Because forms are interactive, the *iData* toolkit also had to be capable of mapping user actions performed on forms back to the type domain. The challenges and corresponding solutions of this bidirectional mapping problem and other implementation related topics are discussed in [PA05]. Any language could have been used to develop the toolkit as long as it had good support for data types and type-driven programming. Functional programming languages made good candidates because of their strong type systems (section 2.3.2). The *Clean* functional programming language [Pla01] was used to develop the *iData* framework. One way

Clean supports type-driven programming is through *generic programming* [Hin00] (section 2.3.5). This technique is what allows the *iData* framework to generate forms automatically from the type of the value of an editor. More information on the programming method used for *iData* can be found in [PA06].

The *iData* framework is capable of creating complex forms that can be used to create and change values of complex data types, but it does not support the specification of complex control flows (e.g. the payment process in a web store). This is because every request sent to a server running an *iData* web application results in the application being started from scratch; the application is stateless. The *iTasks* framework, first introduced in [PAK07], is the result of including control flow in the *iData* toolkit. The addition of control flow was inspired by contemporary work flow systems in which work flow situations were specified that had to be executed by humans and computers. *iTasks* is used for the specification of "interactive multi-user web-based work flows" [PAK07]. Because *iTasks* is built on top of the *iData* toolkit, *iTasks* specifications can be defined with a high level of abstraction thanks to type driven programming and other features provided by the functional host language *Clean*: strong typing (section 2.3.2), higher-order functions (section 2.3.4), lazy evaluation (section 2.3.3), monadic programming (section 2.3.6), etc. *iTasks* is defined as a combinator library meaning *iTasks* applications are written in an *iTasks* language (technically: a Domain-Specific Embedded Language, or DSEL [Hud96]). The Task-Oriented Programming paradigm was born from the principles and philosophy behind *iTasks*.

2.2 Principles

Task-Oriented Programming aims to facilitate the development of "interactive, distributed, multi-user applications" [AKP13], as mentioned in section 2.1. Here, "interactive" refers to the interaction between an application and its user(s). A user typically interacts with an application through a Graphical User Interface (GUI). A system is "distributed" when its hardware or software components are located at networked computers communicating only by passing messages [CDKB11]. A distributed system typically involves many different devices (e.g. desktop computers, smartphones, embedded systems, etc.). Finally, an application is "multi-user" when used by multiple users simultaneously. These users typically have a common goal, meaning the application is used as a tool to collaborate. A great example of this is collaborative editing, used in tools like "Google Docs", where the resulting document is composed of individual contributions. The internet is obviously well-suited for Task-Oriented Programming applications, because its architecture closely relates to the aforementioned properties.

Similar to how programmers can view forms on a higher level of abstraction by thinking only about the involved data types when using *iData*, programmers using *iTasks* are tasked with specifying *what* needs to be done rather than *how*. This is one of the core principles of Task-Oriented Programming. This *how* aspect results from the addition of control flow to *iData*, as discussed in section 2.1. Programmers using *iTasks* are essentially specifying *what* needs to be done and what the involved data types are.

The *how* part encompasses different aspects of an application. User interactivity for example involves the challenge of creating and maintaining a user interface. Extensive knowledge on GUI toolkits is required to tackle such a challenge. When there is no strict separation between the *what* and *how* parts, the structure of the specifications will influence each other. The GUI toolkit (dealing with a *how* part of the application) mentioned earlier for example, will typically enforce a programmer to structure his or her application in a certain way [Mye03]. This structure may not always be intuitive for the application logic (i.e. the *what* part of the application). Another example on which this principle applies, is network communication. A programmer that creates a Task-Oriented application does not (want to) have to concern himself with the intricacies of network communication [Ste93].

When a programmer develops a task-oriented application, he specifies *what* needs to be done by means of “tasks”. A task is a fundamental building block of Task-Oriented Programming, representing *work* that needs to be done. Tasks have a typed interface accessible by their observers. The type of a task reflects the result (or more specifically: the type) of the work it performs. This means an observer of a task has no notion of what the task is doing internally, which is in line with the principle of focusing on the *what* rather than the *how*. This results in a high level of modularity: tasks having the same interface can be used interchangeably.

An observer can obtain the current state of a task at any time, through the *task value* that all tasks emit while they are being performed. Task values are the only way for observers of a task to get to know anything about the task. Based on its observations, on observer can decide on how to proceed. For example, it can decide to continue performing the task, stop it or start a new task. When a task is finished, a *stable* task value will be observed. Up until that point, one might either observe an *unstable* task result, representing an intermediate result, or no task value at all. While it is possible to observe an unstable task value after observing no task value and vice versa, it is not possible to observe either of them after observing a stable task value. This relation between task values is depicted by figure 2.1.

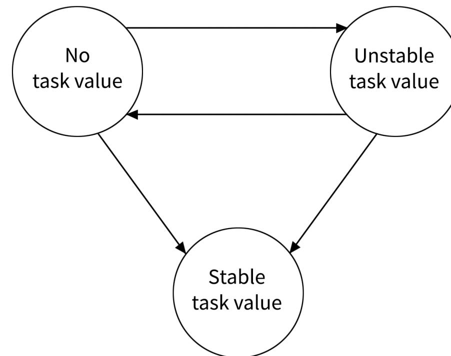

Figure 2.1: Possible transitions of task values [AKLP13].

A task-oriented application is typically developed through a process of *refinement*. When work is sufficiently simple, it can be expressed by a basic task, supplied by the used Task-Oriented Programming framework. Work that is more complicated is refined into smaller portions of work. Hence, an important aspect of task-oriented programming is identifying sub-tasks and how they cooperate. A Task-Oriented Programming framework typically contains common programming patterns for use in this refinement process.

Task-Oriented Programming was born from the *iTasks* framework as discussed in section 2.1. This framework uses a pure functional language as a host language. In section 2.3 we will first discuss some properties of such languages before proceeding with a brief overview of *iTasks* in section 2.4.

2.3 Functional languages as a host language

As mentioned in section 2.1, *iTasks* was developed using the pure functional language *Clean* [Pla01] as a host language. This means *iTasks* utilizes *Clean*’s language aspects. This section will briefly discuss some of aspects of pure functional languages like *Clean*. This section does not aim to be a complete and formal functional programming specification. The goal of this section is to provide a basic understanding that will be beneficial when reading the remaining sections of this thesis.

2.3.1 Purely functional languages

Functional programming languages can be subdivided into pure and impure languages. A formal definition of purity is proposed in [Sab98]. Intuitively, a functional language is considered pure when functions are a pure mapping from arguments to results. In other words: functions do not have any side-effects. Arguably one of the biggest advantages of this property is that writing applications for parallel computers is easier in these languages than in imperative languages [PJ89, HM00]. Additionally, this property opens up new ways of thinking about data structures [Oka98]. For a programmer, this property helps with reasoning as functions are isolated in the sense that they only depends on their parameters. By consequence, it does not matter in what order the arguments of functions are evaluated, regardless of the used evaluation strategy (section 2.3.3). This property is also called “referential transparency”.

2.3.2 Static type systems

Static type systems play an important role in typed functional programming languages. The primary role of a static type system is to prevent a class of errors from happening at run-time [Car96]. The type system determines whether or not a program is *well behaved*. A well behaved program can still result in execution errors when run (e.g. when dividing by zero), but a large subset of possible execution errors cannot occur. In particular, no “untrapped errors” can occur. These are errors that go unnoticed for a while and later cause arbitrary behavior. Most functional programs are *statically checked*, meaning static checks (i.e. compile time) are peformed to ensure the program is well typed. This prevents errors that occur when doing things like adding two strings together when addition is only defined for integral types, or applying an integer to a function.

A static type system also has some efficiency benefits because objects do not need to carry their types around at run-time, because their types have already been established at compile-time. The removal of type information at compile time is known as “type erasure” [MLS08].

While a language does not need to be typed in order to be functional, a host language of a Task-Oriented Programming framework requires a typing system because of how important types are in Task-Oriented Programming (section 2.2). A strong typing system allows a host language to derive implementation details from type definitions. *iTasks* for example uses generic programming (section 2.3.5) to automatically generate user interfaces from type definitions [AKLP13]. This makes *iTasks* applications appear as a high-level specifications, while they are complete implementations.

A type system may also be capable of inferring types. This means that when no type has been explicitly mentioned, the type system will assign it the most general type. These most general types may be polymorphic, meaning their operations are applicable to values of more than one type [CW85].

2.3.3 Lazy evaluation

The way in which expressions are evaluated is determined by an evaluation strategy. Functional programming languages use either eager (strict) or lazy evaluation (non-strict) [CLW+15, Ses02]. Informally, eager evaluation results in expressions being evaluated as soon as possible, while lazy evaluation defers this until their results are actually needed. Recall that in purely functional programming languages the outcome of a calculation is not effected by the order in which subcomputations are performed (section 2.3.1). This is what allows the deferment of computations. [Gol96] identifies two advantages of lazy evaluation. The first advantage is that a programmer does not need to concern himself with structuring things in a manner that results in the desired order of evaluation, because expressions will be evaluated when needed as mentioned earlier. The second advantage is that lazy evaluations allows the construction of infinite data structures. This would

not be possible in a strict language, because using such a structure would result in evaluating it in its entirety, which obviously never terminates. Non-strict functional programming languages typically have language constructs to force strict evaluation. *Clean* for example uses strictness annotations to achieve eager evaluation [Pla01]. A downside of lazy evaluation is that generally it is less efficient than eager evaluation.

2.3.4 Function currying and higher-order functions

Currying is the idea that functions having multiple arguments are the result of iterative function application. [BB00] gives an example in λ-calculus where $f(x, y)$ intuitively depends on x and y. One can then define a function where x is a free variable, meaning only y is bound: $F_x = \lambda y.f(x, y)$. A function that binds x can then be defined as $F = \lambda x.F_x$ so that $(Fx)y = F_x y = f(x, y)$. In a functional programming language this is typically used in the form of partial function application. The result of partial function application is the function that accepts the remainder of the arguments. This property is especially powerful in combination with "higher-order functions". These are functions that take one or more functions as arguments and/or return a function as their result [Sta94]. Higher-order functions carry out the idea that functions should be treated the same as values of basic types like integers and booleans.

2.3.5 Algebraic data types and generics

Strictly speaking, there are two common classes of data types, namely product types and sum types [HHPJW07]. Tuples and records are examples of product types. Algebraic data types are sum types. Sum types are defined by specifying multiple alternatives called "products", or in other words: a "sum of products". Every product has zero or more "fields". A typical example of an entity that would be defined as an algebraic data type in a functional programming language is a binary tree. A binary tree consists of two products, one representing a node and the other representing a leaf. The node product would contain two fields, one for each subtree. A leaf contains just one field, representing its value. In programming languages, products are generally referred to as "data constructors". Functional programming languages like *Clean* and *Haskell* support pattern matching against algebraic data types using their data constructors. Pattern matching greatly increases the readability of code and simplifies the process of reasoning formally about functions [BW88].

Generic programming allows a programmer to define functions that apply to all data types. Typical examples of this are comparison functions, pretty printers and parsers. In his paper [Hin00], Ralf Hinze presented a universal representation for all data types. This makes generic programming possible because functions defined for this universal representation can be specialized to arbitrary types. Generic programming fits well in the philosophy of task-oriented programming (section 2.2) because a programmer gets functions for his or her types "for free". He need not concern himself with specifying an instance for the newly introduced type, which would be more of a *how* aspect rather than a *what*. *iTasks* for example makes extensive use of generic programming [AP02, PLM+12]; it is capable of generating complete user interfaces from task- and datatype definitions (section 2.4).

2.3.6 The state monad

While pure functional languages facilitate some aspects of programming (section 2.3.1), they complicate others. One of the things that is problematic in a functional language is dealing with stateful algorithms, because side-effects are prohibited. A structure named "monad" from category theory is a solution for dealing with stateful algorithms and side effects in a pure functional language.

Eugenio Moggi first described the general use of monads to structure programs in 1989 [Mog89] and Philip Wadler built upon this, resulting in a practical solution to the aforementioned problem in 1990 [Wad90].

This section will discuss a specific type of monad named the “state monad”. This monad is used for dealing with stateful algorithms in a pure manner. The reason for discussing this specific monad in this section is that it is a cornerstone of the task-oriented framework that will be presented in chapter 3. The discussion in this section is limited to an informal description by example. Code fragments of this example are given in the pure functional programming language *Haskell*. For a formal description of the state monad, and monads in general, consult [Wad90].

One of the simplest examples of a stateful algorithm is an algorithm for generating fresh integers. An initial attempt to develop a function that yields a fresh integer may start off with defining the following seemingly intuitive function signature:

```
freshInt :: Int
```

However, when trying to define the implementation one will quickly realize that this will not work. After all, side effects are not allowed meaning `freshInt` must always yield the same result when parametrized in the same way. With this definition there is only one way to parameterize the function, namely with zero arguments. Hence, an obvious solution would be to add a parameter to the function that influences the outcome. A counter would a logical candidate for this, as it is essentially the state of the algorithm representing the next fresh integer. In order to make subsequent calls possible, an updated version of the counter also has to be part of the result. This results in the following revised definition:

```
type Counter = Int

freshInt :: Counter -> (Int, Counter)
freshInt counter = (counter, counter + 1)
```

With this definition, `freshInt` can be used to obtain a fresh integer. A function that produces a triple containing 3 fresh integers is defined next. Here `x`, `y` and `z` are fresh integers produced by `freshInt`.

```
example :: Counter -> ((Int,Int,Int), Counter)
example n = let (x, n'  ) = freshInt n
                (y, n'' ) = freshInt n'
                (z, n''') = freshInt n''
             in ((x,y,z), n''')
```

This exposes an obvious downside: the counter pollutes the code because `freshInt` requires the latest version in order to ensure it generates a value that was not yet generated. This is where the state monad comes into play. This example will be transformed into a solution utilizing a state monad which removes the downside of having to explicitly pass around the state (i.e. the counter). First we define the monadic machinery for our state monad:

1. The type for which monadic operations will be defined. `freshInt` is a stateful algorithm for which the counter acts as a state. This means the aforementioned type will be a type

representing functions that operate on this state. For illustration purposes this type will be named `ExampleMonad`. Note that `ExampleMonad` is a type constructor; it is parametrized with an arbitrary type `a`. For simplicity of the example, the state is not one of the parameters (i.e. `ExampleMonad s a`).

```
type ExampleMonad a = Counter -> (a, Counter)
```

2. A unary function named `return` (also called `unit`). This function is parametrized with a value from a plain type (i.e. `a`) and creates a monadic value from it having the monadic type (i.e. `ExampleMonad a`):

```
return :: a -> ExampleMonad a
return x = \s -> (x,s)
```

A monadic value representing the integer 42 is now created by writing `return 42`.

3. A binary function named `bind`. This is more often than not defined as an infix operator named `(>>=)`. This function takes a monadic value (typed `ExampleMonad a`) as its first argument and a function (typed `a -> ExampleMonad b`) that can transform this value as its second argument. Note that this function is parametrized with the plain value (typed `a`); the bind function unwraps the plain value embedded in its first argument.

```
(>>=) :: ExampleMonad a -> (a -> ExampleMonad b) -> ExampleMonad b
(>>=) ma mb = \s -> uncurry mb (ma s)
    where
    uncurry :: (a -> b -> c) -> (a, b) -> c
    uncurry f p =  f (fst p) (snd p)
```

Sometimes the result of the first monadic operation is irrelevant. A bind version (typically named `(>>)`) can be defined which does just that. It is a less general version of `(>>=)`:

```
(>>) :: ExampleMonad a -> ExampleMonad b -> ExampleMonad b
(>>) ma mb = ma >>= const mb
```

With the bind function it is now possible to compose a sequence of monadic function calls.

4. Functions for getting and setting the state. In literature these are named `fetch` and `assign` respectively. In this definition, `()` is the void type which has just one data constructor: `()`.

```
fetch :: ExampleMonad Counter
fetch = \s -> (s, s)

assign :: Counter -> ExampleMonad ()
assign s' = \s -> ((), s')
```

With the monadic machinery in place, we can now define a monadic version of `freshInt`:

```
freshInt :: ExampleMonad Int
freshInt = fetch          >>=
     \n -> assign (n + 1) >>
           return n
```

This new version of `freshInt` results in the following updated definition of `example`:

```
example :: ExampleMonad (Int, Int, Int)
example = freshInt >>=
    \x -> freshInt >>=
    \y -> freshInt >>=
    \z -> return (x, y, z)
```

Note that there is no mention of a counter anywhere in this definition. `freshInt` appears as a function that has no arguments and that simply produces an integer, similar to the initial (impossible) definition. The monadic operations implicitly pass on the state, removing the earlier problem of having to explicitly pass it around.

In this small toy example it may seem like a disproportional amount of effort is required to set up the monadic machinery just to hide the state. In general this is not true. In real-world applications, a state will typically be passed down multiple functions. Not all of these will use the state. It is in these cases that a state monad provides a major benefit because the state will not pollute the function definitions. State monads are the de facto way of elegantly dealing with stateful algorithms in functional programming languages.

2.4 iTasks

As stated in section 2.1, *iTasks* is not just a framework that adheres to the Task-Oriented Programming paradigm (section 2.2), it is the framework from which the Task-Oriented Programming paradigm originated. Because of this, *iTasks* and most of its components are not just considered an instance of Task-Oriented Programming, but are synonymous with it. In this section we will present some of the major concepts of *iTasks*. Some of these concepts are expressed by “combinators”, since *iTasks* is a combinator library. Combinators are named programming patterns that precisely express how smaller pieces of program can be combined into a bigger program [AKP13]. *iTasks* is written in the purely functional programming language *Clean* [Pla01]. All code snippets presented in this section are written in this language.

2.4.1 Tasks and task evaluation

Tasks in *iTasks* are state transforming functions that are driven by events [PLM+12]. These events originate from the user interfaces (i.e. web pages) that are automatically generated by the framework (section 2.4.3). There are three types of events in *iTasks*: (1) A refresh event, indicating that a user wants to refresh a web page, (2) an edit event, containing a new value for use with an interactive task and (3) an action event, representing the action selected by a user on the web page (e.g “ok”, “cancel” or a custom action). The last two events address the task that is to handle the event by an identification number assigned by the framework.

Tasks rewrite to new versions of themselves representing the remaining work to be done and accumulate responses to users along the way. Every time a task is rewritten, it yields a task result. A task result is either an exception or a task value (figure 2.1). In case of a task value, the task result also contains the timestamp of the event that caused the task value’s creation.

On a top-level, there is one main task. This task gets recursively rewritten until either it produces a stable value or an exception is observed. It is possible that neither occurs, resulting in a task that never finishes. Because tasks of *iTasks* are event-driven, every rewrite requires an event.

iTasks stores these events in a queue. When this queue is empty, rewriting of a task is halted until an event becomes available. If the rewriting of the task resulted in more work to be done, clients are sent the accumulated responses before recursively continuing with the remainder of the work.

2.4.2 Shared data sources

Shared data sources, or SDSs, are used by tasks to share information. They are Uniform Data Sources (UDS) which were proposed in [Mic12] as a solution to the problem of dealing with different data sources (e.g. databases and files) in a uniform way. In line with the philosophy of *iTasks* (section 2.2), uniform data sources separate what a data source is from how it is used. Shared data sources are abstract typed interfaces used for reading and writing atomically. The type read from a shared data source may differ from the type written. This makes a form of access control possible (e.g. read-only and write-only). Shared data source combinators for projection, used for changing the read or write type, and composition, used for combining shared data sources into new ones, are proposed in [Mic12]. The resulting type for a shared data source in *iTasks* is `RWShared r w`, where values of type `w` are written, and values of type `r` are read. From this type, less general types are derived, as shown in the *iTasks* definition:

```
:: RWShared r w

:: ROShared r :== RWShared r  ()
:: WOShared w :== RWShared () w
:: Shared   a :== RWShared a  a
```

iTasks makes a distinction between internal and external shared data sources [PLM+12]. The former are used for communication between tasks while the latter is used for communication with external objects. Internal SDSs typically have a limited scope, while external SDSs are accessible throughout the entire application. Internal data sources do not make use of access control, meaning they are of type `Shared a`. Examples of data sources provided by *iTasks* are data sources for obtaining a list of registered users and the current time:

```
currentTime  :: ROShared Time
currentUsers :: ROShared [User]
```

For internal shared data sources, shared memory has to be available. This memory is represented by a list contained in the state that tasks transform 2.4.1. Because the type stored in shared memory is arbitrary, this memory is represented by a heterogeneous list. Such a list is possible because of *Clean*'s `Dynamic` type, which can represent any type [VP03]. Every internal shared data source reserves a position of this list.

2.4.3 Editors

In *iTasks* there are user-interactive tasks that enable a user to modify a value of some type through a visual representation. These interactive tasks are called *editors* [PLM+12]. One of the main selling points of *iTasks* is that these visual representations can be automatically generated for any first order type (section 2.3.5). These visual representations are also known as *views*, and the value they represent are also known as *models*. Events originating from views are handled by the *iTasks* framework to update the corresponding models.

iTasks contains one core editor task [PLM+12], but provides several derived editors for better readability. These derived editors can be divided into three categories: (1) the *enter* variant, used to enter information (2) the *update* variant, used to update information and (3) the *view* variant, used for viewing information. There are variants for both shared (section 2.4.2) and non-shared information. Example of these derived editors are shown below.

```
enterInformation  :: d -> [EnterOpt  m  ] ->      Task m | descr d & iTask m
updateInformation :: d -> [UpdateOpt m m] -> m -> Task m | descr d & iTask m
viewInformation   :: d -> [ViewOpt   m  ] -> m -> Task m | descr d & iTask m

viewSharedInformation   :: d -> [ViewOpt   r  ] -> RWShared r w -> Task r | descr d & iTask r
updateSharedInformation :: d -> [UpdateOpt r w] -> RWShared r w -> Task w | descr d & iTask r
                                                                                    & iTask w
```

One thing to note is that these editors are provided a description. This description will be shown in the visual representation. Another thing to note is that editors are provided with a list of options. These options allow fine-tuning in case the default editor does not provide the desired view. An example of an editor is one that allows entry of a person. This requires a model of a person and nothing more, since *iTasks* generates the interface and handles the communication:

```
:: MyPerson =
    { name          :: String
    , gender        :: MyGender
    , dateOfBirth   :: Maybe Date
    }

:: MyGender = Male | Female

derive class iTask MyPerson, MyGender

enterPerson :: Task MyPerson
enterPerson = enterInformation "Enter your personal information" []
```

From this specification, *iTasks* is capable of generating the user interface shown in figure 2.2 automatically.

Enter your personal information

Name*:

Gender*: Select...

Date of birth:

Select...
Male
Female

Figure 2.2: The visual part of an *editor* task of *iTasks*.

A client sends *events* to the *iTasks* server (section 2.4.1) which responds with *responses*. There are two types of messages in *iTasks*: *editor messages* and *action messages*. The latter will be discussed in section 2.4.4. An *editor event* consists of a new value for a model represented by the editor, and a unique task identification number that makes it possible to relate the view in the client to a task in the code [PLM+12]. An *editor response* informs the client of the latest state of

an editor. This state contains a local value for editing, a shared value for viewing (which can be done by different tasks), a description and if the editor is viewing or editing.

2.4.4 The step combinator

As discussed in section 2.2, observers of a task are only able to inspect its *task value*. The step combinator, one of the core combinators of *iTasks*, keeps an eye on the task value of a running task to decide whether or not to step to a different task. Once a new task is started, the old one is no longer needed; the step combinator is used to perform tasks in sequence.

The *iTasks* step combinator differentiates between three categories of task steps [PLM+12]: (1) steps resulting from the user selecting an action (e.g. a “next” button), (2) steps based on the current task value, (3) steps handling exceptions. The first two are accompanied by a predicate over the observed task value. A step is only taken if the predicate holds. For actions, this means the availability is determined by the model (e.g. an action may not be available if mandatory data is missing). Exceptions are propagated by the step combinator, unless there is a rule of type (3) that contains a handler for that specific exception.

The step combinator provides the client with an *action response*. This action response consists of a list of actions and whether or not they are enabled. A client can respond with an *action event*, which informs the server which action is triggered by the user. Steps that do not depends on the user (i.e. (2) and (3)) are prioritized over steps that do (i.e. (1)). In [PLM+12], steps that do not depend on the user are referred to as “triggers”. When no step can be taken, the process is repeated on the rewritten task.

2.4.5 The parallel combinator

Tasks can not only be performed in sequence (section 2.4.4) but also in parallel. Like performing tasks in sequence, *iTasks* has a single combinator for performing tasks in parallel, named “parallel”. Tasks can be performed in parallel when there is no specific order in which they have to be performed. In *iTasks*, there is a distinction between tasks that can be performed in parallel [PLM+12]: *detached* tasks are distributed to different users, while *embedded* tasks are performed by the current user. In case of the former, it is possible to specify a specific user or role that is to perform the task.

Tasks run in parallel can inspect each other’s progress, because *iTasks* stores their task values in a *shared task list*. The parallel combinator emits a list of values with the timestamp of the event that resulted in their production. This allows something like the step combinator (section 2.4.4) to determine the progress of a parallel operation as a whole. This task list is also used to allow dynamic creation and deletion of tasks running in parallel. More information on this can be found in [Lij13a]. Initially, the task list filled with the tasks that are run in parallel, with a “no value” result. The parallel combinator evaluates all tasks of this list, storing their *task results* and their rewritten version in the *shared task list* along the way. Exceptions that may arise during evaluation are propagated over all tasks that are run in parallel.

The following type definition for the core parallel combinator is provided in [PLM+12]:

```
parallel :: d -> [(ParallelTaskType, ParallelTask a)]
             -> Task [(TimeStamp, Value a)] | descr d & iTask a

:: ParallelTaskType = Embedded | Detached ManagementMeta
:: ManagementMeta   = { worker :: Maybe User
                      , role   :: Maybe Role
                      , ...
                      }
:: ParallelTask   a :== SharedTaskList a -> Task a
:: SharedTaskList a :== ROShared (TaskList a)
:: TaskList       a   = { state :: [Value a]
                        , ...
                        }
```

Multiple parallel combinators have been derived from this core parallel combinator, for example the ones below.

```
(-||-) infixr 3 :: !(Task a) !(Task a) -> (Task a)     | iTask a
(-&&-) infixr 4 :: !(Task a) !(Task b) -> (Task (a,b)) | iTask a & iTask b
```

`(-||-)` performs two tasks in parallel, resulting in the first task that produces a stable value. `(-&&-)` also performs two tasks in parallel, but results in a task that yields the results of both [Lij13a]. Because an observer of a parallel task can observe the task values obtained so far (and the timestamp of the event that caused their production), the definition of the shown derived combinators is relatively straight-forward.

Chapter 3

µTasks: a general-purpose Task-Oriented Programming framework

In chapter 2, the Task-Oriented Programming paradigm and the framework from which it originated, *iTasks* (section 2.4), were introduced. *iTasks* aims to facilitate the development of "interactive, distributed, multi-user applications" [AKP13]. Because *iTasks* is developed around achieving this goal, its semantics and components are closely related to this problem domain. An example of this are *editors*, discussed in section 2.4.3.

The problem domain considered in this thesis is different. While the systems under consideration are interactive, this interactivity involves only other systems. User interactivity is not one of the properties we wish to facilitate, unlike *iTasks*. Additionally, the developed applications are not distributed; they run on a single system. Because of this, we introduce *µTasks*: a general-purpose Task-Oriented Programming framework. The key aspects in which this framework differs from *iTasks* are:

1. *iTasks* is driven by events originating from one or multiple clients (section 2.4.1). In *µTasks* there is no driver; tasks are simply state transformers that calculate a result. The reason for this is that *µTasks* applications are not distributed, removing the necessity to coordinate the work of individual workers through something like events. By consequence there is no client-server architecture in *µTasks*.

2. *iTasks* is capable of automatically generating user interfaces from type definitions through *editors* (section 2.4.3). There is no such thing in *µTasks*, because user interactivity is not considered part of the problem domain. This means there is no such thing as views, models and actions in *µTasks*.

3. In *iTasks*, performing tasks in parallel allows them to be performed by different users or user groups spread over multiple clients. Performing tasks in parallel in *µTasks* means performing them simultaneously on the same system. Conceptually, all parallel sub-tasks can be seen as *embedded* in *µTasks* (section 2.4.5).

µTasks resembles *iTasks* in the fact that it uses many of same concepts (albeit with sometimes different semantics), such as a step combinator, a parallel combinator and shared data sources. Because it is a Task-Oriented Programming framework, it adheres to the principles of Task-Oriented Programming (section 2.2).

The remainder of this chapter is spend on discussing the components of the *µTasks* framework. It is important to emphasize that this framework is built from the bottom up; it has no direct relation to *iTasks*. Like *iTasks*, *µTasks* is written in a purely functional language. All code snippets presented in this chapter are written in the *Haskell* programming language [Lip11]. There was no particular reason for choosing this language. Any other pure functional language, for example *Clean* [Pla01], could have been used.

3.1 Task-related types and definitions

A `Task` is a building block that computes an observable value of some type `a`. It does this in the context of an `Environment` it can manipulate (e.g. modifying a file on the file system). This directly translates into the following type definition:

```
type Task a = Environment -> (TaskResult a, Environment)
```

Note that this definition bears resemblance to a state monad (section 2.3.6). For now, the exact definition of `Environment` is not important. The takeaway message is that a `Task` produces a `TaskResult` in the context of an `Environment` it possibly updates. As shown in figure 2.1, various task values can be observed from a `Task`. If a value is not final (i.e. `Stable`), the `Task` also produces a rewritten version of itself, representing the `Task` that performs the remaining work that is to be done. This results in the following `TaskResult` definition:

```
data TaskResult a = NoValue         (Task a)
                  | StableValue   a
                  | UnstableValue a (Task a)
```

3.2 Basic task functions

Using the definitions from section 3.1, several basic task functions can be constructed. One essential function is the `return` function which allows lifting a value to the `Task` domain:

```
return :: a -> Task a
return value = \env -> (StableValue value, env)
```

A `Task` that calculates the value 42 (and therefore has the type `Task Int`) can now be defined as follows:

```
exampleTask :: Task Int
exampleTask = return 42
```

For combinator definitions (discussed later) it is useful to lift values to any kind of `TaskResult`, and even to produce a `Task` directly from a `TaskResult`:

```
stable   value      = fromTaskResult (StableValue   value     )
unstable value task = fromTaskResult (UnstableValue value task)
noValue        task = fromTaskResult (NoValue             task)

fromTaskResult :: TaskResult a -> Task a
fromTaskResult result = \env -> (result, env)
```

In some cases it is also useful to provide a `Task` that effectively does nothing:

```
emptyTask :: Task ()
emptyTask = stable ()
```

Predicates and accessor functions on `TaskResults`, such as `value` and `isStable`, are also defined. Because their implementation is trivial we will only present their type definitions:

```
value       :: TaskResult a -> a
isStable    :: TaskResult a -> Bool
isException :: TaskResult a -> Bool
```

Another one of these functions is `followup`, for which it worth mentioning that it requires a default `Task` as not all `TaskResult` options contain a follow-up `Task`:

```
followup :: TaskResult a -> Task a -> Task a
followup (NoValue         task) _   = task
followup (UnstableValue _ task) _   = task
followup _                      def = def
```

For similar reasons, `value` is a partial function. Once a `Task` is defined, one might want to continuously evaluate it until a `Stable` value is produced. This is similar to how *iTasks* rewrites its top-level task (section 2.4.1). The `eval` function does just that:

```
eval :: Task a -> Environment -> (a, Environment)
eval eval_a env =
    let (tra, env') = eval_a env
     in case tra of
            StableValue v -> (v, env')
            _             -> eval (followup tra eval_a) env'
```

Finally, several basic tasks are defined. Their implementation is not really relevant to this paper so only their type signatures are given:

```
printInfo   :: ToString a => a -> Task a
printWith   :: ToString b => (a -> b) -> a -> Task a
readInfo    :: Read     a => Task a
sleep       :: Integer -> Task ()
readLine    :: Task String
hReadLine   :: Handle -> Task String
currentTime :: Task UTCTime
```

Using these functions we can perform basic `Tasks`, like printing "Hello World!", or printing a person's age. Note that in the example below, `printPersonAge` results in a `Task` of type `Task Person` as opposed to `Task Int` (as `age` is of type `Int`).

```
helloWorld = Task String
helloWorld = printInfo "Hello World!"

data Person = Person { name :: String
                     , age  :: Int
                     }

printPersonAge :: Person -> Task Person
printPersonAge person = printWith age person
```

3.2.1 List-based tasks

`Tasks` may produce several `Unstable` values before (possibly) producing a `Stable` one. In the end, the `Task` produced a collection of values. This gave birth to the idea of doing it the other way around: creating a `Task` from a collection of values:

```
fromList :: [a] -> Task a
fromList []     = noValue (fromList [])
fromList [x]    = stable x
fromList (x:xs) = unstable x (fromList xs)
```

If a `Task` can be created from a list, it consequently becomes possible to create a `Task` from other `Foldable` types as these implement a `toList` function:

```
fromFoldable :: Foldable f => f a -> Task a
fromFoldable = fromList . toList
```

List functions can now easily be translated into their `Task` equivalent. For example: an `iterateTask` function that computes $x, f(x), f(f(x))$ and so on can be defined as:

```
iterateTask :: (a -> a) -> a -> Task a
iterateTask f x = fromList (iterate f x)
```

Note that this function will never produce a `Stable` value. This is no problem due to the lazy evaluation property (section 2.3.3) of the host language. For example, a `Task` that calculates all pairs of successive fibonacci numbers can be defined using `iterateTask` as follows:

```
fibPairs :: Task (Int, Int)
fibPairs = iterateTask (\(a,b) -> (b, a + b)) (0,1)
```

One may observe that if the function `f` that is provided to `iterate` produces two subsequent values that are equal, all following values will be too. For this reason, a different version of `iterateTask` is defined that stops the calculation (i.e. yield a stable value) upon observing this pattern:

```
iterateLimTask :: Eq a => (a -> a) -> a -> Task a
iterateLimTask f x = fromList (limit (iterate f x))
    where
    limit (x:y:xs)
        | x == y    = [x]
        | otherwise = x : limit (y:xs)
    limit xs        = xs
```

3.3 The step combinator

Now that tasks and several basic functions that operate on either `Tasks` or `TaskResults` have been defined, steps towards tasks interaction can be made. Tasks become a lot more interesting when they can be combined to form new tasks. This allows for a top-down approach when designing software using the Task-Oriented Programming paradigm.

Like *iTasks* (section 2.4.4) we will define a step combinator (`>>*`) as a core combinator for performing tasks in sequence. This step combinator will perform a single step with a task, and then determines how to proceed based on the observed `TaskResult`. This relation between a `TaskResult` and the `Task` representing the next step can be expressed by a function we will refer to as a *continuation function* from now on. The value produced by the continuation function (if any) shall be referred to as a *continuation*. Note that this is essentially different from the `followup` function presented in section 3.2. `followup` extracts the follow-up task contained in a `TaskResult`, while a continuation function maps `TaskResults` to `Tasks`. A `TaskResult` has at most 1 follow-up, but an infinite number of continuation functions can exist that each produce a different continuation for this `TaskResult`. The type definition of a continuation function is given next.

```
type TaskCont a b = TaskResult a -> Maybe b
```

There are two things to point out here:

1. The resulting continuation is a `Maybe` type. This means that a continuation function may not be able to provide a continuation for the supplied `TaskResult`.
2. If the continuation function results in a continuation, it is of type `b` as opposed to the type `Task b` one might expect. This more general type is only used to make continuations more flexible.

The step combinator of μ*Tasks* is implemented as follows:

```
type TaskProducer a b = Task a -> Task b

(>>*) :: Task a -> [TaskCont a (TaskProducer a b)] -> Task b
(>>*) eval_a conts = eval
    where
    eval env = let (tra, env') = eval_a env
                   matches     = mapMaybe ($ tra) conts
                   nta         = followup tra eval_a
               in case matches of
                     (c:_) -> noValue (c nta) env'
                     []    -> noValue (nta >>* conts) env'
```

In this definition, the `($)` function is the application operator and `mapMaybe` is a version of `map` that can throw out elements:

```
($)      :: (a -> b) -> a -> b
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

The supplied `Task eval_a` is evaluated under the `Environment env`, after which the observed `TaskResult tra` is supplied to the list of continuation functions to obtain a list of continuations. If there is a continuation, it will be used to determine the task to step to. If none of the continuation functions produce a continuation, the step operation is repeated on `eval_a`'s follow-up if it has one, or `eval_a` otherwise (which should not happen as it produces an infinite loop). There are a couple of things that are worth pointing out:

1. Instead of a single evaluation function, multiple are given. This allows expressing a more complex continuation as a collection of relatively simple ones.
2. Regardless of whether or not there was a continuation, the step combinator results in a `NoValue TaskResult` containing the task to step to as opposed to directly evaluating this task. Doing the latter would be counterintuitive as this results in multiple steps within a single evaluation. It is important to make steps small, especially when in a parallel context as discussed in section 3.6.
3. The continuation functions produce continuations of type `TaskProducer a b` instead of `Task b`. This is a more general form of a continuation that allows the next step to be parametrized with the rewritten version of the task supplied to the step combinator. This is best illustrated by example:

   Say we have a `Task ta` that produces an endless amount of values (for example because it is defined using `iterateTask` shown in section 3.2) and a `Task tb` that is parametrized with these values. Defining a combinator $\oplus$ so that `ta` $\oplus$ `tb` rewrites to `ta'` $\oplus$ `tb`, where `ta'` is the follow-up of `ta`, is not possible when a continuation is of type `Task b`. Section 3.4.3 presents the `parametrizes` function, which is an implementation of the $\oplus$ used in this example.
4. The first matching continuation `c` is selected to continue: the ordering of continuations determines their precedence.

With a step combinator defined, several functions can be declared that aid in constructing continuation functions. For example, we can now define an `ifStable` function that matches when a stable value is observed:

```
ifStable :: (a -> TaskProducer a b) -> TaskCont a (TaskProducer a b)
ifStable f (StableValue v) = Just (f v)
ifStable _ _               = Nothing
```

Several similar functions can be defined. Because their implementation is straightforward, the listing below is restricted to their types only:

```
hasValue     ::                 (a -> TaskProducer a b) -> TaskCont a (TaskProducer a b)
ifUnstable   ::                 (a -> TaskProducer a b) -> TaskCont a (TaskProducer a b)
ifValue      :: (a -> Bool) -> (a -> TaskProducer a b) -> TaskCont a (TaskProducer a b)
onTaskResult ::     (TaskResult a -> TaskProducer a b) -> TaskCont a (TaskProducer a b)
always       ::                      TaskProducer a b  -> TaskCont a (TaskProducer a b)
```

Section 3.4 presents several `Task` combinators that are expressed using these functions and the step combinator.

3.4 Step combinator instances

A variety of useful combinators can be defined using the step combinator presented in section 3.3. Combined with the functions to construct continuation functions, this allows for concise combinator definitions. This section categorizes these combinators and discusses them by category.

3.4.1 Sequential combinators

As the name implies, sequential combinators perform one task after the other. When to proceed with the `Task` on the right-hand side is determined by the `TaskResult` observed from the `Task` on the left-hand side. Additionally, the result of the `Task` on the left-hand side can be supplied to the `Task` on the right-hand side. Each combination of these choices result in its own combinator, outlined in table 3.1.

Proceed when / Result	Convey	Discard
`Stable`	`>>>`	`>>|`
`Stable` or `Unstable`	`>>!`	`>>-`

Table 3.1: Overview of sequential combinators

The combinators that pass on their result are expressed using the step combinator. `(>>>)` passes on the value once a stable is observed, while `(>>!)` passes on the first value that is observed.

```
(>>>) :: Task a -> (a -> Task b) -> Task b
(>>>) ta tb = ta >>* [ ifStable (const . tb) ]

(>>!) :: Task a -> (a -> Task b) -> Task b
(>>!) ta tb = ta >>* [ hasValue (const . tb) ]
```

The remaining combinators are instances of these more general combinators.

```
(>>|) :: Task a -> Task b -> Task b
(>>|) ta tb = ta >>> const tb

(>>-) :: Task a -> Task b -> Task b
(>>-) ta tb = ta >>! const tb
```

For example, we can now combine the basic `Task` functions (section 3.2) `printInfo` and `readLine` using these combinators to create a `Task` that asks for your name and then greets you:

```
greetUser :: Task String
greetUser = printInfo "Enter your name: "          >>|
            readLine                               >>>
   \name -> printInfo ("Welcome, " ++ name ++ "!")
```

3.4.2 Repetition combinators

Instead of performing a `Task` once, one may want to repeat it. Or more generally: repeat it while a given predicate holds. This is exactly what the `(<!)` combinator is for:

```
(<!) :: Task a -> (a -> Bool) -> Task a
(<!) ta pred = ta >>* [ ifValue  (not . pred) (const . stable)
                      , ifStable              (const . const $ ta <! pred)
                      ]
```

If a value is observed for which the predicate `pred` does not hold, the value is yielded as a `StableValue`. If this is not the case (recall that the continuations are listed in order of precedence) and a `StableValue` is observed, the original `Task` is restarted under the same repetition condition. Endlessly repeating a `Task` is now simple, as it is more specific:

```
forever :: Task a -> Task a
forever ta = ta <! const True
```

A `Task` that repeatedly asks for a new password until one meeting the security requirements is entered can now be defined as follows:

```
type Password = String

enterValidPassword :: Task Password
enterValidPassword = enterPassword <! not . isValidPassword
    where
    enterPassword :: Task Password
    enterPassword = printInfo "Enter your password: " >>|
                    readLine
```

```
    -- A password is valid if it contains at
    -- least one digit and one alpha symbol.
    isValidPassword :: Password -> Bool
    isValidPassword password = any isDigit  password  &&
                               any isAlpha  password
```

3.4.3 Miscellaneous combinators

Often the context in which a `Task` is performed requires a specifically typed `TaskResult`. Because of this, a transformation combinator is introduced:

```
(?) :: Task a -> (a -> b) -> Task b
(?) task f = task >>* [ onTaskResult (const . fromTaskResult . fmap f) ]
```

`fmap` is a higher-order function (section 2.3.4) that is part of the `Functor` type class. Type constructors (e.g. lists or trees) can be made member of this class, so that functions can be applied to their elements. For example, a list of integers can be transformed into a list of strings by applying `fmap` to it with a `toString` function. The instance for `TaskResult` is defined as follows:

```
instance Functor TaskResult where
    fmap f (StableValue   v  ) = StableValue   (f v)
    fmap f (UnstableValue v t) = UnstableValue (f v) (t ? f)
    fmap f (NoValue         t) = NoValue             (t ? f)
```

Another `Task` pattern that can be captured using a combinator is the repeated parametrization of a `Task tb` by the values produced by a `Task ta`. This combinator corresponds to the $\oplus$ combinator discussed in section 3.3 and is named `parametrizes`:

```
parametrizes :: Task a -> (a -> Task b) -> Task b
parametrizes ta tb = ta >>* [ ifStable   (const   . tb)
                            , ifUnstable (newTask . tb)
                            ]
    where
    newTask tb' ta' = tb' >>> flip unstable (ta' `parametrizes` tb)
```

Note that the `(>>>)` combinator is used in the `newTask` definition. This means that `tb` (and thus `tb'`) is expected to produce a `Stable` value. For example, we can define a `Task printUntil` that prints 0 till n using a predefined view:

```
successorList :: (Enum a, Num a) => Task a
successorList = iterateTask succ (fromInteger 0)

-- e.g: "printUntil 2" outputs: "[0] [1] [2]" on separate lines.
printUntil :: Int -> Task Int
printUntil n = successorList ‘parametrizes‘ printWith view <! pred
    where
    pred    = (>) n
    view n = "[" ++ show n ++ "] \n "
```

3.5 Exception handling

Up until now `Tasks` have been able to produce different types of `TaskResults`, but none of these represent anomalous or exceptional conditions. To support this, an exception handling mechanism is implemented. An exception can be thrown and must be propagated until caught. When a `Task` is propagated over it is not evaluated. Several (mostly minor) modifications and additions must be made to the framework to enable this:

1. Addition of the `Exception TaskResult` and associated definitions:

   ```
   data TaskResult a = Exception Dynamic String
                     | ...

   instance Functor TaskResult where
       fmap f (Exception e m) = Exception e m
       ...

   isException :: TaskResult a -> Bool
   isException (Exception _ _) = True
   isException _               = False
   ```

   The `Dynamic` type of *Haskell* represents a monomorphic value dynamically and allows throwing an arbitrary exception type. The `String` component of an `Exception` is used to store a message that is used for display when the `Exception` is uncaught.

2. Modification of the step combinator so it propagates exceptions when not caught. Recall the definition presented in section 3.3. It contained the following match statement to deal with the scenario that there was no matching continuation:

   ```
   [] -> noValue (nta >>* conts) env’
   ```

   In order to support exceptions, this is modified to the following:

   ```
   [] -> case tra of
             Exception m e -> fromTaskResult (Exception m e) env’
             _             -> noValue (nta >>* conts) env’
   ```

   Note that this allows defining a continuation based on exceptions (because checking whether or not there is a matching continuation occurs first) without having to modify existing continuations because the step combinator itself propagates them.

3. Modification of the `eval` function (section 3.2) so that it produces a run-time error when encountering an `Exception`. More specifically, the following match was added for the observed `TaskResult`:

```
Exception _ m -> error $ "uncaught exception: " ++ m
```

4. Addition of a continuation function to produce a continuation when observing an `Exception`:

```
onException :: Typeable e => (e -> Task a) -> TaskCont a (TaskProducer a a)
onException handler (Exception e _) = fmap (const . handler) (fromDynamic e)
onException _ _                     = Nothing
```

   A `handler` is provided. This `handler` is parametrized with the data contained in the observed `Exception`. The type of this data is simply `e` as it can be arbitrary: the context determines the concrete type at runtime.

   `fromDynamic` results in a `Maybe` type (so the `fmap` function used here is the one for the `Maybe` instance). By consequence, this means that when `fromDynamic` is unable to convert the `Dynamic e` back to a value of type `e`, `onException` results in `Nothing`, indicating that it cannot provide a continuation. This means that a handler defined for an exception of a different type than the one thrown will not be produced by this continuation function in the step combinator.

5. Addition of the `throw` and `try` combinators:

```
throw :: (Typeable e, ToString e) => e -> Task e
throw e = fromTaskResult $ Exception (toDyn e) (toString e)

try :: Typeable e => Task a -> (e -> Task a) -> Task a
try task handler = task >>* [ ifStable (const . stable)
                            , onException handler
                            ]
```

   Note that when throwing an exception a message is not included. The message is currently obtained by converting the thrown value to a `String`. This means types that can be thrown must have a `String` conversion function. The message is currently not used for custom messages. The `try` function is parametrized with a `handler` which is sometimes called a “catch block” in other languages. The following example shows how the `try` and `throw` functions can be used to deal with exceptional behavior:

```
data NumberException = DivideByZeroException
                     | NumberOverflowException
                     | NumberUnderflowException
                     deriving (Show)

divideBy :: Int -> Int -> Task Int
divideBy x 0 = throw DivideByZeroException
divideBy x y = stable (x `div` y)
```

```
example :: Task String
example = try division handler
    where
    division :: Task String
    division = 42 `divideBy` 0 >>> \result -> printInfo "result: " >>|
                                              printInfo result     >>|
                                              printInfo ". \n"

    handler :: NumberException -> Task String
    handler DivideByZeroException = printInfo "A divide by zero occurred"
    handler _                     = printInfo "Another exception occurred"
```

3.6 Parallelization

Instead of performing `Tasks` in sequence, one might also want to perform them in parallel. As mentioned in the introduction of this chapter, this parallel combinator is different from the one used in *iTasks* (section 2.4.5). When performing tasks in parallel in *µTasks*, we are rewriting tasks as a group as if it were a single rewrite. Every parallel sub-task gets rewritten at most once per evaluation of the parallel combinator. This is important when nesting parallel combinators. For example, consider the following parallel combinator which performs two `Tasks` in parallel and results in the first `Task` that produces a stable value. The implementation of this combinator is not important at this point and will be given further on in this section.

```
(-||-) :: Task a -> Task a -> Task a
```

Say there is a `Task` `t`α that infinitely outputs α, one per step. Intuitively, the following should then endlessly output `abc`:

```
ta -||- tb -||- tc   -- Equal to: (ta -||- (tb -||- tc))
```

This is only possible when a single rewrite of a parallel task ends when every parallel sub-task is rewritten at most once. It is for the same reason that the parallel combinator cannot be expressed using the step combinator 3.3. If it were, rewriting of a parallel task would be limited to rewriting a single one of its sub-tasks as the step combinator only performs a single step. In the above example this would result in `ta` being rewritten twice as much as `tb` and `tc` individually.

Like *iTasks*, parallel combinators in *µTasks* are based on a single `parallel` combinator. This combinator must be sufficiently flexible to be able to express the various parallel combinators. For example `(||-)`, which performs two `Tasks` in parallel and results in the `Task` on its right-hand side, even when the `Task` on the left-hand side produced a stable first. The parallel combinator used by *µTasks* is based on an early version of the `parallel` combinator used in `iTasks` [Lij13a] and is parametrized by the following arguments:

1. `predOK :: [a] -> Bool`
   A predicate that is evaluated each time a `Task` is finished (i.e. produced a `StableValue`). It is parametrized with a list containing the accumulated results so far.

2. `someDone :: ([a],[Task a]) -> b`
   A function that is used to compute the result of the parallel combinator when it terminated due to `predOK` being satisfied. It is parametrized with a tuple containing the results obtained so far, and a list of `Tasks` that perform the remaining work of the `Tasks` that did not yet finish.

3. `allDone :: [a] -> b`
   Similar to `someDone`, except that this function is called to compute the result when all parallel tasks have terminated without `predOK` ever being satisfied.

4. `tasks :: [Task a]`
   The list of tasks to be performed in parallel.

The combinator rewrites each task in `tasks`, one by one. If this step results in a `StableValue`, the combinator checks whether or not `predOK` holds for the new set of accumulated `Stable` values. If this is the case, the final result is calculated using `someDone`. If all `Tasks` produced a `Stable` value but `predOK` was never satisfied, then the resulting value is calculated using `allDone`.

The function is implemented using a helper function named `parallel'` that contains two accumulators in addition to the aforementioned parameters: one containing the results so far and another containing the follow-up `Tasks` of `Tasks` that have already been rewritten this step. The complete `parallel` implementation looks as follows:

```
1   parallel = parallel' [] []
2
3   parallel' :: [a]                      -- stables   (accumulator)
4             -> [Task a]                 -- followups (accumulator)
5             -> ([a] -> Bool)            -- predOK
6             -> (([a],[Task a]) -> b)    -- someDone
7             -> ([a] -> b)               -- allDone
8             -> [Task a]                 -- tasks
9             ->  Task b
10
11  -- All tasks have completed: use allDone to obtain the result.
12  parallel' stables [] _ _ allDone [] = stable $ allDone stables
13
14  -- All tasks have been stepped this step, but they did not
15  -- yet all finish. Set "followups" as the tasks to be evaluated the
16  -- next step. Reverse the list because the tasks were added at the head.
17  parallel' stables followups predOK someDone allDone [] =
18      noValue $ parallel' stables [] predOK someDone allDone (reverse followups)
19
20  -- There is at least one task to step.
21  parallel' stables followups predOK someDone allDone (x:xs) = \env ->
22      let (tr,env') = x env    -- Perform a step.
23       in case tr of
24
25          -- Case 1: Evaluation led to a stable value.
26          StableValue v ->
27              -- Add the value to the accumulator and
28              -- check whether or not the predicate holds.
29              let stables' = v : stables
```

```
30              in if predOK stables'
31                     -- The predicate is satisfied.
32                     -- Use someDone to obtain the result.
33                     then (stable $ someDone (stables', followups ++ xs)) env'
34
35                     -- The predicate did not hold.
36                     -- Continue with the rest of the tasks.
37                     else parallel' stables' followups predOK someDone allDone xs env'
38
39          -- Case 2: The task threw an exception: propagate it.
40          Exception m e -> fromTaskResult (Exception m e :: TaskResult b) env'
41
42          -- Case 3: Evaluation did not result in a stable value (noValue or unstable).
43          -- Add the followup to the corresponding accumulator, and continue
44          -- with the remaining tasks.
45          _  -> let followups' = followup tr x : followups
46                in parallel' stables followups' predOK someDone allDone xs  env'
```

With this `parallel`, an `anyTask` combinator can be expressed. This combinator performs a list of `Tasks` in parallel, yielding the first `Task` that produces a stable value:

```
anyTask :: [Task a] -> Task a
anyTask = parallel (not . null) (head . fst) undefined
```

The *Haskell* function `undefined` is a function that never completes successfully (often referred to as "bottom"). When there is an `anyTask`, it makes sense to also define an `allTasks` that performs `Tasks` in parallel until all of them have terminated:

```
allTasks ::  [Task a] -> Task [a]
allTasks = parallel (const False) undefined id
```

With these combinators, the implementation of the aforementioned `(-||-)` combinator is trivial as shown below. The remaining parallel combinators will be presented in section 3.6.2.

```
(-||-) :: Task a -> Task a -> Task a
(-||-) ta tb = anyTask [ta,tb]
```

3.6.1 Timeout

Having a `sleep` `Task` (section 3.2) as well as combinators for parallelism, allows defining a timeout mechanism. The `withTimeout` function is part of the framework and performs a `Task` with a timeout of a given amount of milliseconds. The function results in the result of the `Task`, but only if it was obtained before the timeout expired. If this is not the case, an optional given value `def` will be yielded.

```
withTimeout :: Integer -> Task a -> Maybe a -> Task (Maybe a)
withTimeout ms task def = task ? Just -||- sleep ms ? const def
```

3.6.2 Order maintenance

While the `parallel` combinator presented earlier behaves as desired, there is one devious aspect related to ordering. The problem is best illustrated by an example. Say we want to implement a parallel combinator that performs two `Tasks` in parallel, and results in a tuple containing both results. Its implementation could look as follows:

```
(-&&-) :: Task a -> Task b -> Task (a,b)
(-&&-) ta tb = allTasks [ta ? Left, tb ? Right] ? all
    where
    all [Left a, Right b] = (a,b)
```

Note that the transformation combinator is used to equalize the types of `ta` and `tb`, namely to `Task (Either a b)`. This allows them to be placed in a (heterogenous) list that can be supplied to the `allTasks` function presented earlier. The `all` function is responsible for transforming back the result of `allTasks` and illustrates the problem with `parallel`. The way `all` is defined now is erroneous due to non-exhaustive patterns. `tb` may finish before `ta`, meaning the list supplied to `all` would be `[Right b, Left a]` instead.

To allow the implementation above instead of having to define `all` in a way it is defined for every possible ordering of its argument, `parallel` is modified to preserve ordering. The modifications aim to guarantee the following properties:

1. For any two items in the list of stables (supplied to `someDone` or `allDone`) with index i and j, yielded by `Tasks` located at index n and m of the initial `Task` list respectively: $i < j$ if and only if $n < m$.
2. For any two items in the list of remaining `Tasks` (supplied to `someDone`) with index i and j, being the follow-up of the `Tasks` with index n and m in the initial `Task` list respectively: $i < j$ if and only if $n < m$.

To achieve this, every `Task` supplied to `parallel` is assigned a (locally) unique integral identifier. This means `parallel'` is no longer parameterized with a list of `Tasks` (i.e. `[Task a]`), but with a mapping from a `Task` identifier to a `Task` (i.e. `Map Int (Task a)`). Several operations are defined for the `Map` type in *Haskell*[1] and conveniently, most of these turn out to enforce ordering. The relevant functions that have this ordering property are given below.

1. `elems :: Map k a -> [a]`
   Return all elements of the map in the ascending order of their keys.
2. `elemAt :: Int -> Map k a -> (k, a)`
   Retrieve an element by its index, i.e. by its zero-based index in the sequence sorted by keys.
3. `deleteAt :: Int -> Map k a -> Map k a`
   Delete the element at index, i.e. by its zero-based index in the sequence sorted by keys.
4. `insert :: Ord k => k -> a -> Map k a -> Map k a`
   Insert a new key and value in the map. If the key is already present in the map, the associated value is replaced with the supplied value.

[1] http://hackage.haskell.org/package/containers-0.5.7.1/docs/Data-Map-Strict.html

5. `union :: Ord k => Map k a -> Map k a -> Map k a`
   The expression `(union t1 t2)` takes the left-biased union of t1 and t2. It prefers t1 when duplicate keys are encountered.

Using these functions, `parallel` and `parallel'` are modified to the following:

```
parallel tasks = parallel' empty empty $ fromList (zip [0..] tasks)

parallel' :: Map Int a               -- stables   (accumulator)
          -> Map Int (Task a)        -- followups (accumulator)
          -> Map Int (Task a)        -- tasks
          -> ([a] -> Bool)           -- predOK
          -> (([a],[Task a]) -> b)   -- someDone
          -> ([a] -> b)              -- allDone
          ->  Task b
parallel' stables followups tasks predOK someDone allDone


-- There are no more tasks to evaluate this step.
| null tasks =
    if null followups

        -- There are no more followups for a next step,
        -- calculate the result using allDone. "elems"
        -- orders by key.
        then stable  $ allDone (elems stables)

        -- All tasks have been stepped this step, but they did
        -- not yet all finish, nor did predOK hold. Set "followups"
        -- as the tasks to be evaluated during the next step.
        else noValue $ parallel' stables empty followups predOK someDone allDone

-- There is at least one task to step.
| otherwise = \env ->
    -- The next task to step is the one in "tasks" with the lowest
    -- task identifier. Abbrviate the remainder of the map as "xs".
    -- Once obtained: step it.
    let (i,x)      = elemAt   0 tasks
        xs         = deleteAt 0 tasks
        (tr, env') = x env

     in case tr of

       -- Case 1: Evaluation led to a stable value.
       StableValue v ->

           -- Add the value to the accumulator.
           let stables' = insert i v stables

               -- The values obtained so far, ordered by the
               -- identifiers of the tasks that produced them.
               results  = elems stables'

            -- Test predOK
            in if predOK results
```

```
50
51                      -- The predicate holds, so we can stop. Use someDone
52                      -- to compute the result. The union of "followups" and
53                      -- "xs" represent the remaining work. "elems" ensures
54                      -- correct ordering.
55                      then (stable $ someDone (results, elems (union followups xs))) env'
56
57                      -- The predicate did not hold: continue with
58                      -- the remaining tasks.
59                      else parallel' stables' followups xs predOK someDone allDone  env'
60
61          -- Case 2: The task threw an exception: propagate it.
62          Exception m e -> fromTaskResult (Exception m e :: TaskResult b) env'
63
64          -- Case 3: Evaluation did not result in a stable value.
65          --         Add the follow-up task to the accumulator.
66          _  -> let followups' = insert i (followup tr x) followups
67               in parallel' stables followups' xs predOK someDone allDone env'
```

The earlier implementation for `(-&&-)` works correctly using this `parallel` implementation. The implementations for `(-||)` and `(||-)` are similar:

```
(||-) :: Task a -> Task b -> Task b
(||-) ta tb = parallel [ta ? Left, tb ? Right] (not . null) id undefined >>> res
    where
    res ([Right v], _     ) = stable v
    res (_          , [tb']) = tb' ? fromRight

(-||) :: Task a -> Task b -> Task a
(-||) ta tb = parallel [ta ? Left, tb ? Right] (not . null) id undefined >>> res
    where
    res ([Left v], _     ) = stable v
    res (_         , [ta']) = ta' ? fromLeft
```

3.7 Shared data sources

Using the combinators defined in section 3.6, `Tasks` are able to run in parallel. `Tasks` that run in parallel are not (and need not be) aware that they are being performed in a parallel context; they are isolated from each other. However, situations exist where `Tasks` that run in parallel need to communicate with each other or access a common resource. For this purpose, Shared Data Sources (or `SDS`) are introduced. These are similar to the ones used in *iTasks* [PLM+12] (section 2.4.2).

Shared data sources provide an abstract interface for reading and writing. A timer for example, might be represented as a shared data source where one can read the elapsed time and write timer commands such as `Start`, `Stop`, `Reset`, and so on. This example also shows that the type of values read from a shared data source can differ from the type of the values written to it. Like *iTasks* (section 2.4.2) we make a distinction between internal and external shared data sources. Internal shared data sources are used for sharing data between tasks. All other data sources are considered external. A shared data source is named `ReadWriteShared` and is defined as a record:

```
type ShareId = Int

data ReadWriteShared r w = ReadWriteShared
    { shareId :: ShareId
    , read    ::      Environment -> (r, Environment)
    , write   :: w -> Environment ->     Environment
    }
```

The `read` and `write` functions represent the abstract interface discussed earlier. `shareId` is an identifier used to uniquely identify the shared resource which is needed for internal shared data sources as we will see later. Of course, this type can be used to create less general types:

```
type Shared          rw = ReadWriteShared rw rw
type ReadOnlyShared  r  = ReadWriteShared r  ()
type WriteOnlyShared w  = ReadWriteShared () w
```

A shared data source has a state. For example, different data may be read from a shared queue on subsequent calls, depending on the queue's contents. For internal shared data sources, this state needs to be stored at a location accessible to all `Tasks`. There is one obvious candidate for this: the `Environment`. Containers are heterogeneous in most if not all functional programming languages, so in order to store different shared resources in a single collection, they are converted into a dynamically typed value of type `Dynamic`:

```
type SharedStore = Map ShareId Dynamic

data Environment = Environment
                   { shares :: SharedStore
                   , -- Other values omitted.
                   }
```

Every internal shared data source reserves a slot in the shared store. The `read` and `write` functions simply read from and write to this store. This is shown in the `sharedStore` function, used for creating an internal shared data source:

```
sharedStore :: Typeable r => ShareId -> r -> (w -> r) -> ReadWriteShared r w
sharedStore sid init conv = ReadWriteShared { read  = reader
                                            , write = writer
                                            }
    where
    reader   env = let mem = shares env
                   in case lookup sid mem of
                             Just e  -> ((fromJust . fromDynamic) e, env)
                             Nothing -> (init                      , env)

    writer w env = let mem = shares env
                       w' = toDyn (conv w)
                   in env { shares = alter (const (Just w')) sid mem }
```

A conversion function `conv` is supplied, which allows writing a value to a share of a type different than type read from it. The shared data source is also supplied with an inital value. This value will be read when no value has been written yet. An example of an internal shared data source is given below. In this example, a share is defined to which an age can be written and the corresponding age category can be read (if the written age was valid):

```
type Age = Int

data AgeCategory = Children
                 | Youth
                 | Adults
                 | Seniors

ageCategory :: Age -> Maybe AgeCategory
ageCategory age
    | age <  0  = Nothing
    | age <= 14 = Just Children
    | age <= 24 = Just Youth
    | age <= 64 = Just Adults
    | otherwise = Just Seniors

ageShare :: ReadWriteShared (Maybe AgeCategory) Age
ageShare = sharedStore 42 Nothing ageCategory
```

Two essential operations to perform on a shared data source are reading and writing. This is done using the `Task` functions `get` and `set` respectively.

```
get :: ReadWriteShared r w -> Task r
get rws = uncurry stable . read rws

set :: w -> ReadWriteShared r w -> Task w
set v rws = stable v . write rws v
```

A function that parametrizes a set of `Tasks` with a shared resource and performs them in parallel using `anyTask` can now easily be defined:

```
anyTaskWithShared :: Typeable t => t -> [Shared t -> Task a] -> Task a
anyTaskWithShared init tasks =
    freshInt >>>
    \sid -> let share = sharedStore sid init id
             in anyTask $ map ($ share) tasks
```

`freshInt` is a `Task` that produces a fresh (i.e. unique) integer. To enable this, the `Environment` is equipped with a counter named `nextI` representing the next fresh integer:

```
data Environment = Environment
    { nextI :: Int
    , -- Other values omitted.
    }
```

```
freshInt :: Task Int
freshInt env = let result = nextI env
                in stable result env { nextI = succ result }
```

Note that share identifiers generated by *µTasks* may overlap with user-defined share identifiers. No mechanism to prevent this has been defined currently, but one solution would be to extend the share ID with a marker that informs whether or not a share is anonymous or not. A function that parametrizes a single `Task` with an anonymous share is simply an instance of `anyTaskWithShared`:

```
withShared :: Typeable t => t -> (Shared t -> Task a) -> Task a
withShared init task = anyTaskWithShared init [task]
```

An often occurring pattern when using shared data sources, is reading a value and then directly writing back an updated result. This pattern is captured in two functions. `preUpdate` yields the value that is written to the given share while `postUpdate` yields the value that was read from the share before the new value was written to it:

```
preUpdate :: ReadWriteShared r w -> (r -> Task w) -> Task w
preUpdate share f = atomic $ get share >>> f >>> flip set share

postUpdate :: ReadWriteShared r w -> (r -> Task w) -> Task r
postUpdate share f = atomic $ get share      >>>
                        \x -> f x            >>>
                              flip set share >>|
                              stable x
```

The `atomic` combinator used here can be ignored for now. This will be discussed in section 3.8. Instead of updating the value using a `Task`, it might sometimes be convenient to use a regular function. This can be achieved by the following two functions:

```
preUpdateF  :: (Typeable r, Typeable w) => ReadWriteShared r w -> (r -> w) -> Task w
preUpdateF share f = preUpdate share (stable . f)

postUpdateF :: (Typeable r, Typeable w) => ReadWriteShared r w -> (r -> w) -> Task r
postUpdateF share f = postUpdate share (stable . f)
```

The following example illustrates how an anonymous shared object can be used to store the number of elapsed seconds, which is just used here by a single `Task` to continuously show a running timer:

```
showTimer :: Task ()
showTimer = withShared 0 runClock
    where
    runClock timer = forever $ postUpdateF timer succ  >>>
                               printClock              >>|
                               sleep 1000
```

```
    printClock seconds = printWith view (h,m,s)
        where
        h =  seconds             `div` 3600  :: Int
        m = (seconds `mod` 3600) `div` 60    :: Int
        s = (seconds `mod` 60)               :: Int

        -- Use the carriage return (\r) to move to the beginning of the line.
        view (h,m,s) = "\r" ++ conv h ++ ":" ++ conv m ++ ":" ++ conv s

        conv n = if n <= 9 then "0" ++ show n
                           else        show n
```

3.8 Atomic tasks

Similar to multithreading, shared resources between `Tasks` introduce synchronization issues. The `preUpdate` and `postUpdate` functions discussed earlier are prone to such errors for example because reading from a share and writing the updated value back happens in multiple steps (because they use the step combinator). This is best shown by an example:

```
-- The shared resource: a simple integer.
sharedCounter :: Shared Int
sharedCounter = sharedStore 0 0 id

-- Task t1: updates the shared resource using an update function
--          that consumes some time.
t1 = preUpdate sharedCounter updater
    where
    updater x = sleep 5000 >>|
                stable (x + 1)

-- Task t2: updates the shared resource after some time, using a
--          fast update function.
t2 = sleep 1000 >>| preUpdate sharedCounter updater
    where
    updater x = stable (x + 10)

-- Task task: performs t1 and t2 in parallel, until both are finished.
task = (t1 -&&- t2)
```

In this example, a counter is shared between two `Tasks` that are run in parallel. Task `t1` increases the counter by one, using an update `Task` which takes a long time to complete. Task `t2` first sleeps for a short time and then updates the counter by 10 using an update function that is very fast. One may expect that the final value of the counter will be 11 once both `Tasks` finish. This is only the case if these update operations are atomic. If this is not the case, the outcome will be one with the presented update functions, as the update done by `t2` is effectively ignored. This phenomenon occurs because the value read by `t1` has become invalid because `t2` updated it before `t1` has the chance to. The same argument holds for writing to a share, but this is already an atomic operation (section 3.7). In this section we will present `atomic`, a combinator for making a `Task` atomic, meaning it will not be interleaved by other `Tasks` running in parallel.

`Task` parallelism can be represented by a tree structure where each node represents a `Task` composed of sequential steps (i.e. applications of the step combinator) and edges between nodes depict children as `parallel` instances. These trees represent the state of a task oriented application at a single point in time. For example, consider the following `Task` definitions:

```
A = T01 >>| (T02 -||- T03) >>| T04 >>| (B -&&- C)

B = T05 >>| T06 >>| (T07 -|| T08)

C = T09 >>| (D -&&- E)

D = T10 >>| (T11 -||- T12)

E = T13
```

Figure 3.1 depicts different tree representations for the given code fragment, which are all valid, depending on the state of the program. While all of these trees are binary, a node can have an arbitrary amount of children as the framework can run an arbitrary amount of `Tasks` in parallel (section 3.6).

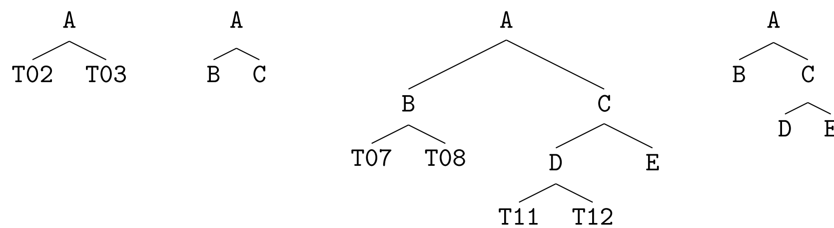

Figure 3.1: Possible tree representations.

In order to achieve atomicity, the `parallel` combinator will have to know its position in this tree. A common way of doing this, is by assigning each node a sequence of integers. This sequence is composed of the sequence of a node's parent and a node's own integral identifier. This implies that (direct) children of a node are never assigned the same identifier. Figure 3.2 shows this hierarchy annotation for one of the trees presented in figure 3.1.

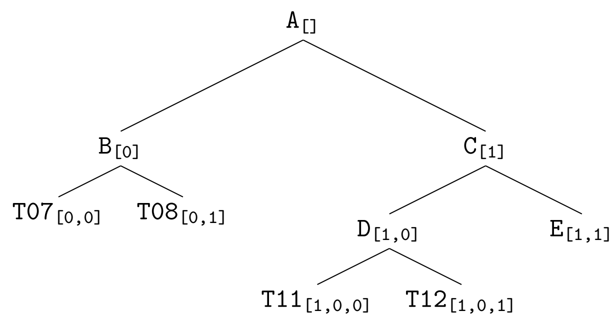

Figure 3.2: Hierarchy annotations for one of the trees presented in figure 3.1.

The next step will be to include and maintain this hierarchy information in the `Environment`, so

that the `parallel` combinator can obtain it. To do this, we first add a member named `parallelId` to the `Environment`:

```
type ParallelId = Seq Int

data Environment = Environment
    { parallelId :: ParallelId
    , -- Other fields omitted.
    }
```

`parallelId` is of type `Seq Int` instead of `[Int]` one might expect. This is merely for efficiency reasons: the `Seq` (sequence) data structure is based on finger trees, which allow for efficient addition to the back and front[2]. `parallelId` is managed by the `parallel` combinator. Recall that `Tasks` are already assigned a locally unique identifier to provide an ordering property (section 3.6.2). Conveniently, this identifier is also usable here. A `parallelId` can either be extended, or shortened:

```
-- Add the Task identifier "i" to the parallelId contained in "env".
extendParallelId i env = env { parallelId = parallelId env |> i }

-- Shorten the parallelId contained in "env" by 1 level.
shortenParallelId env = let xs :> _ = viewr $ parallelId env
                         in env { parallelId = xs }
```

All that remains to manage the parallel hierarchy information, is to call these function at the appropriate location(s). While `parallel` is a relatively big combinator, there is only one point of `Task` evaluation (i.e. the environment being applied to a `Task`): directly after determining that the `tasks` parameter is not empty:

```
let (i,x)       = elemAt   0 tasks
    xs          = deleteAt 0 tasks
    (tr, env’) = x env
    in ...
```

By enclosing this with the aforementioned functions, `parallelId` will be updated correctly:

```
let (i,x)       = elemAt   0 tasks
    xs          = deleteAt 0 tasks
    (tr, env’) = x (extendParallelId i env)
    env’’       = shortenParallelId env’
 in ... -- The next operation involving the environment now uses env’’ as opposed to env’.
```

Now that it is possible to query the `Environment` for the position in the parallel hierarchy tree, it is time to return to the initial problem: `Task` atomicity. When a `Task` is marked as atomic,

[2] http://hackage.haskell.org/package/containers-0.5.7.1/docs/Data-Sequence.html

it means only its parent nodes in the parallel hierarchy tree may be evaluated. Storing just the parallel ID of that `Task` allows the `parallel` combinator to determine whether or not it is allowed to rewrite other `Tasks`. There is one thing to take into account however: sub-`Tasks` of a `Task` may be marked as atomic as well. This may result in overwriting the parallel ID without a way to recover it. For this reason, the `atomic` function recovers the ID of the parent `Task`:

```
data Environment = Environment
        { parallelId  :: ParallelId
        , atomicLevel :: ParallelId
        , -- Other fields omitted.
        }

atomic :: Task a -> Task a
atomic task = envRead atomicLevel      >>>
      \old -> envRead parallelId       >>>
      \pid -> envWrite (setAtomic pid) >>|
              task                     >>>
      \res -> envWrite (setAtomic old) >>|
              stable res
    where
    setAtomic pid env = env { atomicLevel = pid }
```

One thing to take into account here, is that the atomic `Task` can throw an exception. If not caught within the `Task` itself, this will circumvent recovery of the atomic level. For this reason, the `throw` function is modified so it clears atomicity (i.e. sets the atomic level to the root node of the hierarchy tree). All that remains now is modification of the `parallel` combinator. It will be required to determine whether or not we are a parent of the atomic `Task` or not. A helper function named `isParent` tells us whether or not one `ParallelId` is the parent of another:

```
-- e.g isParent []    [0]     = True
--     isParent [0,0] [0,0]   = False
--     isParent [0]   [0,1,1] = True
--     isParent [0,1] [0,0,1] = False

isParent :: ParallelId -> ParallelId -> Bool
isParent parent child = maybe False (not . null) (stripPrefixSeq parent child)

-- e.g. stripPrefixSeq [0]   [0,1,2] == Just [1,2]
--      stripPrefixSeq [1,2] [0,1,2] == Nothing

stripPrefixSeq :: Eq a => Seq a -> Seq a -> Maybe (Seq a)
stripPrefixSeq (viewl -> EmptyL) ys = Just ys
stripPrefixSeq (viewl -> (x :< xs)) (viewl -> (y :< ys))
    | x == y = stripPrefixSeq xs ys
stripPrefixSeq _ _ = Nothing
```

The following part of the `parallel` combinator can now be modified:

```
-- Case 3: Evaluation did not result in a stable value.
--         Add the follow-up task to the accumulator.
_  -> let followups' = insert i (followup tr x) followups
       in parallel' stables followups' xs predOK someDone allDone env'
```

By checking if the combinator is a parent of a `Task` that is performing an atomic action, atomicity is now achieved:

```
-- Case 3: Evaluation did not result in a stable value (noValue or unstable).
_  -> if isParent (parallelId env'') (atomicLevel env'')
            then let xs' = insert i (followup tr x) xs
                  in parallel' stables followups  xs' predOK someDone allDone env''

            else let followups' = insert i (followup tr x) followups
                  in parallel' stables followups' xs  predOK someDone allDone env''
```

3.9 I/O-related actions

I/O-related actions with regards to the world (e.g. reading from a file on the file system) require special attention because they are enabled by special mechanisms and/or properties of the typing system that prevent them from doing "impure" things. *Clean* for example uses uniqueness typing [Pla01] and *Haskell* uses the `IO` monad for this purpose. This means that if the framework is to perform I/O, it must abide by the rules imposed by these mechanisms. In our case, it means we must embrace the `IO` monad. Recall our `Task` definition presented in section 3.1:

```
type Task a = Environment -> (TaskResult a, Environment)
```

By changing this definition to the following, `Tasks` can now perform I/O-related actions:

```
type Task a = Environment -> IO (TaskResult a, Environment)
```

All types and definitions handled so far did not contain the `IO` monad to keep things simple. The modifications required are minimal: they mostly consist of adding `return` calls and `do` blocks at the appropriate places. Some of the functions shown in section 3.2 actually require the `IO` monad. For example, the basic `Task` function `hReadLine` is defined as follows:

```
hSetBuffering :: Handle -> BufferMode -> IO ()
hReady        :: Handle -> IO Bool
hGetLine      :: Handle -> IO String
```

```
hReadLine :: Handle -> Task String
hReadLine handle = eval
    where
    eval e = do hSetBuffering handle LineBuffering
                isReady <- hReady handle
                if isReady
                    then do line <- hGetLine handle
                            stable line e
                    else noValue (hReadLine handle) e
```

Sometimes, a direct translaton is required from the `IO` domain to the `Task` domain. The `liftIO` function has been added for this reason.

```
liftIO :: IO a -> Task a
liftIO action = \env -> action >>= \v -> stable v env
```

Note that the `(>>=)` function, is the “bind” function and not a `Task` combinator. The basic `Task` `currentTime` is an example of an application of `liftIO`:

```
currentTime :: Task UTCTime
currentTime = liftIO getCurrentTime
```

3.10 Miscellaneous components

3.10.1 Tasks as a sequence of steps

Often, a certain (equivalently typed) group of `Tasks` represent a sequence of steps. This sequence can be interrupted if one of the steps result in an erroneous value, in which case the result of the sequence as a whole equals this value. In order to more conventiently express such a sequence, the `Steppable` typeclass has been added to the framework:

```
class Eq a => Steppable a where
    stopStep :: a -> Bool

    step :: Task a -> Task a -> Task a
    step ta tb = ta >>> \x -> if stopStep x then stable x
                                             else tb

instance         Steppable Bool      where stopStep = id
instance Eq a => Steppable (Maybe a) where stopStep = isNothing
instance Eq a => Steppable [a]       where stopStep = null
```

A steppable type is expected to define a `stopStep` function. Once this is defined, `Tasks` resulting in this type can be specified as a sequence of steps using the `step` function (not to be confused with the step combinator `(>>*)`). The example below forms a sequence of steps from several checks:

```
data RequestData = RequestData { username :: String
                               , password :: String
                               , action   :: Action
                               }

data Action = AddUser | DeleteUser | ViewUser

validateUsername :: RequestData -> Task Bool
validatePassword :: RequestData -> Task Bool
isAuthorized     :: RequestData -> Task Bool

-- A request is valid, if it contains valid credentials,
-- and the user is authorized for requested action.
validRequest :: RequestData -> Task Bool
validRequest info = validateUsername info `step`
                    validatePassword info `step`
                    isAuthorized     info
```

3.10.2 Task equivalents of existing functions

One often occurring problem is requiring a function that is not available in the `Task` domain. For that purpose, the `lift` and `taskApply` functions have been defined. The `taskApply` function is a variant of the `(?)` combinator discussed in section 3.4.3.

```
taskApply :: (a -> b) -> Task a -> Task b
taskApply f ta = ta >>> stable . f

taskApply2 :: (a -> b -> c) -> Task a -> Task b -> Task c
taskApply2 f ta tb = ta >>> \x -> taskApply (f x) tb

taskApply3 :: (a -> b -> c -> d) -> Task a -> Task b -> Task c -> Task d
taskApply3 f ta tb tc = ta >>> \x -> taskApply2 (f x) tb tc

lift :: (a -> b -> c -> Task d) -> Task c -> a -> b -> Task d
lift f v x y = v >>> f x y
```

Some instances of these functions have been included in the framework because they often occur:

```
taskMaybe :: Task (Maybe a) -> Task d -> (a -> Task d) -> Task d
taskMaybe = lift maybe

taskIf :: Task Bool -> Task d -> Task d -> Task d
taskIf = lift thenElseIf

thenElseIf :: t -> t -> Bool -> t
thenElseIf x y cond = if cond then x else y

taskAnd :: Task Bool -> Task Bool -> Task Bool
taskAnd = taskApply (&&)
```

```
(+++) :: Task [a] -> Task [a] -> Task [a]
(+++) = taskApply (++)
```

For optional and boolean values, it often happens that additional actions are to be performed for just one of its options, propagating its result for the other outcome. These patterns have been captured by the following functions:

```
whenJust :: Task (Maybe a) -> (a -> Task (Maybe a)) -> Task (Maybe a)
whenJust prod just = taskMaybe prod (stable Nothing) just

whenNothing :: Task (Maybe a) -> Task (Maybe a) -> Task (Maybe a)
whenNothing prod nothing = taskMaybe prod nothing (stable . Just)

whenFalse :: Task Bool -> Task Bool -> Task Bool
whenFalse prod false = taskIf prod (stable True) false

whenTrue :: Task Bool -> Task Bool -> Task Bool
whenTrue prod true = taskIf prod true (stable False)
```

Another pattern that often occurs, is that a `Task` possibly produces a value (i.e. a `Maybe` type), and we want to know if a certain predicate holds for this value. No value being produced should be considered the same as the predicate not holding. The `taskMaybePred` function does just this:

```
taskMaybePred :: Task (Maybe a) -> (a -> Bool) -> Task Bool
taskMaybePred m pred = m >>> stable . (==) (Just True) . fmap pred
```

Instead of obtaining the boolean result, we may also want to branch on it directly. The `maybeIf` function provides the result of `taskMaybePred` directly to `taskIf`:

```
maybeIf :: Task (Maybe a) -> (a -> Bool) -> Task b -> Task b -> Task b
maybeIf maybe pred = taskIf (taskMaybePred maybe pred)
```

3.10.3 Collections of tasks

While similar to the previous section in the sense that the functions discussed here are `Task` equivalents of existing functions, they all concern collections:

```
taskConcat :: [Task [a]] -> Task [a]
taskConcat = foldr (+++) (stable [])

taskConcatMap :: (a -> Task [b]) -> [a] -> Task [b]
taskConcatMap f = taskConcat . map f

foldrMap :: (a -> a -> a) -> (b -> Task a) -> [b] -> Task a
foldrMap ffold fmap = foldr1 (taskApply ffold) . map fmap
```

```
taskSeq :: [Task a] -> Task [a]
taskSeq []     = stable []
taskSeq (x:xs) =   x  >>>
    \v  -> taskSeq xs >>>
    \vs -> stable (v:vs)

taskSeq_ :: Foldable t => t (Task a) -> Task ()
taskSeq_ = foldr (>>|) (stable ())
```

Chapter 4

Case study: a task-oriented payment terminal implementation

In order to develop a better understanding of the problem domain described in chapter 1, a case study was performed. This chapter contains the results of this case study. In this case study, a task-oriented implementation was developed for a payment terminal transaction using the μ*Tasks* framework presented in chapter 3. The first step of this case study involved analyzing what a payment terminal transaction entails. This analysis is presented in section 4.1. Based on this analysis, several core characteristics of the process have been identified. From these characteristics, several task-oriented candidate approaches have been developed that aim to facilitate the development of systems possessing the aforementioned characteristics. One of these candidate approaches was selected as best suited for this case study. These characteristics and the resulting task-oriented approaches are discussed in section 4.2. Lastly, the approach that was determined most suitable was used for developing a proof of concept which shows that Task-Oriented Programming can be an elegant alternative for systems like payment terminals. This proof of concept and a qualitative analysis of it is presented in section 4.3.

4.1 Background: payment terminals and transactions

A payment terminal facilitates financial exchanges between a customer (i.e. the cardholder) and a merchant. During a transaction, the terminal communicates with an *acquirer*. An acquirer is a bank or financial institution that processes credit or debit card payments on behalf of the merchant for specific brands (e.g. Visa, Mastercard and Maestro).

A payment terminal is initially supplied with, among other things, data on card identifiers, brands, and acquirer information. A merchant can then configure the terminal to his/her own preference (e.g. assigning specific acquirers to specific card brands). This is required to enable the payment terminal to perform card recognition and acquirer selection. Multiple aspects of a transaction are variable. These include, but are not limited to, the following:

1. A payment terminal itself can either be attended (i.e. under direct merchant control), unattended (i.e. a self-service environment) or operating by mail/telephone order.
2. Multiple card entry modes exist, each having its own rules and standards: EMV smartcards, magnetic stripe, contactless cards and manual card number entry.

3. The terminal operation mode. When in “On-line authorization” mode, the terminal requests authorization from the acquiring host during a transaction. In “semi-online mode”, the terminal starts locally processing a transaction based on information provided by the acquiring host at initialization, and on-line authorization is only performed if needed. Finally, there is also “semi-offline mode” (or “offline mode”), which is similar to “semi-online mode”. The details are not relevant to this thesis.

4. There are multiple transaction types (also named *services*). “Payment”, “refund” and “cancellation” are all considered transactions for example.

The *Common Terminal Acquirer Protocol* (C-TAP) specification [Acq12a, Acq12b, Acq12c] abstracts over these things: a terminal supporting C-TAP can communicate with an acquirer supporting C-TAP, regardless of details and specifics. Figure 4.1 depicts the scope of the C-TAP specification.

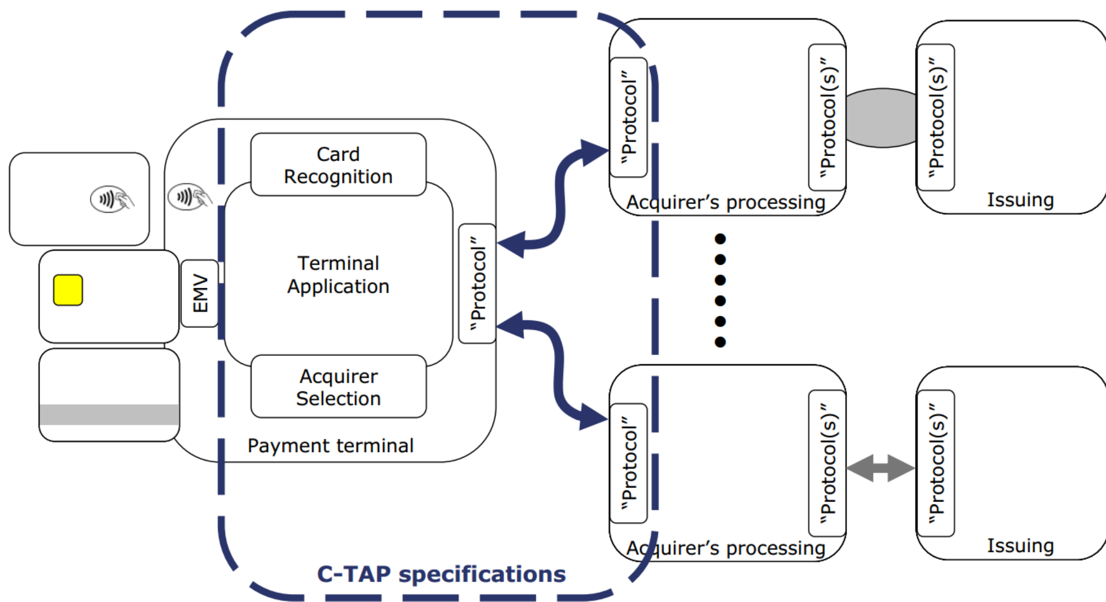

Figure 4.1: Scope of the C-TAP specification [Acq12a].

Note that the actual card communication is out of scope of C-TAP. This is defined in the corresponding standard for the card entry method (e.g. the EMV specification for an EMV smart card [EMV11a, EMV11b, EMV11c, EMV11d]). The process named *card transaction* is described by C-TAP as follows:

> *“The aim of the “Card Transaction“ is first to request the acquirer to authorize, and next to notify the acquirer to record, a given card-based transaction for the card that the cardholder presents on the terminal of the card acceptor, this transaction relating to a particular service.” [Acq12b]*

The *card transaction* is realised in two steps thanks to a double exchange protocol. First, the terminal sends an *authorization request*, to which the acquirer replies with an *authorization response*. This first message exchange is called the *authorization exchange* where the acquirer gives to the terminal an approval or refusal for the requested service.

Next, the terminal informs the acquirer of the transaction completion (i.e. the final result) by sending a *completion advice*, to which the acquirer responds with a *completion acknowledgement*. This second message exchange is called the *completion exchange* which finalizes the transaction. This double exchange protocol between the terminal and the acquirer is depicted by figure 4.2.

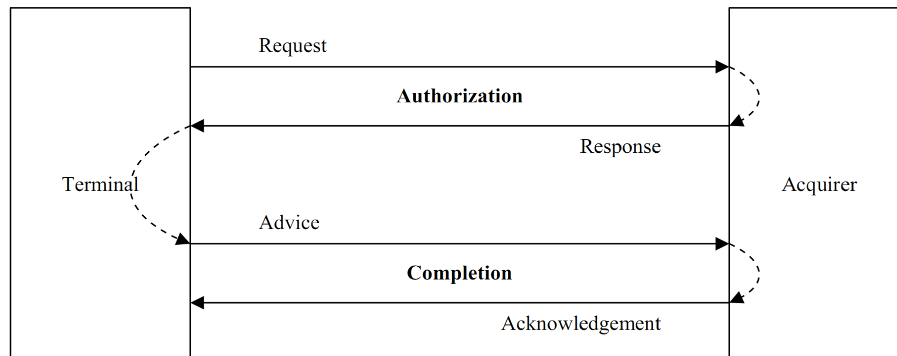

Figure 4.2: “card transaction” message flow between a terminal and an acquirer [Acq12b].

The messages sent in these two exchanges are all structured equally (i.e. the C-TAP “transaction message” structure). However, some fields are omitted in one message, but present in another (or in some cases conditionally present). C-TAP categorizes the activities performed by the terminal by *actions*. Figure 4.3 shows these actions and the transitions between them. Note that the authorization and completion exchange shown in figure 4.2 are included as actions in this model. A brief description of each action is provided next. These descriptions are not complete and merely try to convey the general idea.

Preliminary actions: The goal of the preliminary actions is to identify the presented card (i.e. the card brand) and to obtain the required parameters (i.e. an acquirer) to accept the card and to process the transaction accordingly. This includes activities like cardholder verification which, for services requiring a card, ensures that the person presenting the card is the person whom the card was issued to.

Authorization exchange: The is the first message exchange with the acquirer. The terminal has built up an *authorization request* consisting of all data required by the acquirer to make a decision on whether or not to authorize the transaction. Aside from an “approval” or “rejection”, the acquirer may also respond with an “alterable refusal”. In this case the terminal is to perform certain resubmission actions before re-entering the authorization exchange.

Resubmission actions: This action is performed when the acquirer responds with an “alterable refusal” in the request exchange, indicating that more actions are required for (potential) approval. Examples of such actions are re-entering a PIN in case the first one was incorrect and providing additional authorization data.

Completion actions: The completion actions are performed when the terminal receives an “approval” response at the authorization exchange. At this stage the transaction was approved, meaning the goods or service(s) may

be delivered or rendered, if not done already. The acquirer may still request additional data even though it approved the transaction.

Completion exchange: The completion exchange is the last message exchange of the card transaction process. This message exchange intends to notify the acquirer that the transaction was either completed or cancelled at the point of service, and to inform of the result of the transaction that it closes. This exchange finalizes and closes the transaction. This action may have to be performed multiple times in case of failure (e.g. the terminal may not be able to connect to the acquirer)

Cancellation actions: The cancellation actions are triggered by the terminal on any of the cancellation conditions that may occur in other phases of the card transaction process. The goal of this phase is to store the required information for the later (negative) *completion advise*.

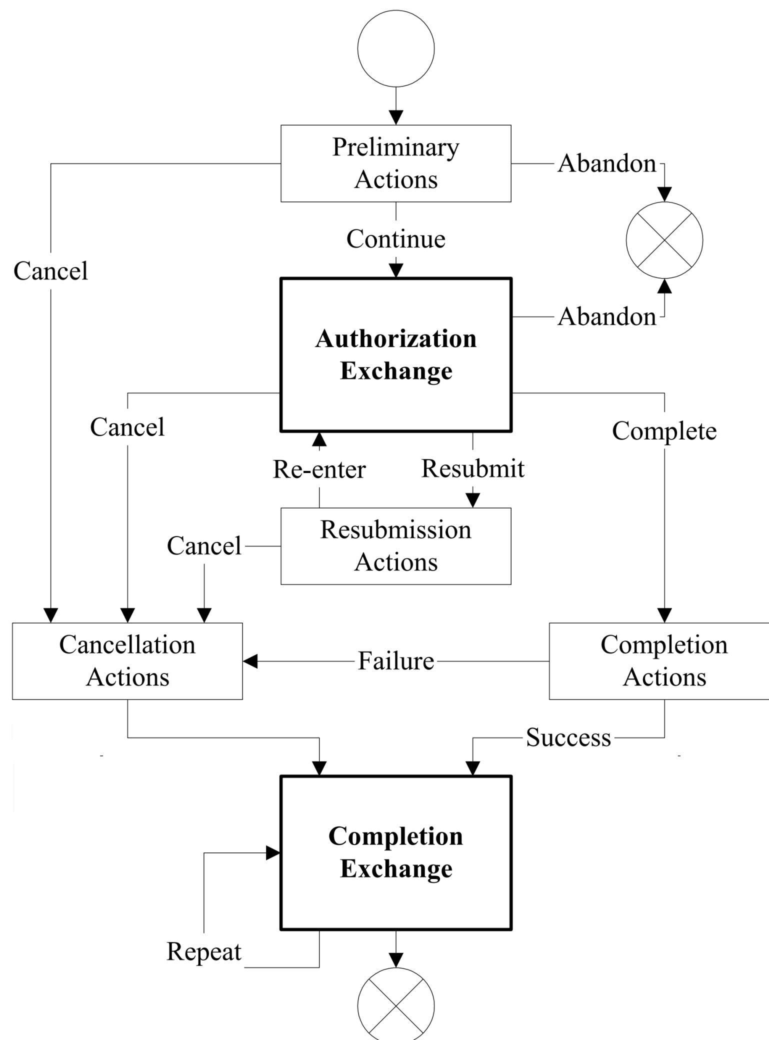

Figure 4.3: Online card transaction from a terminal perspective [Acq12b].

One thing of importance is that from a certain point in the transaction, the terminal is expected to finalize it by sending a (possibly negative) *completion advise.* More specifically, this "point of no return" is reached when the terminal sends its *authorization request.* The acquirer has no knowledge of the transaction prior to this, meaning the terminal may locally abandon it up until this point. In figure 4.3 this is reflected by an "abandon" transition for local abandonment and a "cancel" transition for on-line finalization using a *completion advise.* Some acquirers require a terminal to always send a completion advise, hence the "cancel" transition from "preliminary actions".

After having accepted (and therefore identified) a card, the terminal initializes and maintains registers that reflect the transaction's progress and results. For transactions performed with an EMV smartcard, the terminal also obtains the application profile from the card indicating its capabilities. Card recognition and acceptance are the first steps of the "preliminary actions" shown in figure 4.3. An overview of the aforementioned registers is presented next.

Application Interchange Profile (AIP): Reflects the capabilities and preferences of the Integrated Circuit Chip (ICC) contained on the card. Examples of bits that can be set in this register are "cardholder verification is supported" and "terminal risk management is to be performed".

Terminal Verification Results (TVR): Contains information on established results. Examples of bits that can be set in this register are "ICC data missing", "PIN try limit exceeded" and "requested service not allowed for card product".

Terminal Status Information (TSI): Gives an indication of the transaction progress. Examples of bits that can be set in this register are "cardholder verification was performed" and "offline data authentication was performed".

As mentioned earlier, communication with the card itself is not defined by C-TAP, but by the specification of the corresponding card type. For EMV smartcards for example, the EMV specification prescribes how the terminal can interact with the card (or more specifically, the ICC contained on it). Figure 4.4 depicts an example flow of a transaction from an EMV smartcard point of view, taken from the EMV specification.

C-TAP refers to these specifications where needed; C-TAP is complementary as opposed to conflicting. For example, the EMV activities up to the "online / offline decision" correspond to the "preliminary actions" for EMV-based transactions in C-TAP. Likewise, "online processing & issuer authentication" as specified in the EMV specification corresponds to the "authorization exchange" in C-TAP. Of course, the EMV specification only takes into account activities involving the card. Terminal-only actions are only defined in the C-TAP specification.

An initial (negative) completion advise is safely stored in the terminal's non-volatile memory just before sending out the *completion request.* This allows the terminal to repeat its advise even after a power loss. This is essential, because a terminal can not start a new transaction (with the same acquirer) before a pending transaction has been finalized.

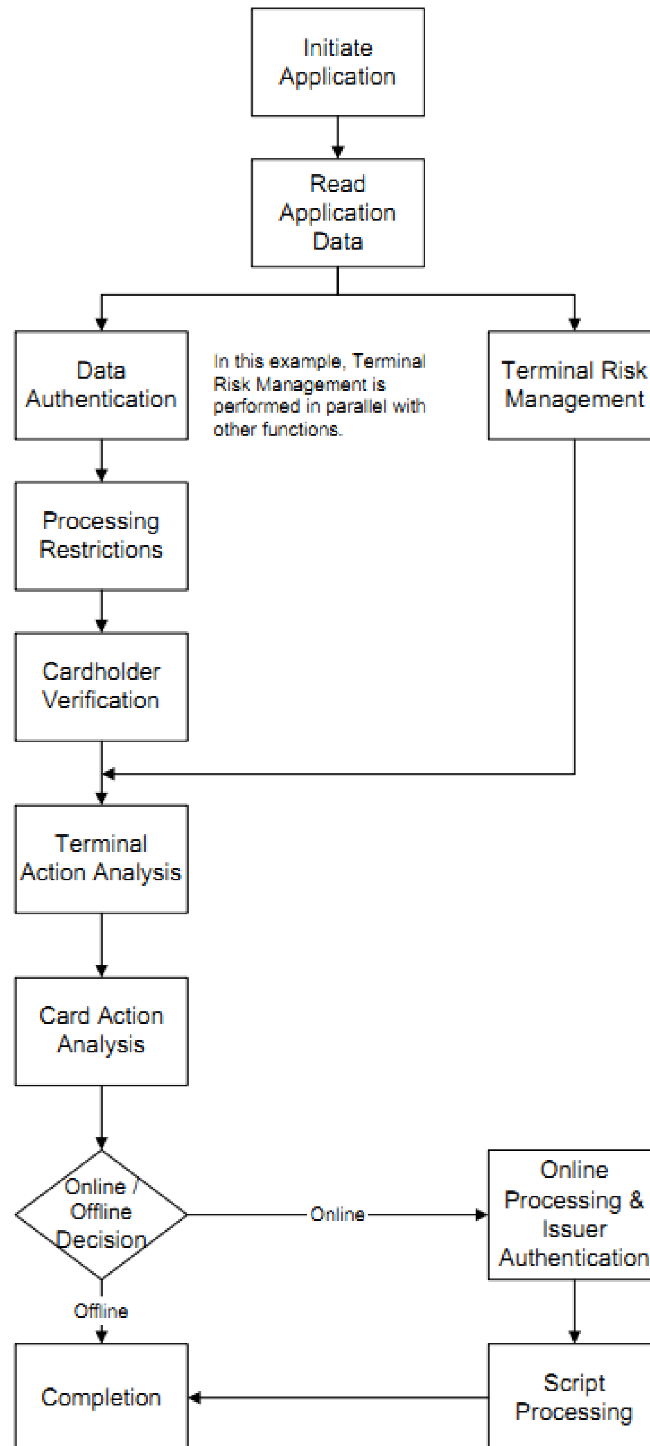

Figure 4.4: Example transaction flow from an EMV smartcard point of view [EMV11c]

.

4.1.1 Transaction simplifications

For the proof of concept code accompanying this thesis, simplifications were applied to the transaction process. The goal of applying these simplifications was not to enable a solution that would otherwise not have been possible, but to focus on the essential characteristics. It is after all a proof of concept, aiming to give credibility to the possibility of a task-oriented solution. The following simplifications have been made:

1. The scope is limited to a transaction only. This means it is assumed that the terminal is initialized accordingly and contains all parameters required to be C-TAP operational.
2. There is no interaction with actual hardware. Hardware will be simulated by stubs. Likewise, no actual networking driver will be addressed for TCP/IP communication.
3. Transaction message (the structure followed by the messages in both message exchanges) content has been minimized, and types have been simplified for cases where this does not result in a change of design.
4. A request is always responded to by a response. Additionally, this response is never malformed or incomplete. This applies to both responses obtained from the ICC and the acquirer.
5. The terminal and the ICC are under specific conditions allowed to perform certain tasks in parallel in order to be faster. “Terminal risk management” for example is performed in parallel in figure 4.4. In the proof of concept, such speed optimizations are not taken into account even though performing tasks in parallel is easy with the dedicated combinators contained in μ*Tasks* (section 3.6).

4.2 System characteristics and candidate solutions

A task specification for the payment terminal must abide by the C-TAP and EMV specifications discussed in section 4.1. The actions defined by C-TAP depicted in figure 4.3 are typical candidates to be modelled as tasks. The following system characteristics are identified from analyzing the aforementioned specifications:

common
: There is a common set of data that multiple tasks require access to. Examples of elements contained in this set are the TVR, TSI and AIP registers.

dynamic
: This set of data is dynamic: it may grow as the transaction task progresses. For example, prior to the authorization exchange (section 4.1) this set does not include an authorization response but after, it does.

recoverable
: The set of data mentioned in common must be recoverable when the task is interrupted. Removal of the card will interrupt a transaction for example. Subsequent actions taken depend on the progress of the interrupted process, which is reflected by the aforementioned set of data.

The challenge of this case study is to come up with a task-oriented approach that respects these characteristics while still maintaining the elegance of Task-Oriented Programming (chapter 2). In the following subsections, a couple of candidate solutions are presented. Because these characteristics are not specific to the payment terminal domain, the presented approaches will not be specific to that domain either. The candidate solution discussed in section 4.2.2 is determined best-suited for this case study, and is used in section 4.3 for the development of the proof of concept. When discussing the candidate solutions, the characteristics that were just presented will be referred to by their label (i.e. common, dynamic or recoverable).

4.2.1 Candidate solution: implicit data relay by use of state

One approach is to encapsulate the data in (a) state(s) hidden by combinators (similar to how the `Environment` is hidden when defining `Tasks`). This results in a new task type `Task2` having the following form where `S` represents the state:

```
type Task2 a = S -> Task (a, S)
```

For recoverable, `S` should contain all elements that are to be recovered in case of interruption. This is likely to result in optional values because not all values are obtained at the same time, as stated in dynamic. Without recoverable, optional values could be avoided by making multiple task types and state size could be kept minimal by defining a hierarchy of task derivations. For example, a task type `Task3` based on the shown `Task2` type would have the following form:

```
type Task3 a = S2 -> Task2 (a, S2)
```

To make `S` accessible from a task at the highest level (that is still interruptible), which is required for recoverable, this task itself needs to be of type `Task2`. For this reason, the type definition for `Task2` should be modified into the following, as none of the state members will be present initially (if they are they should not be part of the state). Alternatively, all state members are made optional.

```
type Task2 a = Maybe S -> Task (a, Maybe S)
```

The main issue with this approach is that existing combinators can no longer be used. After all, these are defined for values of type `Task a` and not `Task2 a`. Framework modifications resulting in task combinators being applicable to arbitrary (possibly nested) task derivations are possible, but not trivial. In this case study, there is only one additional state on top of the `Environment`. For this reason, the idea of hierarchical tasks is discarded.

Defining a hierarchy of state monads is more of a functional programming approach than a task-oriented one. The candidate solution discussed in section 4.2.2 is a proper task-oriented solution. This candidate is better suited for this case study and is used for the proof of concept presented in section 4.3.

4.2.2 Candidate solution: state as a shared resource

Instead of deriving a new task type as proposed in section 4.2.1, the `Environment` can be used for state storage, as every `Task` has access to this `Environment`. The most straightforward approach would be to store data in one or multiple shared data sources (section 3.7) having pre-arranged share IDs. The downside of this approach is that one needs to explicitly state which share to access. In this section we will present a class-based alternative which allows share identification by type. This means a share can be obtained without having to explicitly mention the share ID because the type system can automatically deduce the type (section 2.3.2). Using a class definition has the consequence that there can only be one share per type. For this case study this is acceptable, because there is just one state. An older type can always be wrapped in an algebraic data type to form a new type. This way multiple shares can be defined for the "same" type.

The `RecoverableState` class represents the aforementioned class. Once the `shared` function is defined, the other functions of the class can be used to manipulate the state similar to how the `Environment` is manipulated.

```
class Typeable s => RecoverableState s where

    -- We write the state, and read the state (if it's there).
    shared :: ReadWriteShared (Maybe s) s

    -- Read and write from / to a state.
    rsRead :: (s -> a) -> Task a
    rsRead f = taskMaybe rsGet
        (throw RecoverableStateException)
        (stable . f)

    rsWrite :: (s -> s) -> Task ()
    rsWrite f = preUpdate shared upd ? const ()
        where
        upd = maybe (throw RecoverableStateException) (stable . f)

    -- Get and set the state.
    rsSet :: Task s -> Task ()
    rsSet ta = ta >>> flip set shared ? const ()

    rsGet :: Task (Maybe s)
    rsGet = get shared
```

Note that the type read from the share is `Maybe s`. This is done for the same reason as `S` was changed to `Maybe S` for in the `Task2` type presented in section 4.2.1. As shown, a share ID needs to be defined explicitly. One could likely generically (section 2.3.5) derive a share identifier from a type (for example by exploiting the fact that a type name is unique), in which case there would not longer be a need to specify a share ID. Alternatively, the type of a share ID could be changed, for example to a string so that the name of the (non-polymorphic) type can be used. This route has not been pursued any further in this case study.

With `RecoverableState`, a function to solve the problem of state recovery can be defined. The idea of this function is that a state builder `Task` is run in parallel with an interruption `Task` that can interrupt it. This interruption occurs when the interruption functions terminates with a value that satisfies a given predicate. If this predicate does not hold, or the state building function terminates first, the function simply results in the result of the state builder.

```
withStateRecovery :: RecoverableState s  =>
                     (b -> Bool)         ->
                     (Maybe s -> Task a) ->
                     Task a              ->
                     Task b              ->
                     Task a
withStateRecovery pred recoveryFunction stateBuilder interruptionFunction =
  parallel [stateBuilder ? Left, interruptionFunction ? Right] (not . null) id undefined >>> res
  where
  res ([Right x], [stateBuilder']) = if pred x then rsGet >>> recoveryFunction
                                               else stateBuilder' ? fromLeft
  res ([Left  x], _)               = stable x
```

Detection of card removal happens using a heartbeat function. Metaphorically, the heart keeps on beating while the card is present. In other words: termination of this function indicates that the card is no longer present. This means any observed stable result will result in execution of the interruption function. State recovery using a heartbeat function is simply a less general version of `withStateRecovery`:

```
withStateRecoveryUsingHeartbeat :: RecoverableState s =>
                                   (Maybe s -> Task a) -> Task a -> Task b -> Task a
withStateRecoveryUsingHeartbeat = withStateRecovery (const True)
```

The following example shows how the discussed mechanism are applied. Here, `State` contains three elements that are to be recovered, of which one (`z`) is not present initially. The task `task` assigns the members of `State` as it progresses.

```
data State = State { x :: Int
                   , y :: Int
                   , z :: Maybe Int
                   }

instance RecoverableState State where
    shared = sharedStore 42 Nothing Just

interruptibleTask = withStateRecoveryUsingHeartbeat handler
                                                    task
                                                    interrupt
    where
    task = first >>| second >>| third

    handler :: Maybe State -> Task ()
    handler s = -- 'task' was interrupted. All needed information is present in 's'.
```

Because `x`, `y` and `z` are contained in the `Environment`, these do not appear in the definition of `task`. In this example, `first` could set the initial state using the `rsSet` function, e.g.:

```
rsSet $ stable State { x = 41
                     , y = 42
                     , z = Nothing
                     }
```

`second` can now access `x` and `y` and ultimately set `z`. Similarly, `third` has access to `x`, `y` and `z`.

```
second =   rsRead x >>>
    \x' -> rsRead y >>>
    \y' -> ...      >>|
           rsWrite $ set z (Just 43)
```

4.3 Proof of concept

With a task-oriented solution available that specifically targets systems with characteristics of a payment terminal (section 4.2.2), the implementation of one becomes relatively straightforward as we will see in this section. The transaction process itself is sequential, as shown by figure 4.3. The presented task-oriented solution allows this sequence to be directly translated to code, because the actions of which it is composed need not be aware that they can be interrupted (since this is accounted for by the state recovery function):

```
transaction amount = preliminaryActions amount >>> evalAction
    where
    evalAction ApproveTransaction         = -- Out of scope.
    evalAction DeclineTransaction         = -- Abandon locally, no cancellation actions.
    evalAction AuthorizeTransactionOnline = authorizeOnline

    authorizeOnline = authorizationExchange >>>
        \exchangeResult -> case exchangeResult of
            Approval                 -> taskIf completionActions
                                              completionExchange
                                              cancelTransaction

            AlterableRefusal actions -> taskIf (resubmission actions)
                                              authorizeOnline
                                              cancelTransaction

            DefinitiveRefusal        -> cancelTransaction
        where
        cancelTransaction = cancellationActions >>| completionExchange
```

Note that the result of the preliminary actions is a decision on whether to approve the transaction, decline the transaction or authorize it online. The constructor names have been chosen to reflect this. This is why they are named differently than the transition labels shown in figure 4.3. The state containing data that must be recoverable looks as follows. This state enables a terminal to determine its transaction progress and status through fields like the TVR and TSI registers, and even to resume it as it contains the request message:

```
data TerminalState = TerminalState
    { _card              :: CardBrandIdentifier
    , _acquirer          :: AcquirerIdentifier
    , _aip               :: ApplicationInterchangeProfile
    , _tsi               :: TransactionStatusInformation
    , _tvr               :: TerminalVerificationResults
    , _request           :: TransactionMessage
    , _response          :: Maybe TransactionMessage
    , _applicationData   :: ApplicationData
    , _acquirerConnected :: Bool
    }
```

This shows another reason why implementation of the terminal becomes straightforward using the task-oriented approach: all data that needs to be recovered in case of interruption that is required by a task is available in the `Environment`. Through the `RecoverableState` class this data is easily

accessible. This allows implementing the various steps of the transaction process independently. After assigning a share for this state by defining the `shared` function of the `RecoverableState` class, the transaction task can be interrupted by removal of the card because the interruption handler is provided with all information necessary to properly act on this event:

```
transaction :: Int -> Task ()
transaction amount = withStateRecoveryUsingHeartbeat fallbackProcedure
                                                     (tryTransaction amount)
                                                     cardReaderHeartbeat
    where
    fallbackProcedure :: Maybe TerminalState -> Task ()
    fallbackProcedure Nothing  = -- No initial state was made yet.
    fallbackProcedure (Just s) = -- Determine the action based on 's'.
```

In order to more easily access members in the state, helper functions were defined. For example, the following functions facilitate access to the TVR register:

```
getTVR field       = rsRead  $ view (tvr . field)
setTVR field value = rsWrite $ (over tvr . set field) value
```

All that remains to do now is to:

1. Implementing the various actions defined by C-TAP (figure 4.3), refining them into the actions specified by the EMV specification (figure 4.4) where needed. An example of this is given in section 4.3.1. The remaining modules will not be listed in this paper, but are available[1].
2. Dealing with physical components like the card reader and the safe storage module. Physical components are typically modeled using shared data sources (section 3.7), because their limited availability forces them to be shared by `Tasks`.

4.3.1 Example: cardholder verification

One of the steps of the "preliminary actions" defined by C-TAP is "cardholder verification" which aims to ensure that the person presenting a card is also the person the card was issued to. This section shows an example of how this action could be implemented using the discussed approach. The goal of this example is to show how the discussed approach (section 4.2.2) is applied in practice; the details are not important at this point.

```
1  data PINEntryResult = PINEntrySuccessful EncryptedPIN
2                      | PINEntryTimeout
3                      | PINEntryCancelled
4
5
6
7
```

[1]The complete sourcecode is available upon request. Contact by mail: jasper.piers@gmail.com

```
8   -- C-TAP.200 Section 3.3.1.5, EMV Section 10.5
9   cardholderVerification :: Task ()
10  cardholderVerification =
11      -- Check in the AIP if we support at least one CVM.
12      taskIf (getAIP cardholderVerificationIsSupported)
13          performCardholderVerification
14          (stable ())
15
16  performCardholderVerification :: Task ()
17  performCardholderVerification =
18      taskMaybe (searchAppData Tag_CardholderVerificationMethodList)
19          (stable ()) -- No CVM on the ICC
20          (\CVMList { _amountX
21                    , _amountY
22                    , _cardholderVerificationRules
23                    } -> case _cardholderVerificationRules of
24                           []        -> stable ()
25                           otherwise -> evalRules _amountX _amountY _cardholderVerificationRules >>|
26                                        setTSI cardholderVerificationWasPerformed True)
27
28  -- This function should not be called with an empty ruleset, because that should not
29  -- set the TVR bit (and ofcourse also not the TSI bit)
30  evalRules :: Int -> Int -> [CardholderVerificationRule] -> Task ()
31  evalRules x y [] =
32      setRequest (emvChipTechnologyGroup . cardholderVerificationMethodResults . cvmResult)
33                 (Just False) >>|
34      setTVR cardholderVerificationWasNotSuccessful True
35
36  evalRules x y (cvr@(rule,cond):xs) =
37      -- Check whether or not the rule in the condition is satisfied.
38      taskIf (ruleConditionSatisfied cond)
39          -- Condition satisfied: try to perform the CVM.
40          -- Set this CVM as the last performed CVM in the CVMResults.
41          (setRequest (emvChipTechnologyGroup . cardholderVerificationMethodResults . cvmRule)
42                      (Just cvr) >>|
43
44           taskIf (performCVM action)
45              (stable ())
46              -- The CVM was not recognized, not supported, or it failed.
47              -- If the rule was CVMA_Fail, we try no further rules, regardless.
48              (if view failWhenUnsuccessful rule || action == CVMA_Fail
49                  -- If failure was critical, set the TVR bit (by calling the base case).
50                  -- otherwise continue with trying the remaining rules.
51                  then evalRules x y []
52                  else evalRules x y xs)
53          )
54          -- Condition not satisfied: try the next rule.
55          (evalRules x y xs)
56      where
57      action = view cvmAction rule
58
59
60  terminalSupportsCVM :: CVMAction -> Task Bool
61  terminalSupportsCVM CVMA_Fail                          = stable False
62  terminalSupportsCVM CVMA_NoCVMRequired                 = stable True
63
```

```
64  terminalSupportsCVM CVMA_PlaintextPINByICCAndSignature  =
65      terminalSupportsCVM CVMA_PlaintextPINByICC  ‘taskAnd‘
66      terminalSupportsCVM CVMA_Signature
67
68  terminalSupportsCVM CVMA_EncipheredPINByICCAndSignature =
69      terminalSupportsCVM CVMA_EncipheredPINByICC ‘taskAnd‘
70      terminalSupportsCVM CVMA_Signature
71
72  terminalSupportsCVM action =
73      -- For all other cases, we need to see if both the terminal and the acquirer
74      -- in charge supports the cvm corresponding to the supplied action.
75               getCard >>>
76      \card -> let terminalCVMs = view (managementTerminalGroup . terminalCapabilities
77                                     . cvmCapabilities) terminalConfig
78                   acquirerCVMs = view cardholderVerificationModes (cardBrandTable ! card)
79                in
80                stable $ any (match action) (terminalCVMs ‘intersect‘ acquirerCVMs)
81      where
82      match :: CVMAction -> CardholderVerificationMode -> Bool
83      match CVMA_PlaintextPINByICC    PlaintextPIN          = True
84      match CVMA_EncipheredPINOnline  EncipheredPIN_Online  = True
85      match CVMA_EncipheredPINByICC   EncipheredPIN_Offline = True
86      match CVMA_Signature            Signature             = True
87      match _                         _                     = False
88
89
90  -- Assumption : we recognize all CVMs, so we do not need to check for that.
91  -- Consequence: the "unrecognized CVM" bit in the TVR will never be set.
92  performCVM :: CVMAction -> Task Bool
93  performCVM action =
94      taskIf (terminalSupportsCVM action)
95        -- The CVM is supported, set it in the CVMResult
96        (performAction action)
97
98        -- The Terminal does not support this CVM, if this was a PIN operation,
99        -- set the corresponding TVR bit.
100       (setTVR pinEntryRequiredAndPinpadNotPresentOrNotWorking (isPINCVM action) >>| stable False)
101     where
102     isPINCVM CVMA_PlaintextPINByICC             = True
103     isPINCVM CVMA_EncipheredPINOnline           = True
104     isPINCVM CVMA_PlaintextPINByICCAndSignature  = True
105     isPINCVM CVMA_EncipheredPINByICC            = True
106     isPINCVM CVMA_EncipheredPINByICCAndSignature = True
107     isPINCVM _                                  = False
108
109
110 performAction :: CVMAction -> Task Bool
111 performAction CVMA_Fail          = stable False
112 performAction CVMA_NoCVMRequired =
113     -- This is the only case where CVM is successful.
114     -- Other successful actions will be "unknown" (see EVM4 6.3.4.5).
115     setRequest (emvChipTechnologyGroup . cardholderVerificationMethodResults . cvmResult)
116                (Just True) >>|
117     stable True
118
119
```

```
120 performAction CVMA_EncipheredPINOnline =
121                 requestPIN >>>
122   \pinResult -> case pinResult of
123      PINEntrySuccessful pin -> setRequest (onlinePINGroup . cardholderEncryptedPIN) (Just pin) >>|
124                                stable True
125      _                      -> stable False
126 performAction _ = stable False
127
128
129
130 -- PIN entry can be successful, cancelled or timed out.
131 requestPIN :: Task PINEntryResult
132 requestPIN = getCard >>>
133     \card -> enterPIN (view pinLengthType (cardBrandTable ! card))
134
135 -- "Encrypts" a PIN.
136 encryptPIN :: PIN -> Task EncryptedPIN
137 encryptPIN pin = stable (Encrypted pin)
138
139 -- A simple Task for entering the PIN.
140 -- PIN entry has a timeout; hance the maybe type.
141 enterPIN :: PINLengthType -> Task PINEntryResult
142 enterPIN lengthType = taskMaybe (withTimeout 10000 enterPINWithLength)
143                           -- PIN entry was cancelled
144                           (stable PINEntryCancelled)
145                           -- PIN entry was not cancelled
146                           (\pin -> if null pin then stable PINEntryTimeout
147                                                else encryptPIN pin >>>
148                                                     (stable . PINEntrySuccessful))
149     where
150     enterPINWithLength = case lengthType of
151                             PINLengthFixed4       -> enterPIN' 4 4
152                             PINLengthVariable4To6  -> enterPIN' 4 6
153                             PINLengthVariable4To12 -> enterPIN' 4 12
154
155     enterPIN' lower upper = (printInfo "Enter PIN: " >>| readLine <! invalid) ? map digitToInt
156         where
157         invalid = \input -> let n = length input
158                              in n < lower || n > upper || (not $ all isDigit input)
```

In section 4.3.2 the results of the proof of concept will be discussed. This example will be representative for the entire proof of concept.

4.3.2 Discussion

For the proof of concept, implementations were developed for each action defined in C-TAP (figure 4.3). Focus was placed on the flow of the transaction for a specific instance (i.e. EMV-based transactions with online authorization) with the possibility of flow interruption through card removal. This means many details were omitted in the proof of concept as discussed in section 4.1.1, for example the intricacies of things like communication with other systems. The proof of concept is not an application that will perform an actual transaction. A full implementation would have to take many more variables into account. The case study, and in particular the proof of concept, revealed that a task-oriented approach yields several benefits:

1. The actions of a transaction have a high level of independency. This makes it easy to remove or add them. The main reason for this high level of modularity is that the data that is needed by these actions is available through the environment, which is accessible by any task. The development of the proof of concept is summarized by the following steps:

   (a) Defining the set of data (i.e. `TerminalState`) that is to be recovered when the transaction is interrupted.

   (b) On the top level, start a transaction task using the state recovery function, and define a fallback procedure for the defined `TerminalState`.

   (c) Implement the transaction through a process of refinement without having to take card removal into account. Every sub-task of the transaction task has access to the `TerminalState`. Every C-TAP action is defined in its own module, which exports one task that performs the action.

2. The code is concise and reads much like the specification on which it is based. One of the reasons for this is that task combinators hide details that are irrelevant to the specification, like the state. Similarly, things like I/O would be abstracted over by shared data sources 2.4.2. In the proof of concept the latter is only done for communication with the ICC.

3. Because performing tasks in parallel is one of the core features of Task-Oriented Programming (section 2.4.5), dealing with interruptions that can occur at any time (e.g. card removal) becomes a lot easier than when this would not be the case. Because detection of such events can occur in parallel as opposed to having to poll for events periodically, the reaction time of a task-oriented solution to such events is short.

These findings support the hypothesis that a task-oriented solution for these systems is better maintainable and that it allows for a short reaction time to events that can occur at any time.

Chapter 5

Related Work

5.1 Payment terminal implementation at CCV

CCV develops multiple types of payment terminals, for example the one depicted by figure 5.1. Before going into the used programming paradigms and techniques, the architectural landscape of CCV payment terminals will be sketched. This will place the software discussion held later in the appropriate context. As suggested by the word “sketched”, the goal is only to provide the appropriate context, not an exact and complete overview.

5.1.1 Architectural landscape

A payment terminal is a device consisting of various components, the most important ones being the screen, keypad and card reader. The card reader is not always integrated into the payment terminal itself; some payment terminal models require an external one. Multiple components are optionally integrated, e.g. printers used for printing receipts.

Figure 5.1: a CCV payment terminal.

A payment terminal is installed at a merchant where it is part of the merchant’s payment setup. Aside from the payment terminal, this setup typically contains a cash register and things like a card reader if these are not part of the terminal itself.

The payment terminal also communicates with systems outside of the merchant’s payment setup. Examples of such systems are the Terminal Management System (TMS) for configuring and parameterizing the terminal, a logging server that allows CCV to analyze terminal behavior and an acquirer for payment processing.

Two software processes are present on the payment terminal: the payment application (i.e. the application considered in this thesis) and the secure module application (also referred to as the SCM). The main reason for splitting the software into 2 processes is that it makes certification easier. A terminal needs to pass certain certifications (e.g. PCI PTS 3.0 [Kin10], a security standard for PIN entry devices) to be usable in the field. By placing functional-

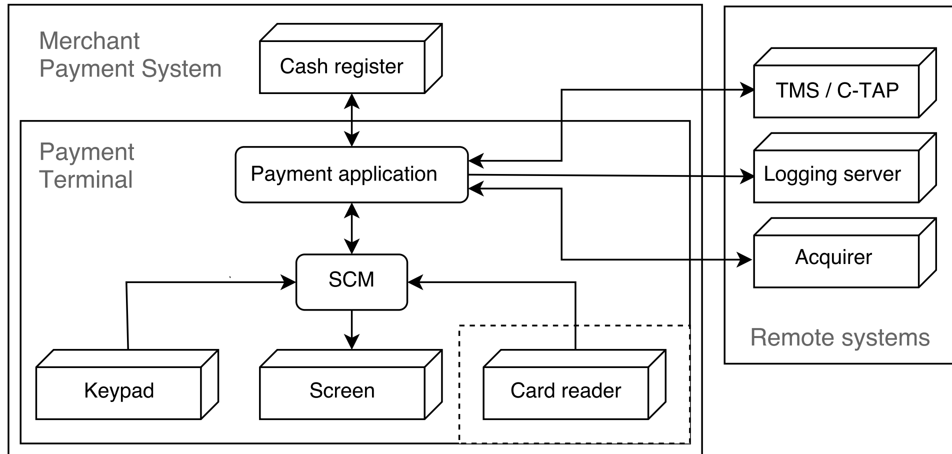

Figure 5.2: sketch of the CCV payment terminal landscape.

ity relevant for these certifications in the SCM, the amount of code subject to certification is minimized. From now on, "payment terminal software" will refer to the payment application. In short, a payment application on the terminal is not a stand-alone piece of software, but is interconnected with various other components having their own standards and protocols. This landscape is depicted by figure 5.2.

Within CCV, the payment terminals are subdivided into two categories: VxTree and CCV#. Payment terminals that fall under the former category are developed in the procedural programming language C. C used to be the programming language of choice because payment terminals had limited capabilities (e.g. a low amount of memory) [Bar98]. Nowadays, payment terminals (and portable electronic devices in general, e.g. smartphones) are equipped with much better hardware opening up more possibilities implementation-wise. Object-oriented programming is one of the most prominent programming paradigms today being followed by popular languages like Java, C# and C++ [Sta16]. Terminals that fall under the CCV# category are implemented in the object-oriented programming language C# [AA15]. The secure module (SCM) running on these terminals is implemented in C++ [Sav14]. In this thesis, focus is placed on the payment applications running on CCV# terminals.

5.1.2 Top-level payment application design

The software running on a CCV payment terminal is divided into *sessions*. A session is made up out of *actions*. Sessions can be derived from other sessions. For example, one type of session is a "payment session", used for processing a payment. Multiple types of sessions can be used where a payment session is needed: a terminal might perform a "C-TAP session" (section 4.1) when located at a department store or an "IFSF session" [IFS16] when used in the petrol sector.

The execution of sessions is controlled by a *session controller* running in its own thread. The session controller starts sessions one after another. In other words: at most one session is being executed at any given time. The session controller houses a (thread-safe) list of pending sessions from which it selects the next session to start once the running session finishes. This selection procedure is determined by logic built into the session controller (e.g. a maintenance session might get prioritized over a payment session).

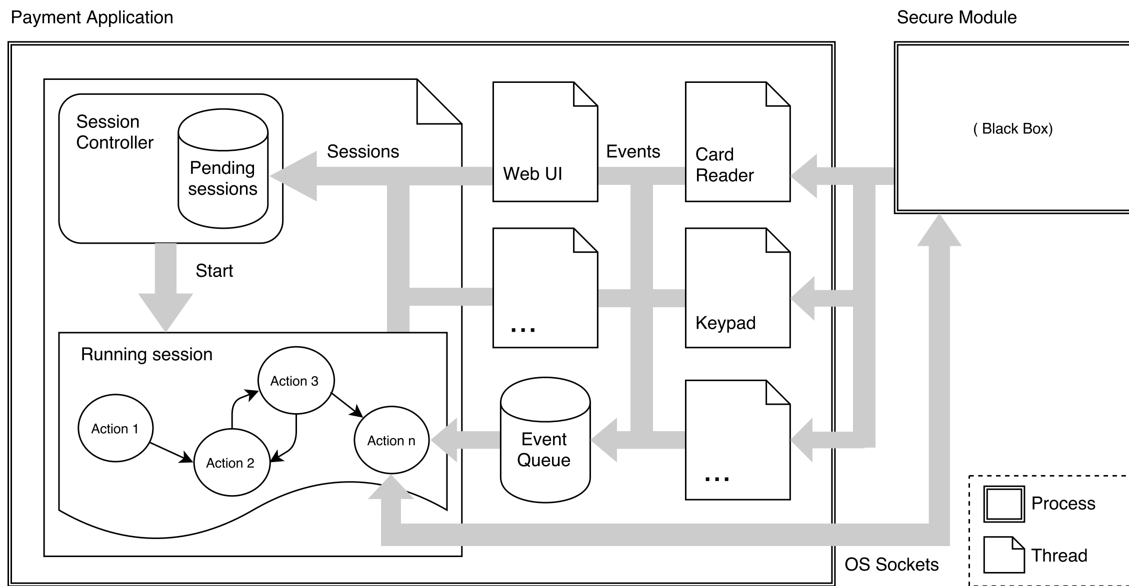

Figure 5.3: CCV terminal software architecture.

Events originate from event sources. For each event source, there is a thread for processing its events. Processing of events means placing them into a (thread-safe) *event queue* of which there is only one. Actions can obtain events from this queue and events arriving at the aforementioned threads will always be placed into the event queue, even when they are irrelevant to the currently execution action or session. Examples of event sources are the web UI accessed by a service engineer and the keypad and card reader accessed by the card holder.

Events originating from one of the terminal's hardware components (e.g. the keypad) go through the secure module, as shown in 5.2, because these need to be in a single process for certification reasons. As mentioned before, there is a thread for each event source and this is no different for events that arrive in the secure module process instead of the payment application process. In this case, a thread subscribes itself to a particular event type (e.g. keypad events) on the secure module. The secure module broadcasts the events it receives to its subscribers. The secure module acts as a server and the thread on the payment process acts as a client where (inter-process) communication occurs using OS sockets. Some activities, like PIN entry, are completely taken care of by the SCM because they are subject to strict security requirements. In these cases, related events will not be placed into the payment application's event queue.

Some event sources can also provide sessions. The web UI example mentioned earlier is one of these event sources. A service engineer may select an activity he wishes to perform (e.g. "initialize C-TAP") on the web UI. This will place the corresponding session in the session controller's session list and serves the requested page to the service engineer. Sections of the page that require session-related information are not yet loaded. Once the session is started, the UI components related to the session are loaded. The service engineer interacts with these pages, resulting in events being placed in the event queue accessible by the started session (or more specifically, the actions it is composed of). The discussed architecture is depicted by figure 5.3.

5.1.3 Problem areas

The discussed software architecture applies to many payment terminals that have been successfully operating in the field for years. Nonetheless, there is room for improvement in multiple aspects:

(1) The current architecture makes it easy to timely act on events when expecting them, but difficult when you do not. An example of an activity that involves expected events is PIN entry. It is clear that events related to PIN entry (i.e. keypad button presses) can be expected after asking the card holder to enter his or her PIN. No other activities are to be performed while the PIN is being entered, making it easy to deal with.

An example of an unexpected event is a cancellation request instantiated by the cashier through the cash register. This event is not expected by the terminal at any particular moment: it can occur at any time. It can occur for example while the card holder is entering his or her PIN. This scenario is depicted by figure 5.4. In this figure, the progress of the payment session on the terminal is informally represented by display messages. Strictly speaking, the display is under control of the SCM as shown in figure 5.2.

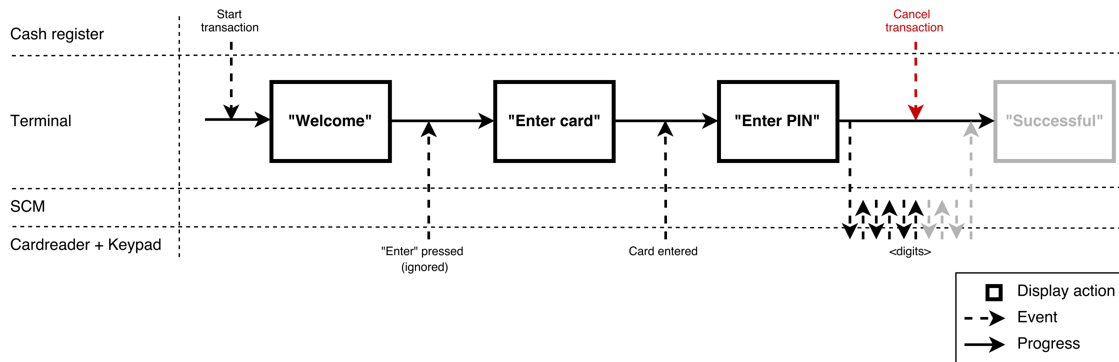

Figure 5.4: Interruption while processing other events.

There are two reasons for delayed event detection in the payment application:

a) Blocking communication with the SCM. The aforementioned PIN entry example falls under this category. PIN entry is functionality performed completely by the SCM, as discussed in section 5.1.2. The payment application invokes this functionality through a custom-made Remote Method Invocation (RMI) [HDH02] module. These RMI calls are blocking, meaning the terminal simply waits for them to complete. Because events are processed in separate threads, these will still be placed in the event queue. However, the terminal is not able to query the event queue until after the RMI call. As a partial solution, the SCM can throw exceptions. These exceptions can be caught by the payment terminal application and acted upon accordingly. This allows the terminal to act without delay on events that go through the SCM (e.g. card removal), but not on events that originate from sources not regulated by the SCM (e.g. the cash register or the web UI). This results in a delayed response time for those events, and potentially work being performed for nothing (e.g. finishing PIN entry while the transaction was canceled mid-way through by the cash register).

b) There is no real parallelism. Events are processed in separate threads, but the executing session is single-threaded. In some cases a finite hierarchical state machine [Sam08] is used to model the interaction between actions. This state machine supports state entry- and exit actions, transition guards and functions and actions on transitions. The difference between an action and a function here, is that an action cannot fail; it returns void. However, actions can throw exceptions. The state machine catches all exceptions and cancels the transitions they originated from, or in case of state entry it stops the state machine in a predefined erroneous state. When a transition is not defined for the executing state machine, the corresponding event is propagated to the

parent state machine if there is any. When a parent state machine transits, the child state machine is stopped, triggering the corresponding parent node exit actions.

In summary, the state machine implementation is fairly straight-forward. The possibility of defining a hierarchy allows for more concise definitions, but event-processing and action/function execution still happen sequentially. This results in a delayed event response because event-processing does not occur while executing an action or function (e.g. a state entry action). Regularly checking for the occurrence of events inside these actions and functions is possible, but this is obviously not a solution; it only shortens the delay but will not remove it. This also comes at the price of polluting the code with event-checking duplicates. The current code does not perform such checks. By consequence, it will not detect such events until attempting to utilize the components from which they originate. For example, removal of the card from the card reader is not detected until invoking card reader functionality (or, in case of a state machine, when processing events to determine the next transition to take).

(2) The maintainability of the code. Object-oriented programming promises easy maintainability compared to procedural programming due to high readability and reusability. This is not a guarantee however, supported by the fact that there is a lot of research on (automated) maintainability measurement through metrics, e.g. [LH93] and [DR11]. A typical way to improve the maintainability in object-oriented programs is to add a level of abstraction through design patterns [GHJV95]. The payment application makes extensive use of design patterns, the most common ones being:

a) The “factory” patterns “abstract factory” and “factory method”. The abstract factory pattern allows the creation of objects that are similar, without specifying their concrete classes. This is achieved by defining an abstract factory class that declares functions for creating abstract objects representing these similarities. The actual creation of these objects is done by the concrete factories that implement the interface of the abstract factory. A client uses the abstract factory to create (abstract) objects. This means a client has no notion of the concrete instance that is being used, making it more flexible. This concept is depicted by figure 5.5.

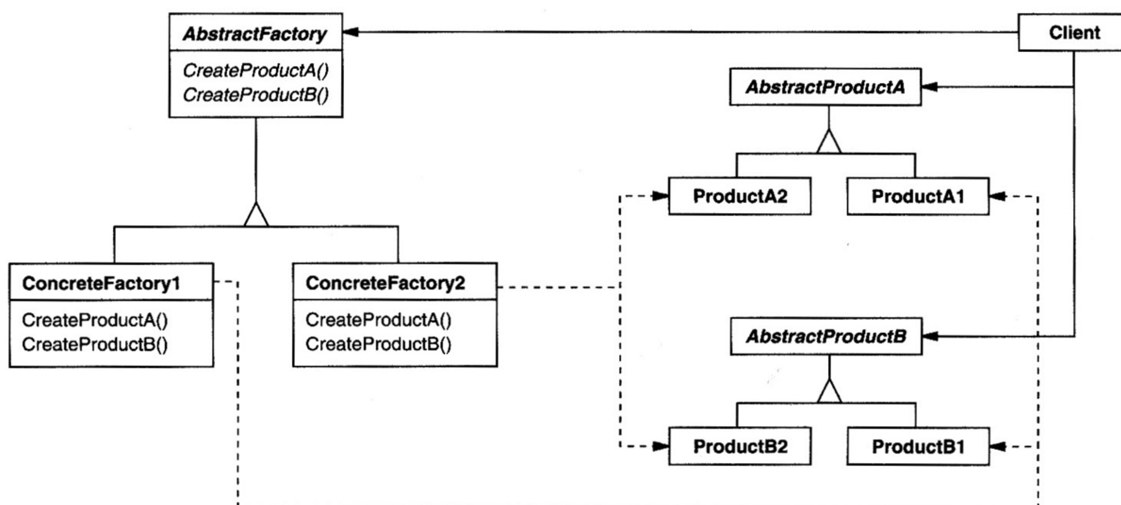

Figure 5.5: The “abstract factory” design pattern [GHJV95].

The factory method pattern is used when a subclass is to determine what concrete instantiation of a class to use, for example because the (abstract) parent class cannot

anticipate this. Literature often refers to the object that is to be created as the "product" and the abstract parent class as the "creator". The creator is given a factory method so that a subclass can instantiate the right product. Figure 5.6 summarizes this pattern.

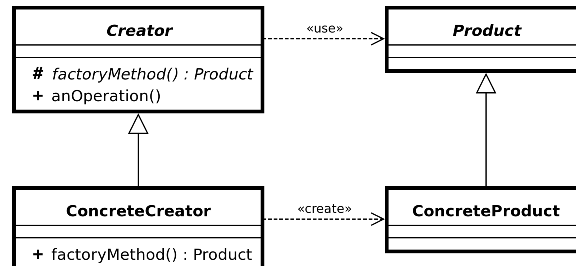

Figure 5.6: The "factory method" design pattern [GHJV95].

b) The "adapter" pattern. This pattern is used to convert one interface (the adaptee) to another (the target), much like a real-life adapter. Figure 5.7 depicts this pattern. In the payment application this pattern is used for example in an EMV adapter which abstracts over the calls to the SCM for EMV-related activities.

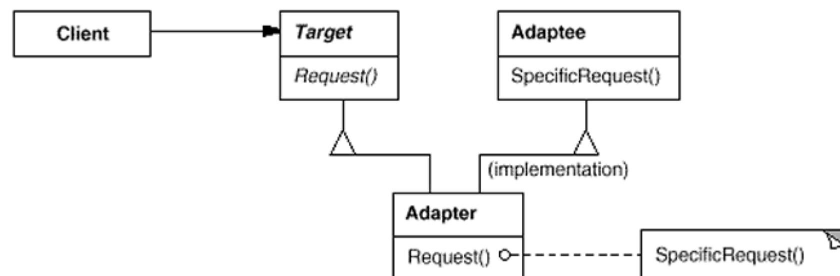

Figure 5.7: The "adapter" design pattern [GHJV95].

Another way the payment application aims to improve maintainability is by use of "dependency injection" [Mar04]. Dependency injection makes use of something called "inversion of control" for resolving dependencies. Conventionally, high-level modules depend on low-level ones. With dependency injection, the dependencies are supplied to the dependent object, resulting in independent high-level modules (also referred to as "loose coupling"). This is an example of inversion of control. The dependencies are interfaces instead of concrete implementations, allowing for greater flexibility (e.g. this allows testing of classes in isolation).

The payment application makes use of the "simple injector" framework [Sim] to achieve dependency injection. When a class depends on certain interfaces, they are supplied to the constructor of that class, as mentioned earlier. The website of "simple injector" provides the following example to illustrate this:

```
public class UserController : Controller {
    private readonly IUserRepository repository;
    private readonly ILogger logger;
```

```
    // Use constructor injection for the dependencies
    public UserController(IUserRepository repository, ILogger logger) {
        this.repository = repository;
        this.logger = logger;
    }

    // implement UserController methods here:
    public ActionResult Index() {
        this.logger.Log("Index called");
        return View(this.repository.GetAll());
    }
}

public class SqlUserRepository : IUserRepository {
    private readonly ILogger logger;

    // Use constructor injection for the dependencies
    public SqlUserRepository(ILogger logger) {
        this.logger = logger;
    }

    public User GetById(Guid id) {
        this.logger.Log("Getting User " + id);
        // retrieve from db.
    }
}
```

The `UserController` class depends on the interfaces `IUserRepository` and `ILogger`. Likewise, `SqlUserRepository` depends on `ILogger`. Using simple injector, implementations for these interfaces can be registered. One can register implementations either as "transient" or "singleton". In case of the former, a new object is created every time the interface is requested, while in case of the latter only one instance will be created. These registrations are performed using a container, as illustrated by the following example taken from the simple injector website:

```
protected void Application_Start(object sender, EventArgs e) {
    // 1. Create a new Simple Injector container
    var container = new Container();

    // 2. Configure the container (register)
    container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Transient);

    container.Register<ILogger, MailLogger>(Lifestyle.Singleton);

    // 3. Optionally verify the container's configuration.
    container.Verify();

    // 4. Register the container as MVC3 IDependencyResolver.
    DependencyResolver.SetResolver(new SimpleInjectorDependencyResolver(container));
}
```

Using this configuration, simple injector can create an object graph for the creation of an object. When requesting a `UserController` for example, the following object graph is created:

```
new UserController(
    new SqlUserRepository(
        logger),
    logger);
```

This makes constructing objects easy, especially when there are a lot of nested dependencies (e.g. in the shown example, the dependencies for `SqlUserRepository` were resolved too) as the resulting object graphs can become quite large.

While the discussed techniques certainly make the code more maintainable (and objects more reusable), there is still room for improvement with regards to maintainability:

a) Extensive knowledge about various techniques unrelated to the problem domain is required in order to work with the code. Examples of this are the techniques discussed earlier (i.e. design patterns and dependency injection) but also hierarchical finite state machines and remote method invocation (discussed in (1)). This makes that adding to the framework (e.g. a new session or action) requires some effort. For example, a factory needs to produce the action, the action must be registered for dependency injection, etc.

b) Dependency injection can greatly reduce code complexity but has also been a source of problems. The main culprit of these problems is the lifestyle (i.e. transient or singleton) of registered implementations. During development, it has occurred multiple times that someone decided that an implementation should have a different lifestyle. In a project of this scale, such a change can result in bugs that are difficult to track down, because the object is typically used in deep and complex object graphs.

c) The code is quite verbose at some points. One of the typical places where this shows, is class initialization (either through the constructor or through a separate initialization function). The following code snippet for example shows the `Initialize` member function of the `CtapInstance` class, which is the top module for the C-TAP payment session:

```
public void Initialize(IServiceProvider serviceProvider)
{
    ctapScheduling = serviceProvider.GetInstance<ICtapScheduling>();
    pinpad = serviceProvider.GetInstance<IPinpad>();
    selectableAidRecords = serviceProvider.GetInstance<ISelectableAidRecords>();

    configStore = serviceProvider.GetInstance<IConfigStore<CtapObjectName>>();
    sessionFactory = serviceProvider.GetInstance<ICtapSessionFactory>();
    tmsService = serviceProvider.GetInstance<ITmsService>();
    transactionInfoRetriever = serviceProvider
                              .GetInstance<ITransactionInfoRetriever>();
    ctapSessionFactory = serviceProvider.GetInstance<ICtapSessionFactory>();
    tmsConfig = serviceProvider.GetInstance<ITmsConfig>();
    keypadCapabilitiesProvider = serviceProvider
                                .GetInstance<IKeypadCapabilitiesProvider>();
    sredCardRecognition = serviceProvider.GetInstance<ISredCardRecognition>();
    adminServiceStatusWatcher = serviceProvider
                              .GetInstance<IStatusInformationNotification>();

    cardDetectionResponse = serviceProvider.GetInstance<ICardDetectionResponse>();
    followupTransaction = serviceProvider.GetInstance<IFollowupTransaction>();
```

```
    responseToEcr = serviceProvider.GetInstance<IResponseToEcr>();
    dateTime = serviceProvider.GetInstance<IDateTime>();

    var config = configStore.GetConfig<CtapConfig>(CtapObjectName.CtapConfig);

    var configChanged = false;
    if(config == null)
    {
        configChanged = true;

        config = new CtapConfig
        {
            Tcd = new Tcd(),
            AdditionalParameter = new AdditionalParameter(),
            Terminal = new TerminalGroup(),
            TerminalManager = new TerminalManagerGroup(),
            CardBrandLinkTable = new CardBrandLink[0],
            SecuritySchemeTable = new SecuritySchemeRecord[0]
        };
    }

    // be safe, don't reference but copy
    configChanged |= CopyArrayIfChanged(pinpad.Config
                                            .TerminalCapabilitiesByEnvironment
                                        ,ref config.Terminal.TrmCapa_9F33);
    configChanged |= CopyArrayIfChanged(keypadCapabilitiesProvider.GetCapabilities()
                                        ,ref config.Terminal.AddTrmCapa_9F40);
    configChanged |= config.Terminal.TrmType_9F35
                != (short)pinpad.Config.TerminalTypeByEnvironment;
    config.Terminal.TrmType_9F35 = (short)pinpad.Config.TerminalTypeByEnvironment;

    if(configChanged)
        configStore.SetConfig(CtapObjectName.CtapConfig, (object)config);

    Enabled = config.AdditionalParameter.AppEnabled > 0;

    if(config.Acquirers == null)
    {
        config.Acquirers = new AcquirerGroupCollection();
        configStore.SetConfig(CtapObjectName.CtapConfig, (object)config);
    }
    if(Enabled) {
        ctapScheduling.ScheduleInitialize(true, false);
    } else {
        ctapScheduling.RemoveAllScheduledItems();
    }
}
```

As shown, a lot of code is spent on object instantiation (e.g. obtaining the implementation from the given service provider, which is a module containing the dependency container). Similar to this example, a lot of code in the project is spent on defining things like factories and interfaces instead of actual functionality. This produces code that does not always read like the specification it is supposed to implement which reduces readability, and with that maintainability.

Chapter 6

Conclusion

This thesis tested the hypothesis that using Task-Oriented Programming for implementing *interruptible, non-distributed* embedded systems results in code with a higher maintainability and in an application that is able to more effectively deal with unexpected events, compared to modern-day object-oriented approaches.

A case study served as the main method of study. In preparation of this case study, a general-purpose Task-Oriented Programming framework given the name *µTasks* was developed, since the existing TOP framework, *iTasks*, targets a different category of applications (i.e. user interactive, distributed, multi-user applications). In the case study we looked at the financial transaction process performed by payment terminals. After analysing this process, we identified the core characteristics of the process in the context of interruptibility. Based on these characteristics, it was determined that defining the state of the process as a shared resource is the most effective approach for task-oriented implementation, because it allows state recovery after interruption without having to introduce new task types (and a corresponding set of combinators). The result of the case study was a proof of concept in which the aforementioned approach was applied using *µTasks* to implement a simplified financial transaction. In order to test our hypothesis, a modern-day object-oriented payment terminal implementation was analysed. A comparison between the task-oriented proof of concept and the object-oriented implementation yielded evidence supporting our hypothesis:

1. The task-oriented approach resulted in a higher level of modularity. Reasons for this are that (1) tasks have a high modularity by design (i.e. the progress of a task is only revealed through its typed interface) and (2) information required by a task-oriented module is available through the environment accessible by any task. Addition or removal of a module in the object-oriented solution proved to be more difficult, because a lot more knowledge is needed on the software architecture before knowing how to do so (e.g. used design patterns, dependency injection for obtaining dependencies, used (hierarchical) state machines).
2. In the object-oriented architecture it was easy to effectively deal with events when expecting them, but difficult when you do not. The primary culprit for this, is that the architecture does not contain any real parallelism. A threading model was used to remain responsive to events, but the transaction process itself was essentially single-threaded. The response time to events in the system was delayed, because their occurrence was only detected when trying to utilize functionality related to them (e.g. card removal was only detected when invoking card reader functionality). Another reason event handling was delayed for some events were blocking Remote Method Invocation (RMI) calls. This was partially solved through exceptions, but this is not a solution for events originating from a source other than the other end of the RMI call.

Due to the availability for combinators for parallelism, the task-oriented solution was able to deal more effectively with events that can occur at any time. A task for dealing with these events can be run in parallel to the main task, avoiding the definition of the latter to be polluted with code related to event handling. Because the events are handled in parallel, the reaction time for the task-oriented solution is short.

3. The code of the task-oriented solution reads much more like the specification it is based on than the object-oriented alternative. One of the reasons for this is that task combinators hide things like the environment in definitions that do not make use of it. Additionally, details like card I/O were abstracted behind shared data sources. The object-oriented solution contained a lot of code that was unrelated to the system's functionality. Examples of this are the definition of interfaces, code related to design patterns and object initialization and instantiation. This resulted in code that was significantly more verbose than the task-oriented code.

In summary, the case study provided evidence that in comparison to the analysed task-oriented implementation, the task-oriented alternative is able to more effectively deal with unexpected events and is better maintainable due to a higher level modularity and code that reads more closely like the specification.

6.1 Future work

1. By design, Task-Oriented Programming is well-suited for distributed applications. A future topic of research could be to determine the effectiveness of Task-Oriented Programming for implementing distributed interruptible systems.
2. The existing *iTasks* framework strongly evolves around user interactivity. It may be possible to separate the generation of user interfaces from the core combinators in a way that allows these combinators to be parametrized with a context. For example, a specification stating that an integer is to be obtained might result in a user interface being generated to obtain it from a user in a user interactive context, while an embedded context might obtain it from a database.
3. In this thesis we compared a task-oriented approach only against a modern-day object-oriented one. Further research can be done on different system design methods, like procedural languages like C or model-based environments like MATLAB.

Bibliography

[AA15] Joseph Albahari and Ben Albahari. *C# 6.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, Inc., 6th edition, 2015. http://dl.acm.org/citation.cfm?id=2911326. 2, 62

[Acq12a] Acquiris. C-TAP specifications - Document C-TAP.000 - Introduction to C-TAP, Version 10.0 (Final), March 2012. http://www.acquiris.eu/specifications. 46

[Acq12b] Acquiris. C-TAP specifications - Document C-TAP.200 - Online Terminal Functionality, Version 10.0 (Final), March 2012. http://www.acquiris.eu/specifications. 46, 47, 48

[Acq12c] Acquiris. C-TAP specifications - Document C-TAP.220 - Data Dictionary, Version 10.0 (Final), March 2012. http://www.acquiris.eu/specifications. 46

[AKLP13] P. Achten, P. Koopman, B. Lijnse, and R. Plasmeijer. Task-Oriented Programming, July 2013. dsl2013.math.ubbcluj.ro/files/Lecture/Plasmeijer.pdf. 1, 6, 7

[AKP13] P. Achten, P. Koopman, and R. Plasmeijer. An Introduction to Task Oriented Programming. In V. Zsók, Z. Horváth and L. Csató, editor, *Central European Functional Programming School, Revised Selected Papers*, volume 8606, pages 187–245. Springer International Publishing, Cluj-Napoca, Romania, July 2013. http://dx.doi.org/10.1007/978-3-319-15940-9_5. 2, 4, 5, 11, 16

[AP02] Artem Alimarine and Marinus J. Plasmeijer. A generic programming extension for clean. In *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, IFL '02, pages 168–185, London, UK, UK, 2002. Springer-Verlag. http://dl.acm.org/citation.cfm?id=647980.743392. 8

[Bar98] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1998. http://dl.acm.org/citation.cfm?id=552624. 1, 62

[BB00] Henk Barendregt and Erik Barendsen. Introduction to lambda calculus, March 2000. http://www.cs.ru.nl/~erikb/onderwijs/sl2/materiaal/lambda.pdf. 8

[BW88] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988. http://dl.acm.org/citation.cfm?id=113903. 8

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996. http://doi.acm.org/10.1145/234313.234418. 7

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. http://dl.acm.org/citation.cfm?id=2029110. 2, 5

[CLW+15] Y. Chen, X. Liu, L. Wang, C. Ji, Q. Sun, Y. Ren, and X. Wang. *Systems and Computer Technology: Proceedings of the 2014 Internaional Symposium on Systems and Computer technology, (ISSCT 2014), Shanghai, China, 15-17 November 2014*. CRC Press, 2015. https://books.google.nl/books?id=MNGYCgAAQBAJ. 7

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. http://doi.acm.org/10.1145/6041.6042. 7

[DR11] Sanjay Kumar Dubey and Ajay Rana. Assessment of maintainability metrics for object-oriented software system. *SIGSOFT Softw. Eng. Notes*, 36(5):1–7, September 2011. http://doi.acm.org/10.1145/2020976.2020983. 65

[EJ09] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009. http://dx.doi.org/10.1109/MC.2009.118. 1

[EMV11a] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Book 1: Application Independent ICC to Terminal Interface Requirements, Version 4.3, November 2011. https://www.emvco.com/specifications.aspx?id=223. 46

[EMV11b] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Book 2: Security and Key Management, Version 4.3, November 2011. https://www.emvco.com/specifications.aspx?id=223. 46

[EMV11c] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Book 3: Application Specification, Version 4.3, November 2011. https://www.emvco.com/specifications.aspx?id=223. 46, 50

[EMV11d] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, Version 4.3, November 2011. https://www.emvco.com/specifications.aspx?id=223. 46

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. http://dl.acm.org/citation.cfm?id=186897. 65, 66

[Gol96] Benjamin Goldberg. Functional programming languages. *ACM Comput. Surv.*, 28(1):249–251, March 1996. http://doi.acm.org/10.1145/234313.234414. 7

[HDH02] Per Brinch Hansen, Edsger W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. http://dl.acm.org/citation.cfm?id=548079. 64

[Hex14] Hexa Research. Embedded System Market Analysis By Product (Hardware, Software), By Application (Automotive, Telecommunication, Healthcare, Industrial, Consumer Electronics, Military & Aerospace) And Segment Forecasts, 2012 to 2020, May 2014. https://www.hexaresearch.com/research-report/embedded-system-industry/. 1

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. http://doi.acm.org/10.1145/1238844.1238856. 8

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 119–132, New York, NY, USA, 2000. ACM. http://doi.acm.org/10.1145/325694.325709. 5, 8

[HM00] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 2000. http://dl.acm.org/citation.cfm?id=555854. 7

[HS06] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.78.7007. 1

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996. http://doi.acm.org/10.1145/242224.242477. 5

[IFS16] Internation Forecourt Standards Forum (IFSF), 2016. https://www.ifsf.org/documents/ifsf-standards. 62

[Kin10] King, Jeremy. Understanding the PTS Security Requirements Version 3.0, May 2010. PCI SSC PTS Working Group, https://www.pcisecuritystandards.org/pdfs/webinar_100519pci_pts_3.0.pdf. 61

[LH93] Wei Li and Sallie Henry. Object-oriented software object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993. http://dx.doi.org/10.1016/0164-1212(93)90077-B. 65

[Lij13a] B. Lijnse. Evolution of a Parallel Task Combinator. In P. Achten and P. Koopman, editor, *The Beauty of Functional Code*, volume 8106, pages 193–210. Springer Berlin Heidelberg, 2013. http://link.springer.com/chapter/10.1007/978-3-642-40355-2_14. 14, 15, 27

[Lij13b] Lijnse, B. *TOP to the rescue: Task-Oriented Programming for incident response applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, 2013. http://hdl.handle.net/2066/103931. 1, 4

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. http://dl.acm.org/citation.cfm?id=2018642. 17

[Mar04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. Accessed: 2016-08-21, http://martinfowler.com/articles/injection.html. 66

[Mic12] Michels, Steffen and Plasmeijer, Rinus. Uniform Data Sources in a Functional Language, July 2012. http://wiki.clean.cs.ru.nl/images/3/3c/Sharing_Data_Sources.pdf. 12

[MLS08] Nathan Mishra-Linger and Tim Sheard. *Erasure and Polymorphism in Pure Type Systems*, pages 350–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. http://dx.doi.org/10.1007/978-3-540-78499-9_25. 7

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press. http://dl.acm.org/citation.cfm?id=77350.77353. 9

[Mye03] Brad A. Myers. Graphical user interface programming, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.631. 5

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. http://dl.acm.org/citation.cfm?id=280586. 7

[PA05] Rinus. Plasmeijer and Peter. Achten. The Implementation of iData - A Case Study in Generic Programming. In Butterfield, A., Grelck, C. and Huch, F., editor, *Proceedings Implementation and Application of Functional Languages - Revised Selected Papers*, 17th International Workshop, pages 106–123. Springer, Dublin, Ireland, September 19-21 2005. Department of Computer Science, Trinity College, University of Dublin. http://www.springer.com/us/book/9783540691747. 4

[PA06] Rinus Plasmeijer and Peter Achten. *iData for the World Wide Web – Programming Interconnected Web Forms*, pages 242–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. http://dx.doi.org/10.1007/11737414_17. 5

[PAK07] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 141–152, Freiburg, Germany, Oct 1–3 2007. ACM. http://dl.acm.org/citation.cfm?id=1291151. 1, 4, 5

[PJ89] S. L. Peyton Jones. Parallel implementations of functional programming languages. *Comput. J.*, 32(2):175–186, April 1989. http://dx.doi.org/10.1093/comjnl/32.2.175. 7

[Pla01] Plasmeijer, Rinus and van Eekelen, Marko and van Groningen, John. Clean Language Report - Version 2.2, December 2001. Accessed: 2016-08-16. http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf. 2, 4, 6, 8, 11, 17, 40

[PLM+12] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 195–206, New York, NY, USA, 2012. ACM. http://hdl.handle.net/2066/103802. 8, 11, 12, 13, 14, 15, 32

[Sab98] Amr Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, January 1998. http://dx.doi.org/10.1017/S0956796897002943. 7

[Sam08] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes (Elsevier Inc.), Newton, MA, USA, 2 edition, 2008. http://dl.acm.org/citation.cfm?id=1502169. 64

[Sav14] Walter Savitch. *Problem Solving with C++*. Addison-Wesley Professional, 9th edition, 2014. http://dl.acm.org/citation.cfm?id=2636552. 62

[Ses02] P. Sestoft. Demonstrating Lambda Calculus Reduction. In Mogensen, Torben Æ. and Schmidt, David A. and Sudborough, I. Hal, editor, *The Essence of Computation: Complexity, Analysis, Transformation*, pages 420–435. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. http://dx.doi.org/10.1007/3-540-36377-7_19. 7

[SGG08] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. http://dl.acm.org/citation.cfm?id=1412284. 1

[Sim] Simple Injector Contributors. Simple Injector. Accessed: 2016-08-21, https://simpleinjector.org/. 66

[Sta94] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, dec 1994. University of Cambridge Computer Laboratory, http://www.inf.ed.ac.uk/~stark/namhof.pdf. 8

[Sta16] Stack Overflow Developer Survey Results 2016, Most Popular Technologies, 2016. Accessed: 2016-08-05, http://stackoverflow.com/research/developer-survey-2016#technology-most-popular-technologies. 1, 62

[Ste93] W. Richard Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. http://dl.acm.org/citation.cfm?id=161724. 5

[VP03] Martijn Vervoort and Rinus Plasmeijer. *Lazy Dynamic Input/Output in the Lazy Functional Language Clean*, pages 101–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. http://dx.doi.org/10.1007/3-540-44854-3_7. 12

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM. http://doi.acm.org/10.1145/91556.91592. 9