UNIVERSITY OF BELGRADE
FACULTY OF MATHEMATICS

Ivan Čukić

FUNCTIONAL AND IMPERATIVE REACTIVE PROGRAMMING BASED ON A GENERALIZATION OF THE CONTINUATION MONAD IN THE C++ PROGRAMMING LANGUAGE

doctoral dissertation

Belgrade, 2018

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Иван Чукић

ФУНКЦИОНАЛНО И ИМПЕРАТИВНО РЕАКТИВНО ПРОГРАМИРАЊЕ УПОТРЕБОМ ГЕНЕРАЛИЗАЦИЈЕ МОНАДЕ НАСТАВКА У ПРОГРАМСКОМ ЈЕЗИКУ C++

докторска дисертација

Београд, 2018

Advisor:

dr Nenad Mitić, associate professor
University of Belgrade, Faculty of Mathematics

Committee:

dr Miodrag Živković, full professor
University of Belgrade, Faculty of Mathematics

dr Saša Malkov, associate professor
University of Belgrade, Faculty of Mathematics

dr Vladimir Filipović, associate professor
University of Belgrade, Faculty of Mathematics

dr Zoltan Porkolab, associate professor
Eötvös Loránd University, Faculty of Informatics

Date of the defense:

Ментор:

др Ненад Митић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Миодраг Живковић, редовни професор
Универзитет у Београду, Математички факултет

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Владимир Филиповић, ванредни професор
Универзитет у Београду, Математички факултет

др Золтан Порколаб, ванредни професор
Еотвош Лоранд Универзитет, Информатички факултет

Датум одбране:

Dissertation title: Functional and imperative reactive programming based on a generalization of the continuation monad in the C++ programming language

Abstract: There is a big class of problems that require software systems with asynchronously executed components. For example, distributed computations have the distributed nodes that process the data asynchronously to one another, service-oriented architectures need to process separate requests asynchronously, and multi-core and heterogeneous systems need to have multiple separate tasks running concurrently to best utilize the hardware. Even ordinary GUI applications need asynchronous components – the user interface needs to be responsive at all times which means that no matter in what state the program is in, it needs to process and react to the input events coming from the user. The necessity of concurrency and asynchronous execution brings in the added complexity of the *Inversion of Control* (IoC) into the system, either through message passing or through event processing. IoC makes code difficult to develop and reason about, it increases component coupling and inhibits clean functional or object-oriented software design.

In this dissertation, a method for solving the problems that IoC introduces is presented. It presents a way to model both synchronous and different types of asynchronous tasks with the continuation monad. The continuation monad serves as a primitive to build more complex control flow structures that mimic the control flow structures of the host programming language. It also allows for building more complex control structures specialized for parallelism, transactional execution, and for simulating functional programming idioms with asynchronous tasks through a generalization of the continuation monad that allows the asynchronous tasks to generate results one at a time. This allows for writing programming systems with asynchronously executed components by writing seemingly synchronous imperative or functional code while leaving it up to the compiler to do all the heavy lifting and convert the written code to asynchronously executed set of tasks. Another benefit of the presented method is that it allows for easier automatic handling of the data lifetime without the need for garbage collection.

This method has been successfully applied and tested in several Free/Libre Open Source Software and proprietary real-world software projects used by hundreds of millions of people around the world. In this dissertation, an example of a secure project management system is described which is based on a similar system implemented as a part of the KDE Plasma project. This dissertation also contains the important parts of the implementation of the AsynQt library which extends

the Qt library, and its concurrency primitive – QFuture class – with functional reactive programming patterns based on the method proposed in this dissertation.

Keywords: coordination, asynchronous task-oriented design, domain specific languages, programming language design, functional programming, imperative programming, reactive programming, category theory, continuation monad
Research area: Computer Science
Research sub-area: Software development
UDC number: 004.415:004.438C++ (043.3)

Наслов дисертације: Функционално и императивно реактивно програмирање употребом генерализације монаде наставка у програмском језику C++

Резиме: Постоји велики број проблема који захтевају писање програмских система који имају компоненте које се извршавају међусобно асинхроно једне од других. На пример, код дистрибуираних израчунавања постоје дистрибуирани чворови који обрађују податке асинхроно једни од других, системи са сервисно-оријентисаним архитектурама морају да обрађују захтеве које добијају од клијената асинхроно, а у више-језгарним и хетерогеним системима постоји више одвојених задатака који се извршавају истовремено ради што бољег искоришћења хардверских ресурса. Чак и обични графички програми имају асинхроне компоненте – графичко корисничко окружење не сме да буде заглављено ни у једном тренутку, што значи да без обзира на то шта програм тренутно ради, не сме да престане да обрађује и реагује на улазне догађаје који долазе од корисника. Неопходност асинхроног извршавања доноси додатно усложњавање програма због уметања извртања контроле извршавања (*IoC – Inversion of Control*), а због употребе система за асинхрон пренос порука или због употребе система за обраду догађаја. Због извртања контроле извршавања, програм постаје тежак за развој и тешко разумљив.

У овој дисертацији приказан је један метод за решавање проблема који настају због извртања контроле извршавања. Представљен је начин за моделовање и синхроних и различитих типова асинхроних задатака употребом монаде наставка. Монада наставка служи као основна примитива за изградњу сложенијих структура контроле тока извршавања програма које симулирају структуре контроле тока извршавања програмског језика C++. Такође је представљено да монада наставка дозвољава изградњу напреднијих структура контроле тока програма које су специјализоване за паралелизацију или за трансакционо извршавање. Представљен је и метод за симулирање концепата функционалног програмирања кроз генерализацију монаде наставка која дозвољава асинхроним задацима да генеришу више резултата независно. Показано је да ови методи омогућавају имплементацију програмских система са асинхроно извршеним компонентама писањем наизглед синхроног императивног или функционалног кода, док је преводиоцу препуштено да направи асинхроне задатке и преведе написани код у програм код ког се делови извршавају асинхроно. Додатна предност представљеног метода је што омогућава аутоматизацију руковања животним веком података у програму без потребе за скупљачем отпадака.

У уводном поглављу (поглавље 1) је укратко описана мотивација, доприноси и организација дисертације. Поглавље 2 садржи детаљнији опис проблема који методи развијени у оквиру ове дисертације решавају, постојеће начине имплементације програмских система са асинхроним компонентама, као основне појмове теорије категорија. Употреба монаде наставка у контексту асинхроног програмирања и концепта *будућих* вредности је описана у поглављу 3. Ово поглавље дефинише и реактивне токове (генерализацију монаде наставка, као и трансформације над њима) који омогућавају имплементацију програмских система као тока података и низа трансформација над тим током. Поглавље 4 дефинише структуре контроле тока програма базиране на монади наставка, док поглавље 5 приказује имплементацију. Последње поглавље садржи дискусију и поређење уобичајених метода за имплементацију програмских система са асинхроно извршаваним компоненама са методима представљеним у овој дисертацији. Додаци дисертацији садрже историју идеја презентованих у дисертацији (додатак A), анкету заједнице професионалних програмера о читљивости кода имплементираног методима представљених у дисертацији у односу на алтернативе (додатак B), као и примену функционално реактивне технике на библиотеку Qt (додатак C).

Методи представљени у овој дисертацији су успешно примењени и тестирани у неколико слободних и власничких софтверских пројеката које користе стотине милиона људи широм света. У овој дисертацији описан је пример система управљања пројектима који се заснива на сличном систему који је део пројекта *KDE Plasma*. Ова дисертација такође садржи важне делове имплементације библиотеке *AsynQt* која проширује библиотеку *Qt*, и њену примитиву за конкурентно извршавање – класу *QFuture* – функционалним реактивним програмским обрасцима заснованим на методу предложеном у овој дисертацији.

Кључне речи: координација, дизајн асинхроних система базираних на задазима, доменски програмски језици, дизајн програмских језика, функционално програмирање, императивно програмирање, реактивно програмирање, теорија категорија, монада наставка
Научна област: рачунарство
Ужа научна област: развој софтвера
УДК број: 004.415:004.438C++ (043.3)

Contents

1 Introduction

1.1 Motivation

There is a big class of problems that require software systems with asynchronously executed components (we will call these *asynchronous software systems*). For example, distributed computations have the distributed nodes that process the data asynchronously to one another, service-oriented architectures need to process separate requests asynchronously, and multi-core and heterogeneous systems need to have multiple separate tasks running concurrently to best utilize the hardware. Even ordinary GUI applications need asynchronous components – the user interface needs to be responsive at all times which means that no matter in what state the program is in, it needs to process and react to the input events coming from the user. The necessity of concurrency and asynchronous execution brings in the added complexity of the *Inversion of Control* (IoC) into the system, either through message passing or through event processing. The IoC happens because each time we want to execute an asynchronous task, we need to schedule to receive an event to notify us that the task is finished, so that we can process the result. This means that we are giving up the control over the execution of our functions to an external entity – usually to the event processing loop.

The IoC requires the software system to be split into smaller components in an artificial manner. Instead of splitting the system into logical sub-components or sub-routines, the system needs to be split at every invocation of an asynchronous task – one component or routine to start the task, and a separate one to process the result. This separation significantly increases the component coupling and inhibits clean functional or object-oriented software design.

This leads to the software maintainability issues, because changing the behaviour of one component can have significant impact on the rest of the system, while at the same time making it borderline impossible to perform automated testing of isolated components, without mocking all other components that the tested component needs to interact with. It also hinders readability and understandability of the software system, both to developers who need to have an overview of the whole system to be able to understand each separate component, and to static code analyzers which usually fail to see all the interdependencies to be able to effectively analyze the code.

The main aim of this dissertation is to *design a method and a framework for writing software systems that require asynchronous execution of certain tasks. This method should allow programmers to implement asynchronous software systems in*

the same way they would implement programs which do not need any asynchronous execution, leaving it up to the framework and the compiler to automatically convert the functional or imperative code designed in a synchronous-like style to work asynchronously.

1.2 Contributions

During the work on this dissertation, a new method for implementing software systems that have the need for asynchronous execution has been developed. This method has been successfully used in several real-world software solutions. More specifically, the main contributions of this dissertation are as follows:

- Designed a common abstraction over synchronous and different types of asynchronous tasks based on the continuation monad, which allows the programmer to use the same primitives when implementing a software system regardless of whether she is dealing with synchronous or asynchronous tasks. This abstraction has been presented initially at the *Qt Dev Days 2013* conference [Čuk13] in Berlin, Germany;
- Implemented a functional reactive programming (FRP) framework for the C++ programming language for expressing the logic of asynchronous software systems as a series of transformations over the input data streams based on the reactive streams concept. This framework makes implementing asynchronous software systems equivalent to implementing synchronous software systems in the functional style. The fundamentals of this work have initially been presented at the *Central-European Functional Programming Summer School 2015* in Budapest, Hungary [Čuk15a], with an expanded solution accepted for publication in the *Springer's LNCS* series;
- Developed a memory optimization for the data-flow system of the implemented FRP framework which provides memory safety guarantees without the need for making excessive data copies and without the need for garbage collection which is an integral part of most FP languages and systems. This work was presented in a plenary talk at the *C++ Siberia 2017 conference* held in Tomsk, Russia [Čuk15b];
- Created a solution for chaining synchronous and asynchronous tasks using primitives that mimic the control flow constructs of the host programming language to allow writing asynchronous software systems without the need to split programs into small functions which are not logical subroutines of the

program due to the inversion of control (IoC). This work was presented at the Meeting C++ 2014 conference held in Berlin, Germany [Čuk14], and *C++ Russia 2015* [Čuk15c] conference held in Moscow, Russia, and later published in the *Software: Practice and Experience* journal [Čuk16a];

- The developed methods were tested and benchmarked in practice. It was demonstrated that, when implemented correctly, the concepts from the proposed FRP and IRP frameworks have no induced runtime performance penalties, while providing significant improvements to the code readability as shown by the results of the professional C++ community survey;
- An implementation of the IRP framework concept over a 3rd party library to demonstrate that the defined concepts are not dependent on particular platform or framework. This has been presented at *QtCon 2016* [Čuk16b] in Berlin, Germany.

1.3 Organization

Motivation and background overview (Chapter 2) contains the basic concepts and techniques needed for understanding the rest of this dissertation. The first part (Section 2.1) gives an overview of the problems of implementing software systems whose parts need to be executed asynchronously. This part defines tasks and presents the differences between synchronous and various types of asynchronous tasks. It also shows the common approaches to dealing with asynchrony, and the downsides they all bring (Section 2.2). The second part gives an introduction to C++ algorithms, iterators and ranges (Section 2.3) and an overview of the advanced C++ techniques used in this dissertation (Section 2.4). The last part gives a short introduction to category theory as it provides some important mathematical background to the concepts used in this dissertation, and is not widely known amongst C++ programmers and imperative programming academic community;

Functional reactive programming (Chapter 3) discusses the continuation monad in the context of the asynchronous programming and the future value primitive often used to represent the result of different asynchronous operations (Section 3.1). It presents reactive streams – a relaxation of the futures concept that allows the asynchronous operation to keep returning new results after the first result is calculated. This chapter defines the reactive streams by comparing them against C++ iterators and ranges, and presents

the algorithms and transformations that can be used with reactive streams. It discusses the effect of having stateful transformations in an asynchronous system based on reactive streams, and how the data-flow-based software design can make the implementation of an asynchronous software system simpler (Section 3.2).

Imperative reactive programming (Chapter 4) defines the flow control structures – schedulers – based on the continuation monad which allow the programmer to compose synchronous and different types of asynchronous tasks in an imperative way – similar to the flow control structures for synchronous programming provided by the host programming language. This chapter defines (Section 4.2) schedulers that simulate serial execution, loops and branches, and logic operator schedulers which can be used for easy error handling. It also defines (Section 4.3) schedulers that go beyond the flow control structures available in the C++ programming language like the transaction scheduler which executes all the tasks successfully, or if a single task fails, undoes all the tasks like no task has been executed; and also schedulers for parallel task execution to demonstrate that the scheduler abstraction can also cover the traditional parallelization techniques like the fork-join pattern [Hal13]. Finally, it defines (Section 4.4) the data lifetime which simulates the scoping mechanism of the C++ programming language [ISO17, basic.scope] without the need for garbage collection which is commonly required in the alternative approaches.

Implementation (Chapter 5) covers the possible implementation of the scheduler concept using the event loop as the *behind-the-scenes* synchronization mechanism. This chapter shows a common abstraction of all types of tasks which allows synchronous and various types of asynchronous tasks to be handled with the same syntax. It shows the implementation of the serial scheduler in detail (Section 5.2.3) and that the compiler can optimize the code when this implementation is used to be as efficient as hand-written code implemented using the call-callback approach (Section 5.3). This chapter also discusses the negative impact on the compilation times used by this approach (Section 5.4).

Discussion and conclusion (Chapter 6) shows that the approach to writing software systems with asynchronous components presented in this dissertation makes writing these software systems as easy as writing ordinary synchronous code (Section 6.1). It compares the proposed solution to general coroutines (Section 6.2.1), to resumable functions (stackless coroutines sched-

uled for inclusion in C++20, Section 6.2.2), to stateful coroutines provided by the Boost.Asio library [Koh] (Section 6.2.3) and to actor systems (Section 6.2.4). It also demonstrates that this approach allows easier automated testing of asynchronous systems, and that the defined abstractions allow the static analyzer tools to easier comprehend the code and find errors in it.

Appendices provide additional information about the proposed solution. The origin of the ideas presented in this dissertation is explained in Appendix A. Appendix B contains a small survey of the professional C++ community on the readability and understandability of code written using the imperative reactive programming approach presented in this dissertation. Finally, Appendix C shows how the presented functional reactive programming techniques can be applied to a third-party library.

2 Motivation and background overview

2.1 Interactive and concurrent software systems

Most modern software systems require different system components to be executed concurrently. All interactive systems need to be able to process available data while waiting for further interactions – from simple GUI applications which need to process system events while waiting for user input, and multi-threaded applications that perform complex calculations to multi-client servers which need to process multiple client requests at the same time.

Since the terminology is not fully standardized, we will use the term *concurrency* to denote different tasks being executed in the same time-frame. This does not necessarily mean that these tasks are executed in separate threads – we will also consider tasks that are executed cooperatively on a single thread, or tasks from separate system processes to be concurrent tasks. We will cover the different task types in Section 2.1.2.

A significant fraction of software bugs happen due to programmers not being able to fully understand all the possible states their code can be in. In a concurrent environment, this lack of understanding and the issues that arise from it are greatly amplified [Car].

2.1.1 Blocking and synchronization

The usual approach of dealing with concurrency is through using plain threads and mutexes. The problems with plain threads are numerous. The main problem is that the threads are not composable [Lee06]. It is impossible to tell whether another library or a called function creates new threads by itself. This can lead to creating many more threads than the system can effectively handle. At the same time, many of those threads will just be waiting for the data that is calculated by other threads.

Another issue with threading is that concurrency can not be disabled. When using plain threads, the program logic usually becomes dependent on different code paths running at the same time. In that case, it is not easy to modify the code to run on a single thread which could be useful in a number of scenarios, especially for testing and debugging purposes.

Threads also have no standard way to return values to the caller. It usually involves using a shared variable and manual synchronization through locks.

When programmers want to make their program run faster they tend to throw raw power at it by improving CPUs clocks and adding new cores which is ineffective

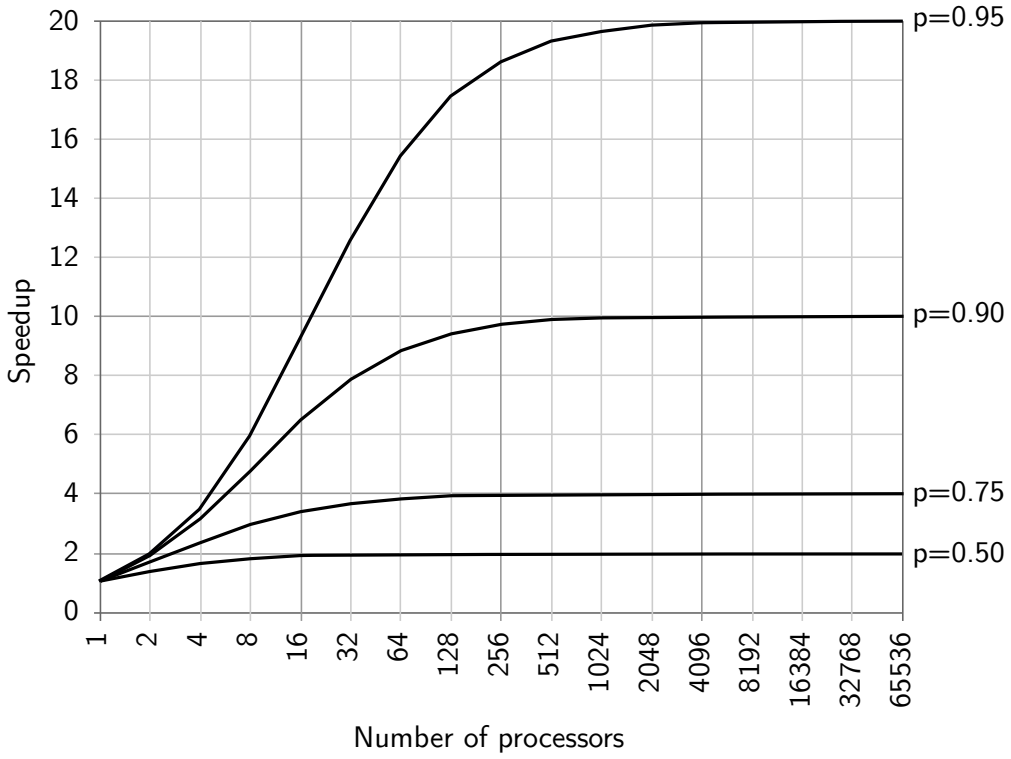


Figure 2.1: Amdahl's Law

according to the Amdahl's Law. Amdahl's law [Rod85] gives us the theoretical limit for the speed-up that can be gained by increasing the number of processing units in our system depending on the percentage of the code which is not possible to parallelize:

$$U(s,p) = \frac{1}{(1-p) + \frac{p}{s}},$$

where U is the theoretical speed-up for the process, s the relative increase of the number of processing units, and p the percentage of the code that is parallelizable. This means that for the system where 95% of the code can be fully parallelized to run on all available processors, and only 5% of the code needs to be executed serially, the maximum theoretical speed-up will be 20 times no matter how many processing units are added (when $s \to \infty$) as shown in Figure 2.1.

In a discussion on recursive mutexes, David Butenhof noted that this basic synchronization primitive should have been called "the bottleneck" [But] instead of creating a "cute acronym" for it (from **mut**ual **ex**clusion). Bottlenecks are useful at times, sometimes indispensable – but they are never good.

These problems all lead to decrease in execution speed. Most threads just end up waiting for the results of other computations to be sent to them [Par15]. Yet, the operating system still has to do all the book-keeping and context-switching to manage all those threads, which is just bureaucracy that takes the CPU time from the actual program that needs to be executed.

2.1.2 Different task types

Tasks are traditionally considered to be asynchronously executed moderately complex sub-systems of the main software system. This definition is overly restricting, and we will use the term *task* in a more relaxed sense. We will consider any piece of code to be a task if it has the following properties:

- it can be started;
- we know when its execution has finished;
- we know whether it was executed successfully or not.

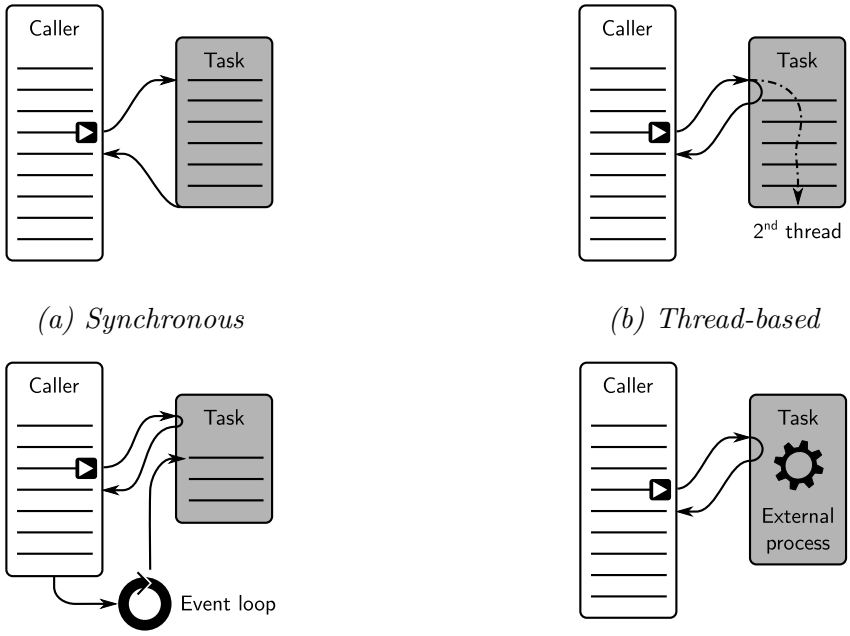
It makes no difference whether the code is executed synchronously or asynchronously, how complex it is, or where it is executed.

We will separate the tasks with regard to the type of execution into the following groups:

Synchronous tasks are the tasks that do not return the control back to the caller before they are finished. Therefore, these tasks should not perform any time-consuming operations. Good examples of synchronous tasks are short functions like `memcpy`, `snprintf` or `strcmp`. Functions like `scanf` can also be considered tasks, but should not be used because they could block the caller indefinitely.

Thread-based asynchronous tasks start a new thread in which the asynchronous operation is executed. These tasks return the control to the caller immediately after thread creation, without waiting for it to finish. The task needs to send a signal to the caller when it finishes execution. (e.g. tasks based on `std::async` or `std::thread`).

Event-based asynchronous tasks are almost the same as the thread-based ones, with the exception that they do not have to run in a separate thread, but are executed by the event loop in the main thread. If designed properly, these tasks have most of the benefits, and no drawbacks compared to the threaded ones [Ous96] (e.g. user interface and interaction, reactive network communication) which makes them the preferred type of tasks in most cases.



(a) Synchronous *(b) Thread-based*

(c) Event-based *(d) Out-of-process*

Figure 2.2: Task types

Out-of-process tasks provide a similar interface to the caller as the previous asynchronous task types, but are not executed in the same process as the caller. These tasks can be modeled as the event-based asynchronous tasks by listening to the process finished event that will be sent by the operating system when the out-of-process task ends (e.g. executing external processes, network distributed computation).

Other asynchronous task types like thread-reusing or fibre-based tasks, tasks computed on specialized hardware (GPUs, heterogeneous cores), and similar can easily be modelled after some of the aforementioned types, so we will not cover them separately.

2.1.3 Asynchronous task execution and the inversion of control

Programmers usually think about program logic in terms of algorithmic steps, loops and branches (in the case of imperative programming) and data transformation (in the case of functional programming) which are easy to visualize and reason about. While this is a natural way to think about the control and data flows, the modern software systems usually can not be implemented in this way.

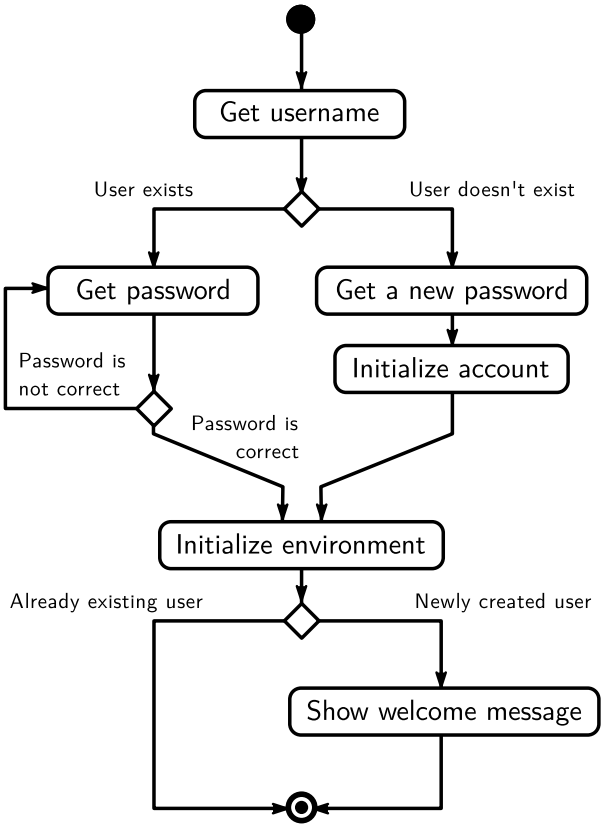


Figure 2.3: User authentication algorithm

Using ordinary control structures like loops and branching stops being a viable option as soon as there are slow operations that need to be performed, or when the program does not have all the needed data available, which can happen if the data is sent to our system in chunks (slow database access, media streaming over network, etc.) or if the program is getting the data from the user. Waiting for operations to finish, or waiting for the data to arrive blocks the current process which, in the case of a system logic implemented using the ordinary control structures, means that the whole system is blocked.

While one part of the software system executes a slow operation or is waiting for the data it needs to proceed, other parts of the system need to be able to continue working. For example, in a GUI program, the program needs to be able to to continue processing the operating system events while processing the data entered by the user; network services need to be able to accept new clients while waiting for the response from an existing one, and similar. This, combined with the complexity that threads introduce, has led to abandonment of blocking APIs in the modern systems [McC11]. Instead, the asynchronous approach is used.

An asynchronous API call returns the control to the caller immediately – before the action it represents is executed, often even before the action is even started. In essence, it only tells the system that it wants a specific action performed, and that it wants to be notified when said action has been completed. This allows the process to continue performing other tasks.

Removing the periods of time where our system is idle improves latency, but adds significant complexity to the implementation. The parts of the algorithm that depend on the result of the asynchronous call need to be moved to a separate function that handles the completion notification message. This is called *inversion of control* (IoC) because there is an external agent that is invoking the code that the programmer writes by sending it messages.

Inversion of control leads to program logic fragmentation – the program logic needs to be split into many asynchronous result handler functions (or classes, as in the state design pattern [Gam+95]) where these functions are not necessarily logical subroutines of the program. The control flow becomes implicit through manipulation of shared state which is not visible nor accessible by the programmer writing the software [CM06].

Let us look at an example of a simplified user creation and authentication process as presented by the activity diagram in the Figure 2.3 (a more complex example is covered in Section 6.1.2). Translating this diagram to any imperative programming language seems straightforward. Implemented in C++, it would look like this (with variable definitions omitted for readability):

```
void login()
{
    username = get_username();
    new_user = !check_if_user_exists(username);

    if (new_user) {
        password = get_password();
        initialize_account(username, password);
    } else {
        do {
            password = get_password();
        } while (!check_credentials(username, password));
    }
```

```
    initialize_environment();
    if (new_user) show_welcome_message();
}
```

This is a naive implementation and not something that should be done in a real world system. Most tasks in this example need to be executed asynchronously in order not to block the execution of the rest of the system. Namely, the tasks that wait for the user input (`get_username` and `get_password`), tasks that communicate with external agents (`check_if_user_exists` and `check_credentials` might need to contact a network service that stores the user information), or tasks that are expensive to execute (`initialize_account` and `initialize_environment`). When this problem is taken in account, the code becomes more complicated because each asynchronous task requires the programmer to separate the code that handles the result of a task from the code that starts that task:

```
class login_process {
    void login()
    {
        get_username(on_got_username);
    }

    void on_got_username(std::string username)
    {
        m_username = username;
        check_if_user_exists(username,
                             on_user_existence_checked);
    }

    void on_user_existence_checked(bool user_exists)
    {
        m_new_user = !user_exists;
        get_password(on_got_password);
    }

    void on_got_password(std::string password)
    {
        if (m_new_user) {
            initialize_account(m_username, password);
```

```
        } else {
            if (!check_credentials(m_username, password)) {
                get_password(on_got_password);
                return;
            }
        }
        …
    }

private:
    std::string m_username;
    bool m_new_user;
};
```

The program becomes significantly more complex by handling the asynchrony of only three calls (`get_username`, `check_if_user_exists` and `get_password`) while ignoring the rest. Instead of having the logic of a process in one place, the code gets scattered into functions which are not logical subroutines of it.

As a side-effect, it is also necessary to create an object to keep the state which raises issues in multi-threaded environments; or to pass them as call/callback arguments which would break encapsulation and make the API more complex. In the first implementation, those variables would have been local, which is not possible any more.

2.2 Traditional approaches to dealing with asynchronous operations

The increased complexity that arose from making the example from the previous section asynchronous is not specific to calls and callbacks. Any abstraction that deals with asynchronous operations by forcing the programmer to separate the task starting from the result handling will have the same problem.

In this section, we cover the most important traditional approaches used for implementing asynchronous tasks in imperative and functional programming languages.

2.2.1 Calls and callbacks

The first approach is to use the callback functions as presented in the previous section. The idea behind callback functions is to have a function that represents an asynchronous operation accept a pointer to the function that is to be called when the asynchronous operation is finished. This is the reason why it is called a *callback* function – the programmer is scheduling the function to be called back by the system when the asynchronous operation ends. The result of the asynchronous operation, if it exists, can be passed in as the argument to the callback function.

In the following example, the `get_page` is the asynchronous function, and the `on_page_retrieved` is the callback. The `get_page` function is designed to follow the API of the Boost.Asio [Koh] library – a well-known callback-based library for asynchronous IO. It can easily be implemented by combining the Boost.Asio library's `async_read_until` which can be used to get the request headers (by reading until it reaches an empty line, which corresponds to the HTTP web server semantics); with the header retrieved, the program will know exactly how many bytes the response body has, and it will able to get the response body with the `async_read` function.

```
void on_page_retrieved(const response &result)
{
    std::cout << "Web page retrieved";

    if (result.is_valid()) {
        std::cout << "Success: " << result.body();
    } else {
        std::cout << "Error retrieving the page\n";
    }
}

void get_page()
{
    wget("http://www.math.rs/", on_page_retrieved);
    std::cout << "Sent the request\n";
}
```

It is important to note that the `wget` call will not block the execution of the rest of `get_page` function, so the user will see the `"Sent the request"` message before the actual request is fulfilled.

Apart from the need to separate the code that starts the asynchronous operation from the code that handles the result, the call-callback approach has another significant drawback – the same function call both starts the asynchronous operation and defines the handler which creates a tight coupling between the two. In our case, the `on_page_retrieved` and `get_page` functions are tightly coupled, and one can not really exist without the other which collides with the structured programming best practices [DDH72]. This can be partially remedied by converting the `on_page_retrieved` into a lambda which is directly passed to the `wget` function, but that only places the result handler *inside* the function that starts the asynchronous operation, but the logic is still artificially separated into two separate functions. This also leads to maintainability problems since changing any synchronous function to be asynchronous, and vice versa, would require changes in all the places where that function was called from.

2.2.2 Signals and slots

Another approach to dealing with asynchrony are signals and slots. Broadly speaking, the signals and slots system is a simple communication mechanism between different objects. A signal is used to emit a message, and a slot that is connected to said signal receives and processes the message [LP16, Chapter 1].

Note that we will be using the syntax for connecting the signals to their respective slots as used in the Qt library for C++, since it is the most prominent library for C++ that uses signals and slots mechanism, but we will keep using the `snake_case` syntax, which is not usual in Qt, to keep uniformity of the code examples.

Let's consider mouse events in an application. Buttons in the GUI are objects that send a `clicked` signal when the user clicks them.

```
button *exit_button;
button *get_page_button;

connect(exit_button, SIGNAL(clicked()),
        main_window, SLOT(quit()));
connect(get_page_button, SIGNAL(clicked()),
        main_window,     SLOT(get_page()));
```

When the `exit_button` is clicked, the connection will invoke the `quit` member function of the `main_window` object. In the same way, clicking on the `get_page_button`

will invoke the `get_page` member function. It is important to note that slot invocation can be synchronous (the slot function is invoked directly by the signal) or scheduled/asynchronous (the slot function is only scheduled to be invoked by the event processing loop when the signal is emitted).

Signals and slots allow easier decoupling of components than callbacks. Starting the asynchronous operation is separated from specifying the handler. The button does not know (nor it needs to) whether anybody is listening for its events – the signal is sent anyway.

While this mechanism is primarily meant for handling the user interface events, it can be as easily applied to any type of asynchronous operation. In the following example, we present a similar example of fetching the contents of a web page we had in the previous section:

```
void init()
{
    connect(page, SIGNAL(page_retrieved(response)),          ❶
            this, SLOT(on_page_retrieved(response)));
}

void get_page()
{
    page.get("http://www.math.rs/"); ❷
}

void on_page_retrieved(const response& result)
{
    std::cout << "Web page retrieved";

    if (result.is_valid()) {
        std::cout << "Success: " << result.body();
    } else {
        std::cout << "Error retrieving the page\n";
    }
}
```

In this implementation, the `get_page` member function only needs to know which page to request ❷. It does not need to care which function will process the result – the handler is defined separately ❶.

2.2.3 Actor systems

The third approach, popular mainly in the functional programming community, are the actor systems [HBS73]. They were popularised by the Erlang [Arm03] programming language, and a few of the more modern languages are adopting the idea (Scala's Akka library [MRO10], the C++ Actor Framework [Cha+13] and the Distributed Haskell [EBP11] to name a few).

Similar to the signals and slots mechanism, the software system is broken into a set of small components that are able to exchange messages. The difference is that in the actor model, these messages are the only mechanism of communication between separate components. All components are completely isolated from one another – there is no shared data, they can not access each other's memory space and similar. The system that uses the actor model needs to be designed as if all components are running on physically separated machines which communicate over the network. This also means that all messages are handled in asynchronous manner – there is no way for one component to directly call a function belonging to another component.

In the software systems based on the actor model, it is common for the sender to decide who will be the recipient for each message, and not the other way round. For example, an actor A can choose to send the message to the actor B. The actor B is not able to request that it wants to listen to all messages from A – it will receive all messages sent to it no matter who sent them.

The previous example of fetching a web page can be implemented using the actor model as follows (notation is as used in the C++ Actor Framework):

```
void get_site(event_based_actor *self, const actor &page_actor)
{
    self->async_send(page_actor, ❶
                     get("http://www.math.rs/")); ❷
}

behaviour receive(event_based_actor *self) {
    return { [=] (const response_t &result) { ❸
                if (result.is_valid()) {
                    …
                }
           } };
}
```

The `async_send` member function of an actor is used to send messages. In this case, the `get_site` function is sending a single `get` message ❷ to the actor identified by `page_actor` ❶. When the request is received and processed, that actor will send back a message of type `response_t` which is handled in the behaviour defined by the `receive` function ❸.

The main benefit of the actor model is that the software systems built on it are highly-horizontally scalable. The problem is that actors are still a very low-level primitive and share many of the drawbacks that the previous solutions had. They are not easily composable and they also require separating the entity that starts an asynchronous operation from the entity that handles the result.

2.3 C++ algorithms and ranges

2.3.1 Iterators and algorithms

The `<numeric>` and `<algorithm>` headers of the C++ standard library (commonly referred to as the STL – the Standard Template Library) provide a set of around 90 generic algorithms that operate on sequences. A sequence, as far as the C++ standard library is concerned, is denoted by a pair of iterators – one pointing to the first element, and one pointing just past the last element of the sequence. Special cases are the output sequences which are usually denoted only by a single iterator to the beginning of the sequence where the assumption is that the output sequence is holding a sufficient number of elements to store the result of the algorithm [Str13, Section 4.4].

We will cover a few of the more distinct algorithms that have different requirements on the input sequences and that return different result types. These algorithms and their differences will be useful later in this chapter.

2.3.1.1 Accumulate

The `std::accumulate` is a generic algorithm that encapsulates a simple pattern of processing sequences in a tail-recursive manner often called *folding* [Hut99]. It takes a collection (as a pair of iterators), the initial value for the accumulator, and a binary operation that produces a new accumulated value given the previously accumulated value and an element from the sequence. The algorithm processes the elements of the given sequence in order, from beginning to the end, and for each element calls the given binary operation on the previously accumulated value and the current

sequence element. The result of the algorithm is the last calculated accumulated value – the value after all elements have been processed.

This algorithm is useful for any sort of aggregation (summing a collection of numbers, concatenating a collection of strings, etc.) but it is much more generic. Since it encapsulates the general form of tail-recursion for sequence processing, it can be used to implement many recursive algorithms and, in fact, most of the algorithms in the C++ standard library can be implemented using `std::accumulate`, albeit not always with the same performance [Dea16].

As previously mentioned, the `accumulate` algorithm encapsulates tail recursion. Since the C++ standard does not require the compilers to perform the tail-recursion optimization even if most of them do [Čuk18, Chapter 2], we can convert the recursion into an iterative loop manually like so[1]:

```
template <typename BeginIt, typename EndIt,
          typename Acc, typename F>
Acc accumulate(BeginIt begin, const EndIt& end,
               Acc acc, BinaryOp op)
{
    for (; begin != end; ++begin) {
        acc = std::invoke(op, std::move(acc), *begin);
    }

    return acc;
}
```

From this implementation, we can deduce the prerequisites for the iterators passed to the algorithm. It is necessary to be able to check whether two iterators are different (representing different positions in the collection), and it is necessary to be able to traverse the sequence once by incrementing the `begin` iterator in each iteration of the `for` loop.

2.3.1.2 Transform

The `std::accumulate` is in the class of algorithms that read the whole sequence to return only a single value as the result. The second class of algorithms, best represented by the `std::transform` algorithm, returns a whole sequence of items,

[1]The algorithm implementations in this chapter are based on the implementations provided by GNU's `libstdc++`, version 7.2, with the types of begin and end iterators separated to support sentinel types [NPS14]

and the resulting items are generated gradually while the input sequence is being read. Due to the design of the standard library, the resulting items are not returned as the function result, but are written to the output collection denoted by the output iterator passed to the algorithm.

For example, we can use `std::transform` to print out string representations of objects in a collection by specifying the destination sequence to be an output stream iterator:

```
std::transform(begin(people), end(people),
               std::ostream_iterator<std::string>(std::cout, "\n"),
               [] (const auto& value) {
                   using std::to_string;
                   return to_string(value);
               });
```

This code will start printing the values to the standard output even before the whole input sequence is processed.

The implementation of the `std::transform` algorithm is quite straight-forward. It gets an input sequence as a pair of iterators, the beginning of the output sequence, and the transformation operation. It iterates over the input sequence, and for each of the values it encounters in the input, it writes the transformed value to the destination sequence.

```
template <typename BeginIt, typename EndIt,
          typename OutputIt, typename Operation>
void transform(BeginIt begin, const EndIt& end,
               OutputIt output, Operation op)
{
    for (; begin != end; ++begin, ++output) {
        *output = std::invoke(op, *begin);
    }
}
```

From the implementation, we can see that, the iterator that represents the start of the input sequence needs to be incrementable and equality-testable with the iterator that represents the end of the input sequence, as it was the case with `std::accumulate`. For the output sequence, it is just necessary to be able to move from one element to the next with the assumption that there are enough elements

in the sequence so that the algorithm does not need to check whether the end of the output sequence is reached.

Usually, having *out* function arguments is a signal of bad API design. But, in this case, it is a bit more complicated. The output iterator is not strictly an *out* argument – it is also an *in* argument. Namely, the input is the *preallocated* sequence that the algorithm can write the data to. If a new sequence was to be returned as the result of the `transform` algorithm, the algorithm would need to allocate memory for that sequence, and return it only when all the values from the input sequence have been processed. Having the *in-out* parameter in this case gives the programmer the complete control over how the resulting sequence should be stored – which data structure it should be, whether it should be a data structure at all, or some special iterator like the `std::ostream_iterator`. It also allows overwriting the input sequence with the transformed data if the original values are not needed anymore.

2.3.1.3 Partition and sort

The next class of algorithms are the algorithms that modify the sequence in-place like partitioning and sorting (both the stable and unstable variants of said algorithms). This means that the source sequence is both the input and the output of the algorithm.

Partitioning is a simple algorithm that takes a sequence of elements (as a pair of iterators), a predicate function, and reorders the elements in the sequence so that all elements that satisfy the predicate function come before the elements that do not. The algorithm also returns the partition point as the function result:

```
template <typename BeginIt,
          typename EndIt,
          typename Predicate>
BeginIt partition(BeginIt first, EndIt last, Predicate pred)
{
    if (first == last) return first;

    while (std::invoke(pred, *first)) {
        if (++first == last) {
            return first;
        }
    }
```

```
    BeginIt next = first; ❶

    while (++next != last) {
        if (std::invoke(pred, *next)) {
            std::iter_swap(first, last);
            ++first;
        }
    }

    return first;
}
```

This algorithm imposes all the same requirements to the iterators as the previous ones – it needs to be able to check whether two iterators are equal or not, and it needs to be able to increment the iterators.

But there is an additional requirement which might not be as obvious. The algorithm is creating a copy of the iterator ❶ which means that, while it still only goes from one item to the next, it needs to be able to traverse the sequence multiple times – both `first` and `next` iterators can traverse the sequence independently from one another. This means that the partition algorithm can only work on proper collections, it can not work on sequences like input streams which forget values as soon as they are read.

While these requirements on iterators are *sufficient* to implement the partitioning algorithm, increasing the requirements by allowing the iterator to also be decremented would give us a more efficient version of the partitioning algorithm:

```
template <typename BeginIt,
          typename EndIt,
          typename Predicate>
BeginIt partition(BeginIt first, EndIt last, Predicate pred)
{
    while (true) {
        while (true) {
            if (first == last) {
                return first;
            } else if (std::invoke(pred, *first)) {
                ++first;
```

```
            } else {
                break;
            }
        }

        --last;
        while (true) {
            if (first == last) {
                 return first;
            } else if (!bool(std::invoke(pred, *last))) {
                 --last;
            } else {
                 break;
            }
        }

        std::iter_swap(first, last);
        ++first;
    }
}
```

2.3.1.4 Iterator types

So far, we have seen three different sets of requirements algorithms impose on the iterators passed to them. All iterators need to have the equality operators defined on them, and they need to be incrementable. Iterators that satisfy these requirements belong to the class of *input* iterators. Adding the requirement that the iterator can traverse the sequence multiple times gives us the class of *forward* iterators, and adding the decrement operator to this class gives us *bidirectional* iterators.

For some algorithms, this is still not sufficient. In the C++ standard library, due to the algorithm complexity requirements for the sorting algorithm [ISO17, alg.sort], it needs to be implemented using a hybrid sorting algorithm such as *introsort* [Mus97]. All of these algorithms impose additional requirements on the iterators – iterators need to be random-access – sequential bidirectional access is not sufficient. This gives four categories of iterators and four different classes of sequences that can be passed to the C++ standard library algorithms as input.

2.3.1.5 Problems with iterators

While designing the initial version of the C++ standard library, Alexander Stepanov focussed on defining the algorithms in a way that would allow him to implement the stable sort algorithm by using the previously defined algorithms [Par17]. For this purpose, having algorithms that operate on iterators instead of working directly with sequences is perfect. It was possible to run an algorithm on a part of a sequence instead of running it over the whole sequence.

For example, in order to implement a simple quicksort algorithm, we could use the `std::partition` algorithm to move items that are less than the pivot element to the beginning of the collection, and items that are greater or equal to the pivot to the end. The partition algorithm will return the partition point – effectively returning two sub-sequences that the quicksort algorithm should be recursively called on. If sorting worked on collections instead on iterators, it would be necessary to create intermediary temporary collections which would increase space complexity of the algorithm, and reduce the runtime performance.

While iterators make the C++ standard library algorithms truly generic, they do have significant drawbacks with regard to code correctness. Since the algorithms know nothing about the data structures the iterators belong to, they can not guard against some common programming errors like overflowing the output sequence, or accessing data that does not belong to a single sequence:

```
std::copy(begin(source), end(source),
          begin(destination));
```

Will overflow if the source sequence is longer than the destination

```
std::sort(begin(source),
          end(destination));
```

The algorithm does not know if the provided iterators belong to two different sequences

Another important problem is that having the standard library algorithms accept iterators as separate function arguments means that one algorithm can not be easily invoked on the sequence created by another algorithm.

Consider the following scenario – we want to calculate the sum of squares of elements in a sequence without using a ready-made algorithm like `std::inner_product`, but only by composing simpler C++ standard library algorithms. In Haskell[2], this could be implemented like so:

```
sum $ map (\x -> x * x) xs
```

[2]Haskell Language `http://www.haskell.org`

This can be implemented in C++ using the `std::transform` algorithm to square all the numbers in a given sequence, and then use the `std::accumulate` algorithm to sum them. The problem is that we need to create a temporary collection to store the results of the `std::transform` algorithm:

```
template <typename T>
auto sum_squares(const std::vector<T>& xs)
{
    std::vector<T> squares(xs.length());
    std::transform(begin(xs), end(xs),
                   begin(squares),
                   [] (auto x) { return x * x; });

    return std::accumulate(begin(squares), end(squares), 0);
}
```

This is not as readable as the Haskell version, it is less generic (it works only on vectors) and it is less efficient because it needs to allocate additional memory. This would be much simpler and more efficient if it was implemented using a simple for-loop.

2.3.1.6 Proxy iterators

If we skip the complexity of the syntax in the previous example, the main problem is the need to create the temporary vector to store the squared numbers.

Is it really necessary? The `std::accumulate` algorithm expects an input iterator, that is, an object that can be dereferenced, incremented, and compared against the end iterator.

We can create such an object that behaves same as a normal iterator, only with a different dereferencing operator. The dereferencing operator will not return the value from the given vector directly – it will return the transformed value [Čuk18, Chapter 7].

```
template <typename Iter, typename Func>
class transform_iterator {
public:
    transform_iterator(Iter iter, Func func)
        : m_iterator(iter)
```

```
        , m_func(func)
    {
    }

    auto operator*() const                          // Dereferencing the transform_iterator will transform the value before returning it
    {
        return std::invoke(m_func, *m_iterator);
    }

    template <typename OtherIter>                   // The equality operator just tests whether the passed iterator is the same as the internal iterator
    bool operator==(const OtherIter& other) const
    {
        return m_iterator == other;
    }

    … // iterator traits and other member functions
private:
    Iter m_iterator;
    Func m_func;
};

template <typename Iter, typename Func>
auto transform(Iter&& iter, Func&& func)
{
    return transform_iterator(std::forward<Iter>(iter),
                              std::forward<Func>(func));
}
```

The `transform_iterator` can be used with the `std::accumulate` algorithm without the need for creating an intermediary vector to hold the results:

```
template <typename Col>
auto sum_squares(const Col& xs)
{
    return std::accumulate(
        transform(begin(xs), [] (auto x) { return x * x; }),
        end(xs), 0);
}
```

Apart from improving syntax and efficiency, this solution is also more generic as it works on any sequence, and not only on vectors.

2.3.2 Ranges

One commonality in all the previous examples was that the used algorithms accepted iterator pairs to denote input sequences – one iterator that points to the beginning, and another that points to the end.

The end iterator is special. In most of the C++ standard library algorithms, the end iterator is only used to check whether the algorithm has reached the end of the sequence. It is never dereferenced (as it would result in an *undefined behaviour*) and it is rarely changed (like we did in the second implementation of the partitioning algorithm). This means that, in most cases, the algorithms do not need the end of a sequence to be denoted by an iterator at all. It can be any object for which we can check whether a given iterator is equal to it or not. Objects that serve just to denote the end of a sequence are usually called *sentinels*.

Therefore, the abstraction over different types of sequences can internally hold just one iterator that denotes the start of the sequence, and a sentinel denoting the end of the sequence. This abstraction is called a *range*.

Just like there are different iterator types, there will be different range types depending on the type of the contained begin iterator. We will focus here on input ranges, and algorithms that can work on them. Input ranges consist of an input iterator – iterator that can be dereferenced and incremented, and a sentinel denoting the end of the range. Since the sentinel is just a dummy object that is used to compare the iterator against when the algorithm needs to know whether it has reached the end of the range, it is not known where the end of an input range is until all the items in the range are traversed. This is similar to a basic implementation of a singly linked list without any optimizations – the implementation which keeps only the list head, and where each node only keeps a pointer to the next element. For such a list, we can not know how many elements it contains, where it ends, etc. until we traverse the whole list. The main difference compared to ranges is that we can traverse a list multiple times, which does not hold for input ranges in general.

With ranges defined like this, we can get much better syntax than iterators gave us, without loss of expressiveness. Instead of algorithms accepting pairs of iterators, they can simply accept a range, and they can also return new ranges as the result.

With ranges, the previous example becomes much more readable:

```
template <typename Col>
auto sum_squares(const Col& xs)
{
    return accumulate(
        transform(xs, [] (auto x) { return x * x; }),
        0);
}
```

This variant of the transform algorithm does not actually perform any transformations on the original sequence, it just returns a new range consisting of a proxy iterator similar to the `tansform_iterator` presented earlier, and a sentinel which tests whether the end of the original sequence has been reached. The element transformations will occur only when the `accumulate` algorithm tries to access the elements of the transformed range in order to sum them.

In a similar manner, we can create ranges that define filtering – by creating a proxy iterator that skips over values that do not satisfy a predicate, grouping – with a proxy iterator that groups all equal elements in sub-ranges, etc.

2.4 Some important C++ features and techniques

2.4.1 Templates

C++ templates [Str13, p. 23.1] allow to create generic definitions of types and functions. It is done by factoring out the common behaviour of the things we want to implement and pass the specifics through the template parameters. This is useful for defining generic collections such as `std::vector` or `std::list` that are able to hold any proper data type, or functions like `std::sort` that can operate on any random-accessible collection.

Templates in C++ are conceptually different than corresponding features in other languages like Java or C#. In Java and C#, generics are just syntax sugar that does not change the semantics of the language – it generates the same code as if the `Object` class was used with a lot of casting down to the needed type – the same bytecode handles both the lists of integers and lists of strings. The template code in C++ does not end up in the compiled code, only specific instantiations for specified arguments do. This means that during compilation, C++ creates separate class

implementations for lists of integers and lists of strings, and separate functions for sorting a vector of integers and an array of bytes. That results in longer compilation times, and increased size of the compiled code, but no runtime performance penalties at all – each of the instantiations is compiled and optimized to run for the specified arguments.

2.4.2 Template Meta-Programming (TMP)

TMP is a meta-programming technique in which templates are used by the compiler to generate code. It has been shown that the templating mechanism in C++ is a Turing-complete language [Cza02], where programs written in it are executed during program compilation.

The main technique for metaprogramming in C++ is based on partial evaluation of templates. We will demonstrate this with a usual example of calculating a factorial at compile-time. TMP for the most part follows a functional programming paradigm. So it is a good practise first to define the function we want to implement in the functional style. In Haskell, it would be as follows:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The C++ TMP equivalent, albeit more verbose, would follow the same logic:

```
template <int n>
struct factorial
{
    enum {
        value = n * factorial<n - 1>::value | The general case
    };
};

template <>
struct factorial<0>
{
    enum {
        value = 1 | Special case - when n = 0
    };
};
```

```
int main(int argc, char** argv)
{
    int array[factorial<5>::value];
}
```

We can use the factorial meta-function in the places where the compiler expects a constant expression

The power of TMP is not in implementing this kind of meta-programs, but rather in the fact that it allows performing calculations and executing (meta) programs at compile-time – it offers a programmable control over the compilation process, including checking the compiler states and characteristics of the generated types.

2.4.2.1 Transforming types with meta-functions

The main use of TMP is for type transformation. When we define a template class or a template `using` declaration, we are creating a *meta-function* that returns a new type when we define its parameters.

For example, if we have a sequence type that can be iterated over, we can create a meta-function that returns us the type of the values in that sequence like so:

```
template <typename T>
using value_type_t = decltype(*begin(std::declval<T>()));
```

For a given type `T`, this checks what is the type of the expression inside the `decltype` statement. The expression dereferences the begin iterator of a value of type `T`, which will be the type of the element contained in the collection of type `T`.

Alternatively, since all generic collection types in the C++ standard library (and many other third party C++ libraries) have a nested `value_type` type definition [Čuk18, Chapter 11], we can implement the `value_type_t` meta-function in a different way:

```
template <typename T>
using value_type_t = typename T::value_type;
```

This is a meta-function that, given a type `T`, returns the type specified by the nested `value_type` definition.

2.4.2.2 SFINAE

Substitution Failure Is Not An Error (SFINAE) is a rule of C++ where an invalid substitution of template arguments does not constitute an error [VJ03]. The compiler just keeps going until a viable substitution is found. An error is reported only if there is no valid substitution to be found. It is one of the more advanced TMP techniques.

For a demonstration of this principle, the following code implements a checker whether a given type `T` has a nested type definition called `inner`:

```
template <typename T>
struct has_typedef_inner {
    template <typename C>
    static std::true_type& test(typename C::inner*);

    template <typename>
    static std::false_type& test(...);

    // Value is a static constant member
    // that will be true iff the test<T>'s
    // result is std::true_type
    static const bool value =
        sizeof(test<T>(0)) == sizeof(std::true_type);
};

struct i_have_inner {
    typedef float inner;
};

has_typedef_inner<int>::value; // false
has_typedef_inner<i_have_inner>::value; // true
```

When the compiler encounters the `has_typedef_inner<SomeType>::value` statement, it replaces the template argument with `SomeType` and calculates the member `value`. For that, it needs to deduce the result type of `test<T>(0)`. It does so by performing pattern matching. If it succeeds to match against the first implementation (if `SomeType` has the specified inner type), the result type will be `std::true_type`. Otherwise, it will match against the second one, whose result

type is `std::false_type`. We will need tests like this one to be able to differentiate between different types of tasks.

2.4.3 Variadic templates

The C++11 standard lifted the requirement of the previous version of the standard that templates must have a fixed number of arguments [Str13, p. 28.6]. Similar to the other TMP techniques, implementing templates with variable number of arguments called *variadic templates* has a functional base. In this case, it takes the usual form of list processing (again, we are using Haskell for the functional notation):

```
doSomething :: [A] -> SomeType
doSomething []     = resultForTheEmptyList
doSomething (x:xs) = resultForTheHeadAndTail(x, xs)
```

Let us demonstrate this on a simple example of defining a `tuple` class. We start by defining an empty tuple (0-tuple). Afterwards, we inductively define a n-tuple from $(n-1)$-tuple:

```
template <typename ...Types>          The variadic
class tuple;                          template definition

template <>                           Specialization for the
class tuple<> {};                     0-tuple

template <typename Head, typename ...Tail>
class tuple<Head, Tail...>            Inductive step - n-tuple privately
    : private tuple<Tail...> {        inherits from (n − 1)-tuple

protected:
    Head m_head;

    // …
};

// Now we can define any tuple we want
tuple<int, double> pair;
tuple<int, int, int> triple;
tuple<int, int, bool, std::string> quadruple;
```

This implementation can be seen as a recursive definition for a tuple. An $n+1$-tuple is a product type of a n-tuple (which we privately inherited from) and a value of a new type (the `m_head` member variable).

2.4.4 Move semantics

One of the notable features of C++ is the automatic clean-up that happens every time an object goes out of the scope. When it happens, the destructor for that object is called, and it frees up all the resources that object acquired [Mey97]. This allows C++ programs to be implemented without the need for garbage collection while, at the same time, not being forced to manually manage the lifetime of all resources (memory, file descriptors, etc.).

While statements like `std::vector<int> *v = new std::vector<int>();` are common in other contemporary languages, and are valid in C++ as well, they are not often present in C++ code. Instead of using a pointer to a vector and allocating the vector dynamically, it is customary to just declare a value of type `std::vector`, and rely on automatic destruction when the vector goes out of the scope. Preferring values to pointers works great when the program safety and resource leakage are concerned, but it had a few significant downsides before C++11.

Consider the following example. We have a collection of values, and we want to print them out grouped by some member variable.

```
template <typename T, typename M, typename C>
std::vector<T> filter_on_member(const std::vector<T>& xs, M member,
                                const C& value);


template <typename T, typename M, typename C>
void print_categorized(const std::vector<T>& xs, M member,
                       const std::vector<C>& categories)
{
    std::vector<T> filtered;

    for (auto category: categories) {
        filtered = filter_on_member(xs, member, category);
        for (const auto& x: filtered)
            std::cout << x << std::endl;
    }
}
```

The `filter_on_member` function will return a temporary vector, which will be assigned to the `filtered` variable. The assignment operator will copy the data from the temporary vector, and then the temporary vector will be destroyed. And this will happen for each category.

To recap, for each category, we just want to store the result of `filter_on_member` function into the `filtered` variable, yet we will:

- Create a temporary vector containing the result of `filter_on_member`;
- Copy the data from the temporary vector to the `filtered` vector;
- Destroy the temporary vector.

Copying the data is expensive. The only reason why we would need to copy the data is if someone else wanted to use the original vector. But since it is temporary, we know that it will not be used by anyone else. So, instead of copying, we could just *steal* the data from the temporary vector and move it into the `filtered` vector.

2.4.4.1 Lvalue and rvalue references

In C++11, we got the ability to tell whether something is a temporary value or not. The values are split into two categories – lvalues and rvalues, and they can be captured by lvalue-references and rvalue-references respectively. One way to deduce whether a value is an lvalue, or an rvalues is to think of whether it can be on the left side of the assignment operator or not. If it can be on the left side, it is an lvalue and it is not a temporary; while if it is a temporary, we can not assign something to it, and it can only be on the right side of the assignment operator, so it is an rvalue. The actual rules are more complex than this, but this is the main gist.

Lvalue-references are denoted by a single ampersand `&`, whereas rvalue-references are denoted by the double ampersand `&&`. We can use these to detect whether a value is temporary or not. Let's demonstrate this using the variables and functions from the previous code snippet:

```
std::vector<T>& lvalue_ref = filtered;  OK
std::vector<T>&& rvalue_ref = filtered; Error
```

`filtered` is not a temporary, it does not bind to rvalue refs.

```
std::vector<T>& lvalue_ref =  Error
    filter_on_member(…);
std::vector<T>&& rvalue_ref = Ok
    filter_on_member(…);
```

A result of a function is temporary vector which can not be bound to an lvalue ref., but it can be bound to an rvalue ref.

```
const std::vector<T>& const_ref = OK
    filter_on_member(…);
```

A special rule allows binding temporaries to const-refs which prolongs the lifetime of the temporary object

We can use different reference types to overload functions, and provide one implementation that works on temporaries, and one that works on normal values. For example, consider the `std::vector`. Usually, the vector contains a pointer to a dynamically allocated buffer that contains the data, along with some bookkeeping data.

If we are constructing a vector out of another vector, it would be beneficial to know whether that vector is a temporary one or not. If it is not, we need to copy all the data from the original vector, and if it is, we can just *move* the data into the new vector – leaving the original vector empty.

```
template <typename T>
class vector {
public:
    vector(const vector& other)
        : m_count(other.m_count)
        , m_allocated(other.m_allocated)
        , m_data(new T[other.m_allocated])
    {
        std::copy(…);
    }

    vector(vector&& other)
        : m_count(other.m_count)
        , m_allocated(other.m_allocated)
        , m_data(other.m_data)
    {
        other.m_data = nullptr;
    }

    ~vector()
    {
        delete [] m_data;
    }
```

The copy constructor needs to allocate a new buffer and copy all the data from the original vector into said buffer

The move constructor can just take the data from the original vector leaving the original vector empty

```
private:
    int m_count;
    int m_data;
    T* m_data;
};
```

2.4.4.2 Perfect forwarding

If we need to write a function that wraps in another function we will have the problem of how to pass the arguments from the wrapper function to the function we need to call.

We can pass the arguments *by-value* which can potentially incur performance degradation due to unnecessary copying, and can lead to compilation errors if the wrapped function expects a reference to an object because it wants to change the original object.

The second option is to pass a reference to the object. This would solve the above problems – no unneeded copying and the called function can access and change the original object through the reference. But we will get new issues – it will not be possible to pass temporaries to the function, nor objects declared as const because a non-const reference can not be bound to them.

The third option is to use a const reference, which will work in most cases optimally. The only case when a const reference can not be used is when the wrapped function needs to change the original object. In this case using a const reference will produce a compiler error.

This is solved in C++11 with the forwarding references (formerly known as *universal references* [Mey15]). A forwarding reference is written as a *double ampersand* on a automatically deduced type (template parameter or on `auto`). In the following code, the `fwd` argument is a forwarding reference to type `T`, while `value` is not (it is a normal rvalue reference because it is not a reference on an automatically deduced type, but on a concrete type – `int`).

```
template <typename T>
void f(T&& fwd, int&& value)
```

The forwarding reference allows us to accept both const and non-const objects, and temporaries. When passing the forwarding reference to the wrapped function, we need to use the `std::forward` helper function.

```
template <typename T>
void f(T&& fwd, int&& value)
{
    g(std::forward<T>(fwd));
}
```

This takes care of passing arguments to the wrapped function, but it does not deal with the returned value. We have the same options for the function result – return it as a value, reference or a const reference – and all of these can be problematic in different situations. The solution is to also perfectly forward the result of the wrapped function to the caller of the wrapper function. This can be achieved by using `decltype(auto)` as the specification of the return type for the wrapper function:

```
template <typename T>
decltype(auto) f(T&& fwd, int&& value)
{
    return g(std::forward<T>(fwd));
}
```

2.4.5 Lambda expressions

Lambda expressions in C++ [Str13, p. 11.4] provide a concise way to create simple function objects. Evaluation of a lambda expression results in a closure object. The closure object may capture the variables from the enclosing scope that are used in the lambda body.

```
int i = 4;

auto multiply_by_i =
    [=] (int number) {
        return number * i;
    }

multiply_by_i(5);
```

A lambda that multiplies its argument with a value from the surrounding scope

invokes the lambda

The square-brackets section of the lambda declaration is called a *lambda head* and it defines which variables from the outer scope should be captured inside of

the closure object generated by evaluating the lambda expression [Čuk18, Chapter 3], and in which manner (by value or by reference). In this case, we captured all the used variables by value. If we wanted the compiler to ensure that we want to capture only the variable `i`, we could have simply written the capture as `[i]`.

Lambdas, like other functor objects, are most useful as arguments of algorithm methods such as `std::count_if`. The following example returns the number of positive elements in a vector of integers:

```
std::count_if(
    numbers.cbegin(),
    numbers.cend(),
    [] (int number) { return number >= 0; }
);
```

2.4.6 Pipe call syntax

We have seen the ranges abstraction over different structures that implement sequences in C++ in Section 2.3.2, and we have seen how they can be helpful when defining composite transformations on sequences.

The problem with the previously shown syntax is that, like regular function application, the first transformation (read left-to-right) is the last to be performed. This is a common idiom in functional programming, but not in the procedural and object-oriented programming languages. In OOP languages, syntax like this would be preferred:

```
xs.filter(some_predicate)
  .transform(transformation_function)
  .group(some_operator);
```

Unfortunately, in C++, this would require that all the types that need to behave like ranges have to implement `filter`, `transform` and similar member functions which would destroy the point of ranges being a common abstraction over different data structures.

There is a proposal for extending the C++ language core to allow free-standing functions like `f(x,y)` to be called with `x.f(y)` syntax similar to the Wafl programming language [Mal10], but this proposal has not yet been accepted for inclusion into the C++ standard. Instead, if we want to provide a syntax that allows us to define a chain of range transformations from left to right, we will need to implement it manually by creating a small domain-specific language.

It is common to use the *pipe* operator for transformation chaining [Sch11] in libraries that implement ranges, much like it is used in most UNIX shells.

```
xs | filter(some_predicate)
   | transform(transformation_function)
   | group(some_operator);
```

If we have the `transform` function which accepts a range and a transformation function, we can easily (though with some boilerplate) implement the support for the pipe syntax for it. We just need to create a unary `transform` function which just takes the transformation function and returns a dummy object that will perform the transformation when connected to a range via the pipe operator:

```
template <typename F>
struct transformation_helper {
    F function;
};

template <typename Rng, typename F>
decltype(auto) operator| (Rng&& range,
                          transformation_helper<F> helper)
{
    return transform(std::forward<Rng>(range),
                     helper.function);
}

template <typename F>
auto transform(F&& function)
{
    return transformation_helper<F>{
            std::forward<F>(function)
        };
}
```

The pipe operator calls the binary transform func.

The unary transform function just creates an object that defines the transformation

The implementation would be the same for other range transformation functions. We will assume from now on that all functions that we use can be called both with regular and the pipe-based syntax.

2.4.7 Function objects and callables

We have seen several higher-order functions in the previous sections. Every time we want to create a function that can accept another function as its argument, we defined it as a function template, where one of the template parameters is the function because C++ does not provide a common type over all different types of functions.

We have a couple of different ways to define functions. One approach is to use duck-typing and to say that anything that we can call with the common function call syntax is a function. In C++, this includes the following:

- Free-standing functions (C functions)
- Pointers and references to free-standing functions
- Objects that have the call operator implemented on them (`operator()`)
- Objects that can be cast to a function pointer

```
void f();                   An ordinary free-standing function

auto f_ptr = &f;            A function pointer
auto f_ref = *f_ptr;        and reference

class f_obj_t {
public:
    void operator() () const;   The call operator defined in a class
                                allows instances of that class to
                                be called like normal functions
};

f_obj_t f_obj;

auto f_lambda = [] () {     A lambda that does not capture anything
    …                       has the casting operator to a function pointer
};
```

We will expand this with pointers to member functions and pointers to member variables. While these are not traditionally considered to be functions, member functions behave mostly the same as free-standing functions with an implicit argument pointing to the object instance whose member function or variable we want

to access (`this`). While these can not be called with the normal function call syntax, we can use the `std::invoke` helper function to call both normal functions and pointers to member functions and variables.

```
std::invoke(&std::string::empty, s);   calls s.empty()
std::invoke(&std::toupper, c);         calls toupper(c)
```

The `std::invoke` helper function introduces boilerplate into every function call, but it is a necessity if we want to make our code as generic as possible. It has no use in a non-generic context.

2.5 Introduction to category theory

Most of the concepts used in this dissertation come from functional programming where the main building blocks are functions. We will now cover a few basic concepts of the algebra of abstract functions – the category theory.

The category theory defines the behaviour that function-like objects should have outside of the constraints of set theory. While most of the notation is shared with the functions from set theory, it is important to remember that this is about abstracting functions, not about defining relations between sets. For this reason, instead of using the term *function*, in the rest of this section, we will be using the term *arrow*, as it is common in the category theory.

Let's start with a simple notion of a *metagraph*. A *metagraph* consists of [Lan98] objects $a, b, c, \ldots$, arrows $f, g, h, \ldots$, and two operations:

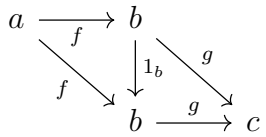
- Domain, which assigns an object $a = \operatorname{dom} f$ to each arrow f;
- Codomain, which assigns an object $b = \operatorname{cod} f$ to each arrow f.

We can introduce shorter notation for specifying the that a and b are the domain and codomain (in that order) of an arrow f like so:

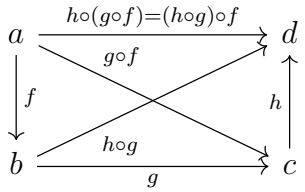
$$f : a \to b$$

Definition 2.1. A *category* is a metagraph with two additional operations:

- Identity, which assigns an arrow $\mathrm{id}_b = 1_b : b \to b$ to each object b where 1_b satisfies the *unit rule* – $1_b \circ f = f$ and $g \circ 1_b = g$ for any $f : a \to b$ and $g : b \to c$. The unit rule can be represented graphically as:



- Composition, which assigns an arrow $g \circ f$ to each pair of arrows f and g with $\operatorname{dom} g = \operatorname{cod} f$. $g \circ f$ is called *composition* of arrows g and f with $g \circ f : \operatorname{dom} f \to \operatorname{cod} g$. Composition needs to be associative, meaning $h \circ (g \circ f) = (h \circ g) \circ f$, or graphically:



It is important to note that a category is anything that satisfies this definition. Sets and functions on sets are just one special category (the `Set` category) where sets are the objects, functions are arrows, and the usual identity function and function composition are used. Another example is a category of all groups – objects are the groups G, H, etc. and arrows are all homomorhpisms from G to H.

2.5.1 Category of C++ types

For us, the most interesting category is the `Type` category. The objects in this category are C++ types (including the instantiated class templates), and arrows are unary C++11 functions (including the instantiated function templates) and other callables (see Section 2.4.7). This category can be seen as a special case of the `Set` category because a C++ type can be seen as a set of all different values of said type.

The identity operation is a function that returns the same value it gets as the argument:

```
template <typename T>
decltype(auto) identity(T &&value)
{
    return std::forward<T>(value);
}
```

The `identity` is a generic function that will be instantiated for any type `T` it is invoked for, and it will return a value of the same type it gets as the function argument. It will correctly handle references and `const`-qualified types (see 2.4.4.2).

The composition operation is a bit more involved. We need to create a function template that takes two functions as arguments, and returns a composed function. The easiest approach is to use a lambda for this:

```
template <typename G, typename F>
auto compose(F&& g, G&& f)
{
    return [g, f] (auto&& value) {
        return std::invoke(g,
                    std::invoke(f,
                        std::forward<decltype(value)>(value)));
    }
}
```

The `compose` function creates a closure which will contain a copy of the functions that need to be composed. When this closure is called, it will return the value it receives as the argument with `f` and `g` applied to it.

Note that this implementation of the `compose` function makes copies of callables `f` and `g`. This approach is idiomatic to the C++ standard library, but it can induce performance issues if used with callables that are expensive to copy. Copying can be avoided either by passing reference wrappers (by using `std::ref`) to the objects that are expensive to copy, or by modifying the `compose` function to use perfect forwarding for lambda captures (see [Rom]).

It would be easy to show that the `identity` function satisfies the *unit law*, and that the `compose` function is associative, which makes this a category.

2.5.2 Functors

Anything that obeys the category laws is a category – it is just necessary to define what objects and arrows are, and to define the identity and composition operations. Let's make a category where other categories are objects, and where arrows are mappings between them.

Definition 2.2. A *functor* F between categories C and D is [Awo06] a mapping of objects to objects, and arrows to arrows which satisfies the following rules:

- $F(f) : F(\mathrm{dom}\, f) \to F(\mathrm{cod}\, f)$ for any arrow f in category C
- $F(g \circ f) = F(g) \circ F(f)$
- $F(1_a) = 1_{F(a)}$

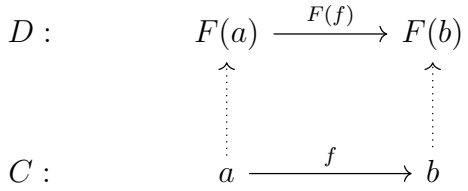


Figure 2.4: Graphical representation of the relation between arrows and objects of categories C and D for a functor $F : C \to D$.

It would be easy to show that functors compose in the right way, and that every category has an identity functor, which gives us another example of a category – the category of all categories in which arrows are functors.

While the definition of a functor seems rather abstract, functors appear everywhere in real-world programming. We will investigate functors that operate on the `Type` category – functors that map types to other types, or in other words, *endofunctors* on the `Type` category.

In order to create something that can operate on all different types in C++, it is necessary to use templates. And since endofunctors on the `Type` category map types to types, the obvious choice for implementing a functor in C++ is to create a class template.

```
template <typename T>
class some_functor {
    …
};
```

The `some_functor` class template takes one argument – any C++ type `T` – and returns a new type `some_functor<T>`.

This is only one half of a functor – the `some_functor` class template transforms types to types. It is also necessary to create a transformation that converts ordinary C++ functions that operate on types from the `Type` category to functions that work on the types with `some_functor` applied to them. Specifically, it needs a lifting transformation `lift_to_some_functor` that, given a function $f : T \to R$ (for any pair of types T and R) returns a function:

$$lift\_to\_some\_functor(f) : some\_functor\langle T\rangle \to some\_functor\langle R\rangle$$

From now on, when we talk about functors, we will mostly talk about endofunctors on the `Type` category, and a functor will be consisted of a class template with a single template parameter, and a corresponding lifting function.

We are now going to cover a few examples of functors that are a part of the C++ standard library, or proposed for inclusion into the standard library in C++20.

2.5.2.1 Vectors and other sequences

The `std::vector` is main type commonly used to demonstrate class templates [Str13, Section 3.4.2]. The internal implementation of a vector is not relevant for this discussion, the only important thing is that a `std::vector<T>` is a sequence of values of the same type `T`. This means that, everything in this section also applies to other similar collections like linked lists and double-ended queues.

The `std::vector` is a class template, and it only needs a lifting function defined for it in order for it to become a functor.

Note that we are going to relax the functor definition a bit. By definition, a functor needs to be able to transform all the types from the `Type` category, which `std::vector` and similar class templates can not do. Namely, we are not allowed to create a vector of reference types which also belong to the `Type` category. While this is a serious limitation from the theoretical standpoint, it is not relevant in practice.

The C++ standard library provides the `std::transform` algorithm which takes a collection of values of type `T`, a transformation function which maps values of type `T` to values of (possibly same) type `R`, and it returns a collection of transformed values.

```
std::vector<std::string> words = { … };
std::vector<int> word_lengths(words.length());

std::transform(std::cbegin(words), std::cend(words), ❶
               std::begin(word_lengths), ❷
               transformation_function);
```

The `std::transform` algorithm lifts the `transformation_function` and applies it to the source collection [Čuk18, Chapter 1]. While conceptually the same, the `std::transform` is not strictly a lifting transformation. It has a few problems:

- The lifting transformation should only accept a function that needs to be lifted, and return a new function that operates on `std::vector<T>`;
- `std::transform` does not accept a collection to be transformed, but a pair of iterators ❶;
- and it does not return a new transformed collection – it accepts the pre-created collection it writes the transformed values to as its argument ❷.

This can easily be remedied by creating a wrapper function `lift_to_vector` which has only the transformation function as its parameter, and it returns a lambda that transforms the given vector:

```
template <typename F>
auto lift_to_vector(F&& transformation_function)
{
    return [transformation_function] (const auto& values) {
        std::vector<decltype(transformation_function(values[0]))>
            result(values.length());

        std::transform(std::cbegin(values), std::cend(values),
                       std::begin(result),
                       transformation_function);

        return result;
    };
}
```

The `lift_to_vector` function is a proper lifting function for the `std::vector` functor. The important thing to note is that it just wraps the `std::transform` algorithm to fit the functor definition, it does not do anything that said algorithm can not. This implies that in context of the C++ programming language, anything that can be used with the `std::transform` algorithm can also represent a functor.

In the context of C++, a functor consists of a template type (like `std::vector`) and a lifting function. We will usually call the template type itself a functor if the lifting function for it is known or if it is trivial to define.

2.5.2.2 Optional values

Another useful functor in the `Type` category is the `std::optional` class template. It is a union (or a sum) type [Hud+07] that extends any C++ value type with a special value denoting an empty state or *diverging computation* [Mog91]:

$$optional\langle T\rangle = T + \{\bot\}$$

Optional values are useful when denoting missing values. A value might be missing because an error has occurred and the value could not have been calculated.

Also, a missing value can be a perfectly valid state. For example, in a system which tracks the currently logged-in user, the value can be missing when there is no user currently logged-in [Čuk18, Section 9.1.4].

The lifting transformation for `std::optional` can be defined in a similar way to that of `std::vector`. It would be easy to make `std::transform` work for optional values – a `std::optional` is semantically the same as a vector which does not have more than one item.

An alternative, much simpler, solution would be to create a custom lifting function tailored exactly for `std::optional`:

```
template <typename F>
auto lift_to_optional(F&& transformation_function)
{
    return [transformation_function] (const auto& optional_value) ->
        std::optional<decltype(
            std::invoke(transformation_function, *optional_value))
        >
    {
        if (optional_value) { ❶
            return {
                std::invoke(transformation_function, *optional_value) ❷
            };
        } else {
            return {}; ❸
        }
    };
}
```

$$
\begin{array}{lcc}
optional\langle R\rangle: & \{\} & \{f(value)\} \\
 & \uparrow\ lift\_to\_optional(f)\ \uparrow & \\
optional\langle T\rangle: & \{\} & \{value\}
\end{array}
$$

Figure 2.5: Applying lifted function $f : T \to R$ to an optional value

The `lift_to_optional` function returns a closure object that represents the lifted `transformation_function`. When the closure is invoked with an optional value, it will check whether the value is missing or not. If the value is present ❶, the closure

will return the transformed value ❷, and an empty optional value otherwise ❸ (also see Figure 2.5).

Optional values are often useful, and lifted functions make it easier to use them. Imagine the following scenario – the program has an optional value `login` that contains the username of the user that is currently logged-in. If no user is logged-in, the optional value is empty. Further, it contains the following functions:

```
user_t user_data(const std::string& username);
std::string user_t::full_name() const;
html_t format_full_name(const std::string& full_name);
```

If the program needs to calculate the formatted full name of the currently logged-in user without lifting, it would need to process the special state when no user is logged-in separately, and it would need to manually extract and process the value in the case when there is a user logged-in into the system.

With function lifting, it is possible to define the required transformation without even considering there is a special state to worry about:

```
auto lifted_user_data = lift_to_optional(user_data);
auto lifted_full_name = lift_to_optional(&user_t::full_name);
auto lifted_format    = lift_to_optional(format_full_name);

std::optional<html_t> current_user_full_name()
{
    return lifted_format(
                lifted_full_name(
                    lifted_user_data(login)
                )
            );
}
```

Implementing a transformation on an optional value is just a matter of composing lifted functions in the same way normal functions would be composed for a normal value.

Even more, if the limitation that the system can have only one user logged-in is removed at some point during the development, and the optional value `login` is replaced with a list of users that are logged-in, changing the above function to generate a list of formatted names for logged-in users would be trivial – instead

of lifting to `std::optional`, it would be possible just to lift the functions to the `std::vector` functor.

If the optional `login` value was processed manually, without function lifting, it would be necessary to make significant changes to the implementation of the above function in order to make it work for a list of logged-in users.

2.5.3 Monads

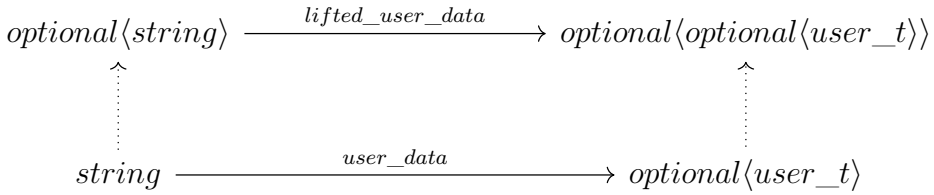
We have shown how function lifting can make chaining functions that transform functor type values as easy as transforming normal values.

The only downside is that in order for the lifted functions to be composable, the functions that are being lifted need to be composable as well. There are situations when this is a serious limitation.

Let us continue with the previous example, but this time, let us change the signatures of the functions a bit. Instead of having the functions that return normal values, the program needs to handle functions that can fail and return an invalid value. This optional error state can be denoted by replacing return types with optional values:

```
std::optional<user_t> user_data(const std::string& username);
std::optional<std::string> user_t::full_name() const;
std::optional<html_t> format_full_name(const std::string& full_name);
```

The problem that arises here is that if these functions are lifted, they will produce functions that are not composable because the original functions are not composable. Let's see what is the exact problem on the `user_data` function:



The `lifted_user_data` function returns a nested optional value – an optional value inside of another optional value. In order to call `user_t::full_name` on its result, it would need to be lifted two times. And, for the same reason, the `format_full_name` function has to be lifted three times. The longer the chain of functions that should be applied, the more lifting needs to be performed. In the end, the result is a value deeply nested in several layers of `std::optional`.

This is highly impractical because getting the result value in the end becomes more difficult with each new level – it needs to be checked whether the optional value is empty for each level until the value is reached. It is also semantically wrong – the result is either a value or it is empty, but instead of representing this as a single optional value, the result can have different invalid states where the differences between those invalid states do not communicate anything.

Since the failures at different levels do not carry any important information, they can be merged into one by flattening out the nested optional value into a normal optional value. Specifically, we can create a `join` function which removes one level from a nested optional value:

```
template <typename T>
std::optional<T> join(const std::optional<std::optional<T>>& outer)
{
    if (!outer) return {}; ❶

    const auto& inner = *outer;
    return inner; ❷
}
```

If the outer optional value is empty ❶, the `join` function returns an empty optional value as the result. Otherwise, it returns the inner optional value ❷. With a `join` function defined like this, the problem of having nested optional values can easily be avoided. If we define a `join` function for a functor that also has a constructor function, we get another useful structure called a *monad*.

Definition 2.3. A *monad* $M = \langle M, make, join \rangle$ on a category C consists of an endofunctor $M : C \to C$, and two transformations $make : C \to M(C)$, and $join : M^2(C) \to M(C)$ satisfying the following rules (usually referred to as *monad laws*):

Left identity: $mbind(make(a), f) \equiv f(a)$
Right identity: $mbind(m, make) \equiv m$
Associativity: $mbind(mbind(m, f), g) \equiv mbind(m, \lambda x.mbind(f(x), g))$

where $mbind(m, f) = join(M(f)(m))$, a is an object in C, f and g are arrows from C to $M(C)$, and m an object in $M(C)$.

The *make* transformation takes an object from C, and *wraps it into the monad* – it returns an object from $M(C)$, while the *join* transformation allows us to unwrap

values that have been wrapped multiple times. This means that it is possible to create a singly-wrapped object from any object in C – if the object x is in $C \setminus M(C)$, then $make(x)$ is in $M(C) \setminus M^2(C)$; whereas if x is in $M^n(C) \setminus M^{n+1}(C)$ where $n > 1$, $join^{n-1}(x)$ will be in $M(C) \setminus M^2(C)$.

The *mbind* transformation takes a monadic value m from $M(C)$, and a transformation $f : C \to M(C)$. It applies the lifted function $M(f)$ (in the context of M being a functor) to the given object m, and returns the result with one level of nesting removed. It behaves similarly to the normal function composition, but it allows us to compose functions that take normal values while returning values wrapped in a monad.

2.5.3.1 Monad laws

The *left identity* law states that wrapping an ordinary value a into a monad and then binding it with function f is same as calling the function f directly on the value a.

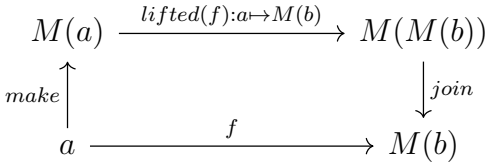


Figure 2.6: Left identity law

The *right identity* law states that binding a monadic value with the *make* function returns that monadic value unchanged.

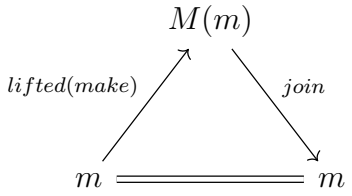


Figure 2.7: Right identity law

The meaning behind the *associativity* law is less obvious. In a nutshell, it states that binding a monadic value m first to function f, and then binding the result to the function g will give us the same result as binding the value m to the *composition* of f and g.

While f and g are not composable in the traditional sense because g takes a normal value and f returns a monadic value, they can be composed with a special composition operator:

$$f \odot g := \lambda v.mbind(f(v), g)$$

This operator is called *Kleisli* composition and it can be used to define the monad laws in a more understandable manner:

Left and right identity: f $\odot$ make $\equiv$ f $\equiv$ make $\odot$ f
Associativity: (f $\odot$ g) $\odot$ h $\equiv$ f $\odot$ (g $\odot$ h)

It is now obvious why the last monad law is called the *associativity* law – while it does not look like associativity in its original form, it is the law that requests the Kleisli composition operator defined on any particular monad to be associative.

2.5.3.2 The partiality monad

So far we have demonstrated what optional values are, and we have shown how functors and lifting help us handle them in a clean and generic way.

Lifting becomes insufficient when transforming optional values using partial functions – functions that are not defined for all values in their domain. Partial functions are usually modeled in practice as ordinary functions whose return type is an optional value. As we have seen before, lifting these functions gives us functions that return multi-layered optional values.

In this context, the Kleisli composition allows us to compose partial functions in the same way normal functions are composed, without the need for special handling of partiality.

The `std::optional` functor, along with the previously defined `join` function and the `std::make_optional` function make the partiality monad (often referred to as the *Maybe* monad).

Let's return to the scenario of having to process the currently logged-in user and generate a formatted representation of the user's full name. The example contained several partial functions represented as functions that return optional values. Passing the optional `login` variable through a chain of partial functions, can be done by applying the Kleisli-composition of those functions to it:

$$(format\_full\_name \odot full\_name \odot user\_data)(login)$$

$$
\begin{array}{lcc}
expected\langle R, E\rangle: & \{error\} & \{f(value)\} \\
 & \uparrow \quad lift\_to\_expected(f) & \uparrow \\
expected\langle T, E\rangle: & \{error\} & \{value\}
\end{array}
$$

Figure 2.8: Applying lifted function $f : T \to R$ to an expected value

The same result could be achieved using the *mbind* function, but it would be more verbose. We will present a convenient syntax for chaining *mbind* operations in Section 2.4.6.

2.5.3.3 The exception monad

The saying goes "The chain is as strong as its weakest link". This is true also for our chain of Kleisli-composed partial functions. The first function in the chain that returns an empty optional object ($\perp$) as its result will break the chain of transformations and the whole expression will return an empty optional value without executing any of the following transformations.

This is useful for error handling when the program does not care about the error itself, but only whether the error has occurred or not. A returned empty optional value denotes that a transformation has failed, but it does not convey the information which transformation has failed, nor what the error was because it can represent only one possible error state.

If we wanted to keep the usage patterns that the partiality monad has, but the information about the error needs to be preserved, it can be done by using a new functor similar to `std::optional` with the *empty* optional state replaced with a set of possible error values:

$$expected\langle T, E\rangle = T + E$$

An instance of the *expected* class template [3] can contain either a value of type T, or an error of type E, but not both at the same time.

It is important to note that the `expected` template is parametrised on two types – one type for the value, and one for the error. This is in odds with the functor definition which requires the functor to be a function that maps one type to another, and not a pair of types to a new type. In order to subvert this problem, we will consider that the type E is fixed and that `expected` is parametrised only on T.

[3] The `expected` class template is proposed for inclusion into C++20 [Esc17]

The lifting function for `expected` can be defined in a similar way to the lifting function for `std::optional` presented earlier. The only difference is that now it needs to carry on the error information:

```
template <typename F>
auto lift_to_expected(F&& transformation_function)
{
    return [transformation_function] (const auto& expected_value) ->
        expected<
            decltype(
                std::invoke(transformation_function, *expected_value)),
            decltype(expected_value.error())
        >
    {
        if (expected_value) { ❶
            return {
                std::invoke(transformation_function, *expected_value) ❷
            };
        } else {
            return unexpected{expected_value.error()}; ❸
        }
    };
}
```

The `lift_to_expected` function returns a closure object that represents the `transformation_function` lifted to the `expected` functor. When the closure is invoked with an expected value, it will check whether it contains a value or an error. If there is no error ❶, the closure will return the transformed value ❷. Otherwise, it will just pass on the previous error ❸ (also see Figure 2.8).

The implementation of the `join` function would also be very similar to the `join` function for `std::optional`. Instead, we will implement the `mbind` function for the `expected` functor, which will, alongside the `expected` class constructor make turn the `expected` functor into the *exception monad* [Mog91].

```
template <typename T,
          typename E,
          typename Function>
auto mbind(const expected<T, E>& expected_value,
```

```
            Function transformation_function) ->
    decltype(std::invoke(transformation_function, *expected_value))
{
    if (!expected_value) {
        return unexpected{ expected_value.error() }; ❶
    }

    return std::invoke(transformation_function, *expected_value); ❷
}
```

The `mbind` function accepts an expected value and a transformation function that takes a normal value and returns a new expected value. If the expected value that is passed to the `mbind` function contains an error ❶, `mbind` will return that error. Otherwise, it will invoke the `transformation_function` on the value and return its result ❷.

There are two important consequences of this implementation:

- If the expected value passed to `mbind` contains an error, the transformation function will not be invoked at all;
- The only way to get a valid value in the resulting expected value is if the original value was valid and if the transformation function succeeded;

When using the `mbind` function to chain multiple functions that return an expected value, we get a sequence of commands that are executed one by one for which the execution stops as soon as any of the commands fail. This is the reason why this is called the *exception monad* – it behaves similarly to the exceptions system in most programming languages with a difference that the exception is encoded in the return type of the function.

2.5.3.4 The continuation monad

In functional programming, a continuation [SS76] is a type C constructed from a function $f : A \to R$ that generates a result of type R. A is the type produced by the continuator, passed on to the continuation. We can connect the continuator and the continuation with the `then` function. In Haskell, it can be defined like so:

```
newtype C r a = CT ((a -> r) -> r)
then (CT action) k = action k
```

If we wanted to compose two continuations, we would need a function that looks like this:

```
cmpCont :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
cmpCont g f = \b -> g (\x -> f x b)
```

If we take a closer look at the signature of the `cmpCont` function, we will see a pattern that we have seen earlier – it has the same signature as the monadic bind function when `M r a = (a -> r) -> r` (where the monad is parametrized on `a`). It can easily be shown that this function obeys the laws that the monadic bind function has to obey, which makes a category of all continuation functions a monad.

3 Functional reactive programming

3.1 Futures and continuations

The traditional approaches to asynchronous programming shown in Section 2.2 are all valid, but have readability and maintainability issues. The task we have is quite simple: request a page, and process it when it is retrieved. Something like this should be easy to implement. It needs to be as simple as:

```
result = get_page("http://www.math.rs/");
if (result.is_valid()) {
    …
}
```

when implemented in the imperative style, and if implemented in the functional style, it needs to be as simple as:

```
get_page("http://www.math.rs/")
    | filter(&response_t::is_valid)
    | transform(…)
```

The only reason we can not write this is that the `get_page` function has to be asynchronous, not to block the execution of the other parts of the system.

All the solutions we saw in the previous chapter require the program logic to be split into two parts – one part that handles the logic before the asynchronous call, and one part that comes after the asynchronous call has completed. We usually need to perform several asynchronous calls which means that we need to split the program logic into many smaller parts which are not its logical sub-routines. This makes the software difficult to write, understand and maintain. We need better abstractions than these.

3.1.1 Decoupling task creation from result handling

The lowest level of abstraction we want to create are the *futures*. A future represents a handler for a value that is not yet known, but which will be known at some point in the future. While this may sound confusing, the concept is quite simple. Consider the following conversation between Alice and Bob:

```
Alice: What is the time?
(Bob looks at his watch)
Bob:   It is 12:30.
```

When Alice asks the question, she only knows the type of the answer – that it will be a type that defines (at least) hours and minutes. We will call that type `time_t`. Her request for the current time can be written as follows:

```
bob.get_time();
```

This way, she is asking Bob for the time, but the problem is that she is not saving the result anywhere – she will never learn what is the current time. Can we fix this by assigning it to a variable of type `time_t`?

```
time_t time = bob.get_time();
```

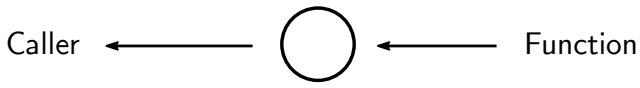


Figure 3.1: Returning a proper value from a function

If we want to be able to save the response like this, we first need to wait for Bob's reply. This means that Alice can not do anything until Bob answers the question. This would be a bad idea since we can not know how much time it will take Bob to reply, and even whether he will reply at all. The data flow for this case can be seen in the Figure 3.1, where the caller is Alice, function is `bob.get_time()`, and the resulting value is represented as a circle.

Since we do not want to block Alice indefinitely, we will have to define the `time` variable not to be a normal value, but a future value:

```
future<time_t> time = bob.get_time();
```

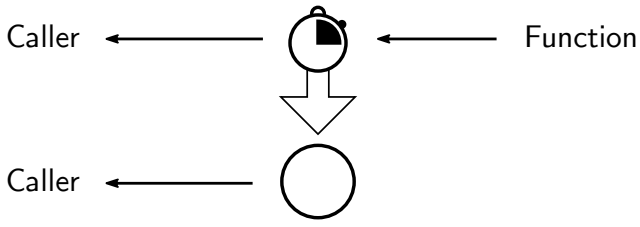


Figure 3.2: Returning a future from function

This creates an empty container that can contain an instance of the `time_t` class. When Bob prepares the answer, he can just put it into the container, and Alice will be able to get it by calling `time.get()`. If the value is already present, the `get`

member function will not block as it can return the value immediately. This will allow Alice to continue working on other things, and still get Bob's answer when it arrives. The data flow in the Figure 3.2 shows that when the caller invokes the function, it will not receive the resulting value, but an object that can later be used to get it, when it becomes available.

3.1.2 Futures in C++

There are a few different classes for C++ that implement futures with slightly different features and semantics. The most notable ones are:

`std::future<T>` provides a minimal feature-set to allow easy implementation of multi-threaded systems based on the fork-join idiom [Hal13];

`boost::future<T>` which is a drop-in replacement for the `std::future<T>`, but extends it with features that are planned for inclusion in the future revisions of the C++ standard like future composition and chaining;

`QFuture<T>` from the Qt library [Čuk16b] which has similar features to the previous ones, but with a syntax based on the signals and slots mechanism; and the

`folly::Future<T>` from the Facebook's Folly library [Fug] which has special semantics inspired by the Future class as implemented in the Scala programming language [Ale13, Chapter 13].

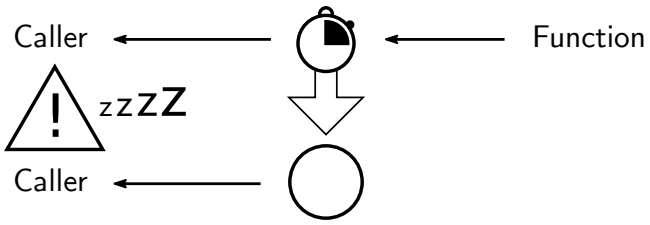


Figure 3.3: Calling the `get` member function on a future

All of these classes have a `get` member function which returns a value contained in the future class. The difference is in the behaviour of this function when the value is not yet available. All but the `folly::Future<T>` block the execution of the caller until the result becomes available (Figure 3.3), which is only useful in the fork-join pattern where the caller creates a few asynchronous requests that should be executed in parallel, and it needs the results of all of them to proceed with the execution. In our example in which we call only `bob.get_time()` it makes the asynchronous execution of the function useless and equivalent to not using the futures at all. In the case

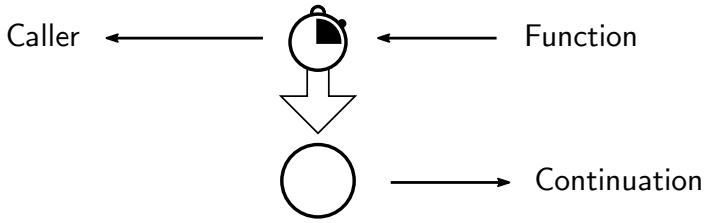


Figure 3.4: Passing the value to the continuation function

of calling the `get` member function on an unfinished future, the `folly::Future<T>` class will throw an exception, which is in accordance with the idea that asynchronous functions should never block the caller, and forces the caller to implement the logic of asynchronous execution in a proper way.

3.1.3 Proper way of handling futures

Having the caller blocked while it is waiting for the result defeats the purpose of the asynchronous API. The main reason why we wanted the time-consuming operations done asynchronously in the first place is to allow the main program to continue working (processing user input, accepting new network connections, creating new asynchronous requests, etc.) while it waits for the result. This means that designing the system around the `get` member function of a future is a bad idea.

Instead of relying on the caller to get the value from a future manually, we only need to provide it a way to define a continuation function for a specific asynchronous operation. The continuation is a function that will be invoked when the result becomes available [Cla99]. This way, we will never try to get the value from an unfinished future, we will just define what should be done with the value once it becomes available. In the following example, instead of Alice getting the result of the asynchronous computation, she is just scheduling a continuation (`print_time`) function which will print the result when Bob produces it:

```
bob.get_time() | continue_with(print_time);
```

The minimal interface a future should have consists of a constructor and a way to chain a continuation to said future. This abstracts all of the previously shown ways to implement asynchronous functions, and is also able to cover the synchronous function calls:

Calls with callbacks can be represented as futures in a straight-forward way. Let f be an $(n+1)$-ary function, where the first n arguments are used for the

computation, and the last argument is the callback function. Then the unary function g we get when we bind those first n arguments to specific values behaves like the future (note that the `std::bind` function has nothing to do with monadic bind, it just implements partial function application in C++ [Čuk18, Chapter 4]).

```
auto g = std::bind(f, arg1, arg2, …, argn, _1);
```

This successfully separates the definition of an asynchronous operation with its arguments (the constructor for this type of future), from the continuation specification.

Signals and slots can also be represented in a simple manner. The signal sender object, along with the signal provide the constructor for the future, while the slot represents the continuation.

Message passing in actor systems needs a bit more effort to in order to represent it as a future. Since we have already modelled callback functions as futures, we can just model message replies with the callback functions.

An asynchronous operation that has a result (a returning value) in actor systems consists of one message that is sent from the caller to the callee, and then the result is returned via another message sent back to the caller. The caller will need to be able to create a unique identifier for every call it makes, and the callee will need to return that identifier along with the calculated result. We will show how to represent this in the call-callback approach which we have shown is easy to represent using futures.

```
void f(auto arg1, auto arg2, …, auto argn, auto callback)
{
    auto request_id = generate_id();
    global_callbacks[request_id] = callback;
    callee.send_message(arg1, …, argn, request_id);
}

void receive_message(auto value, auto request_id)
{
    std::invoke(global_callbacks[request_id], value);
    global_callbacks.remove(request_id);
}
```

The function f gets the arguments that will be passed to the actor that per-

forms the asynchronous computation (`callee`) and the callback function that should be called when the result becomes available. The callback is saved to a structure that maps identifiers to callbacks. The callee will send beck the result, which will invoke `receive_message` function. This will, in turn, invoke the callback associated with the specified identifier.

Synchronous function calls can also be modelled with futures. If the n-ary function f is a synchronous one, we can easily create a callback-based function from it.

```
void g(auto arg1, auto arg2, …, auto argn, auto callback)
{
    std::invoke(callback, f(arg1, arg2, …, argn));
}
```

The fact that we have created an abstraction that works for both synchronous and asynchronous computations will be quite useful later.

3.1.3.1 The continuation monad

Let us abstract out the specific definitions and implementations of various future-like constructs we have seen and define the minimal interface they all should have. The most basic operations we need to have are construction and continuation function specification. These basic operations allow us to specify the asynchronous task that needs to be performed, and to define the handler for the result of the asynchronous operation.

Consider the following problem. Imagine that Alice wants to get the current time, and if it is noon, she wants to ask Bob to lunch. Without asynchronous execution, this would be trivial to implement:

```
if (bob.get_time() == 12h) {
    if (bob.wants_to_eat()) {
        …
    }
}
```

Alice is asking Bob two questions and both of those need to be asynchronous for the same reasons we saw in the previous section. Before asking the second question, Alice first needs to get the result of the first one. The minimal interface for future values we defined would allow us to solve this problem.

```
bob.get_time() | continue_with([=] (auto time) {
    if (time == 12h) {
        bob.wants_to_eat() | continue_with([=] (auto time) {
            …
        });
    }
});
```

The code is localised, but it is not readable, and Alice has no clean way to know when the second asynchronous operation (`bob.wants_to_eat`) has been finished. The only approach that Alice could take to be notified of when the asynchronous operations have been finished is to make the continuation functions explicitly notify her which can not be implemented in a pure way.

Instead, we need to increase the requirements for the `continue_with` function. Since Alice is effectively interested only in the result of the final operation, it would be prudent to make the `continue_with` function to return a future value that will contain the result of the continuation function passed to it. With this improvement, the previous example could become much simpler, and Alice will be able to keep track of which operations have been completed:

```
auto future_time = bob.get_time();

auto future_answer = future_time | continue_with([=] (auto time) {
    if (time == 12h) {
        return bob.wants_to_eat();
    } else {
        return { false };
    }
});

future_answer | continue_with(…);
```

Let us analyze this a bit from the point of category theory. We have two functions `bob.get_time` and `bob.wants_to_eat` that take ordinary values and return a future value:

$$get\_time : person \to future\langle time \rangle$$

$$wants\_to\_eat : person \to future\langle bool \rangle$$

And we have a `continue_with` function that composes them into a single function of type $person \to future\langle bool \rangle$.

We have a generic *wrapper* type parametrized on a single type (an arrow between types) and we have a function that allows us to compose functions that take normal values and return the wrapped values. If we add a constructor function that takes a value and constructs a wrapped value out of it, we get the *continuation monad* [SS76] where the `continue_with` function is the monadic bind (we will call it `mbind` from now on) – the `mbind` function, as seen in Section 2.5.3. (it would be easy to show that the monad laws are obeyed).

3.1.4 Wrapping futures into the continuation monad

Now that we have seen what the continuation monad is, let us see how the previously mentioned implementations of the future concept fit into this definition.

All of them are wrapper types parametrized on the type of the value that will be stored in the future object, and they all provide a constructor function that takes a value and creates a future object containing that value.

3.1.4.1 `std::future` and `boost::future`

The `std::future`, as of writing, has no possibility of connecting a continuation to it. If we wanted to implement the monadic bind function for it, we would need to spin a separate thread which would do a blocking wait for the result and invoke the continuation function when the result is ready. We will not implement this because the `std::future` is planned to be extended in C++20 [ISO15] to support the `then` member function in the same way that `boost::future` implements it now.

The `boost::future` provides a mechanism for chaining continuations using the `then` member function. The problem is that it expects a continuation function that takes a future value, instead of an ordinary value which is needed for monadic bind. This can be easily remedied by implementing our own `mbind` function that will wrap the continuation that expects a normal value into a continuation suitable for the `then` function.

```
template <typename Future, typename Continuation>
decltype(auto) mbind(Future&& future, Continuation cont)
{
    return std::forward<Future>(future).then(
```

```
        [=] (auto f) {
            return std::invoke(cont, f.get());
        }
    );
}
```

The reason why `then` requires a continuation function that accepts a future object instead of a normal value is that the `boost::future` has an additional state beside waiting for a value or having a value – it can also contain a pointer to exception if an exception has occurred during the asynchronous operation. This means that the `boost::future` does not really implement the continuation monad – it implements a composition of the continuation monad and the exception monad (Section 2.5.3.3). The `mbind` function we implemented assumes that no exceptions can happen to model the continuation monad only.

3.1.4.2 `QFuture`

The `QFuture` is similar to the `boost::future` in that it models a composition of the continuation monad and the exception monad. But it has a significantly different syntax as it does not provide the `then` member function. In order to implement the `mbind` function for the `QFuture`, we will need to use Qt's signals and slots mechanism:

```
template <typename T, typename Continuation>
decltype(auto) mbind(const QFuture<T>& future, Continuation cont)
{
    auto* watcher = new QFutureWatcher<T>();
    QObject::connect(watcher, SIGNAL(finished()),
                     watcher, [=] {
                         std::invoke(watcher->future().result());
                         watcher->deleteLater();
                     }
    …
}
```

This implementation demonstrates only how a continuation function could be connected to a `QFuture`. Returning a future value containing the result of the continuation is significantly more involved (see Appendix C).

3.1.4.3 `folly::Future`

The `folly::Future` class has the cleanest design of all. It is also a composition of the continuation and exception monads (in Folly nomenclature called `Try` monad), but unlike `boost::future` and `QFuture` this is communicated clearly through the `then` member function API. Instead of having a continuation function accept an instance of a future object, in `folly::Future`, the continuation function accepts an instances of the `Try` monad.

If we accept that `folly::Future` is a composition of the continuation and exception monads with the error type defined as exception pointer, then the `then` member function is a proper implementation of monadic bind, and the `makeFuture` function is a proper constructor function because it can create a `Future<T>` object from an instance of `Try<T>`.

3.1.4.4 Transforming futures

We have seen that future values can be modeled as monads. This means that we can use all functions that we can define over a generic monad interface, on future values. If we have the `mbind` function and a future value constructor, we can easily implement functions like `transform` and `join`.

The transform function is easy to implement – we just need to compose the future constructor and the transformation function and pass that as the continuation function to `mbind`:

```
auto transform(F future, T tansformation)
{
    return mbind(future, [=] (auto value) {
        return make_ready_future(
            std::invoke(transformation, value));
    });
}
```

As we have seen in Section 2.5.3, the `mbind` function does exactly what `transform` does, but it flattens the result in the end. This implementation of `transform` relies on that fact and returns the result of transformation wrapped inside another future object which `mbind` will unwrap.

Futures also share a few other features with the monads we have seen in the previous chapter – features that are not common to all monads. Since a future can either get a value at some point, or never get a value at all, we can also implement a

`filter` function on it, similar to the partiality monad and ranges, where the filtered future would get a value if the source future gets a value that satisfies a predicate, or never get a value otherwise.

3.2 Reactive streams

It is interesting to see that many transformations defined on ranges can also be applied to futures. If we wanted to get a square of the given future value, we could do it simply by calling `transform` on it:

```
future<int> f = answer_ultimate_question_of_life()
                                        | transform(square);
```

Instead of `f` being a future that will contain 42 (the result that is returned by the `answer_ultimate_question_of_life` function), it will be a future that will contain 1764 (42^2). While, theoretically, we can also apply `sort`, `group`, etc. to a future, it does not make much sense, since a future contains at most one value.

What happens if we remove the limitation that the future can return only one value? Many processes in the real-world do not generate only a single value, but a series of values. Not a series in the sense that we get a collection of items at some point in the future, but in the sense that we get a new value from time to time.

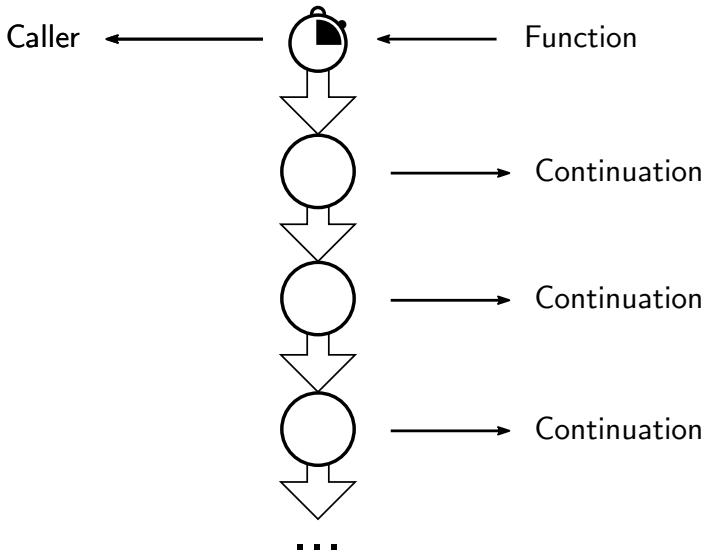


Figure 3.5: Stream of values

Examples of this include mouse movement where we get a new screen coordinate (with some additional data) every time the user moves the mouse cursor, keyboard

events where we get which key is pressed, client connections to a web service where we get some information about a client whenever a new one connects to our service, and similar.

This can not be modelled by futures, but is quite similar in nature. Just think of it as a future that returns a value and another future of the same type.

In order for our previously defined structures to support sending multiple values, the only thing that needs changing is to allow the continuation function (the function passed to `mbind`) to be called multiple times. We are not going to redefine the structures as we haven't encoded the limit of how many times the continuation can be invoked, but we will use a different nomenclature for futures that can generate multiple values – we are going to refer to them as reactive streams [WH00].

3.2.1 Reactive streams as push iterators

Before we move on to how reactive streams can help us implement asynchronous software systems, let us first cover how they relate to proxy iterators and ranges we have seen in the previous chapter.

Reactive streams are sequences just like ranges are, and they can be iterated over. Since we do not know in advance how many values a stream will emit (how many times the continuation function will be invoked), we can not know where the end of a stream is in advance. We can not know whether the stream has an end or not. This means that the reactive streams behave mostly like input ranges.

We based the input ranges on a *proxy-iterator* and *sentinel* pair, where the iterator can be used to access a value (consuming it in the process). The algorithms that operate on input ranges would be able to traverse the range and read the values one by one until enough elements have been found, or the end of a range has been reached.

If we were to allow the same algorithms to access reactive streams, we would encounter a big problem. Namely, the algorithms that operate on ranges request a value from a range which, in turn, requests the value from the range it proxies for, and this process cascades until a proper (non-proxied) value is found. If an algorithm tried to access a value from a reactive stream, this would have the same effect as calling the `get` member function on a future – it would either block the system until the future arrives, or it would throw an exception. This method of operation is usually called the *pull* method because we are pulling values from a sequence.

With reactive streams, we need to take a different approach. We can still have proxy iterators, but those iterators will not be iterators in traditional C++ sense – they can not be dereferenced – they can only be bound to a continuation function. This way, we can have the original stream of values, and the proxy iterators will just need to process the values they receive and emit the results. This is usually referred to as the *push* method because when a value arrives, it is pushed through all the transformations up to the consumer.

3.2.1.1 Source streams

The streams that only emit values are usually called *source* streams. These streams usually serve to handle any kind of input – they introduce values from external entities into the system (like user input, or messages from clients in a web service). As far as our software system is concerned, it is as if the source stream generates those values – we can not know anything about the actual source of the value unless this information is encoded in the value itself.

Since source streams only emit values, we only need to establish a way to connect a continuation function to them. A generic interface that a source stream needs to implement is as follows:

```
template <typename ValueType>
class source_stream {
public:
    using value_type = ValueType; ❶

    template <typename Cont>
    void set_continuation(Cont cont)          ❷
    {
        m_continuation = std::move(cont);
    }

    …

protected:
    std::function<void(ValueType&&)> m_continuation;
};
```

Each stream emits only values of a single type. In order to allow easier meta-programming, this type will be exported as the `value_type` nested type defini-

tion ❶, just like it is done for normal C++ standard library collections. When `set_continuation` is called, it will register the continuation function ❷ that will be notified whenever a new value appears.

It is important to note that streams do not need to inherit from the above type because we have no need for dynamic polymorphism – they just need to provide the same interface.

3.2.1.2 Stream sinks

On the other side of the spectre to source streams lie stream sinks. Sinks just receive values, but they do not emit any values whatsoever. Since sinks do not emit values, we do not need to provide a way to connect a continuation to them – they just need to be able to accept values. This means that any callable satisfies the requirements to be a sink if it can be called with the value type provided to the sink.

```
template <typename Sender,
          typename Function,
          typename InputType = typename Sender::value_type>
class sink_connection {
public:
    sink_connection(Sender&& sender, Function function)
        : m_sender(std::move(sender))
        , m_function(function)
    {
        m_sender.set_continuation(
            [this](InputType&& value)
            {
                process_value(
                    std::move(value));
            }
        );
    }

    void process_value(ValueType&& value) const
    {
        std::invoke(m_function,
                    std::move(value));
    }
```

When the sink_connection object is constructed, it will register itself as the continuation function of the source stream provided to the constructor

When we get a message, we just pass it on to the sink function

```
private:
    Sender m_sender;
    Function m_function;
};
```

In order to support creating connections to sinks using the range-like pipe syntax, we will need to provide some helper types and functions similar to what we had in Section 2.4.6. The difference here is that the object we want to define the pipe operator on is on the right-hand side of the operator:

```
stream | sink(print_value);
```

We need to create a helper function `sink` which just creates a dummy object to hold the function we want to use as the sink (`print_value`) and we need to implement a non-member pipe operator which accepts this dummy object as its second argument:

```
template <typename F>
class sink_helper {
    F function;
};

template <typename F>
decltype(auto) sink(F&& f)
{
    return sink_helper<F>{std::forward<F>(f)};
}

template <typename F,
          typename Stream>
decltype(auto) operator| (Stream&& stream, sink_helper<F>&& sink)
{
    return sink_connection<Stream, F>(
            std::forward<Stream>(stream),
            sink.function
        );
}
```

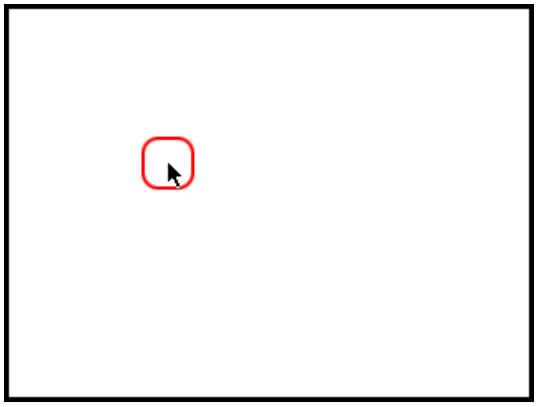


Figure 3.6: The `mouse_cursor` object (rectangle with rounded corners) follows the mouse cursor

3.2.1.3 Example source and sink

We are going to use the example of transforming the coordinates that get generated while the user is moving the mouse cursor, in order to explain reactive streams in a visual way. We will call the source stream that emits the mouse coordinates `mouse_position`. Initially, we can just connect the mouse position stream directly to a marker that will show the current position at all times like this:

```
auto mouse_cursor_sink = sink([] (point_t position) {
        mouse_cursor->move_to(position.x, position.y)
    });

mouse_position | mouse_cursor_sink;
```

This will create a sink to the function that moves the object called `mouse_cursor` to the specified coordinates and connect the `mouse_position` stream to it. Whenever the user moves the mouse, the `mouse_cursor` object will be moved. The window will look like in the Figure 3.6.

3.2.2 Map, or transform

Now that we have sources and sinks defined, we can implement stream transformations that will allow us to treat streams like they were ordinary input ranges.

First, let's implement the `transform` reactive stream modifier. The `transform` modifier needs to be able to listen to new values from the stream it transforms, and for each new value it should transform it with the provided transformation function and emit the result.

```
template <
    typename Sender,
    typename Transformation,
    typename InputValueType =
        typename Sender::value_type,
    typename ValueType =
        decltype(std::declval<Transformation>()(
            std::declval<InputValueType>()))>
class transform_impl {
public:
    using value_type = ValueType;

    transform_impl(Sender&& sender, Transformation transformation)
        : m_sender(std::move(sender))
        , m_transformation(std::move(transformation))
    {
    }

    template <typename Cont>
    void set_continuation(Cont continuation)
    {
        m_continuation = continuation;
        m_sender.set_continuation(
            [this](InputValueType&& value) {
                process_value(
                    std::move(value));
            });
    }

    void process_value(InputValueType&& value) const
    {
        std::invoke(m_continuation,
            std::invoke(m_transformation,
                        std::move(value)));
    }
```

The transformation proxy iterator emits values of the result type of the transformation function invoked on an input value

When we get a stream that wants to listen to our messages, we connect to our source stream

Sending the transformed value to the continuation

```
private:
    Sender m_sender;
    Transformation m_transformation;
    std::function<void(MessageType&&)> m_continuation;
};
```

The constructor only saves the sender and the transformation functions. When we connect a continuation (another proxy iterator or a sink) to it, it will save the continuation and it will start listening to the source steam. We will assume that we have also implemented the necessary helper classes and functions needed to use `transform_impl` with the pipe syntax since it is not significantly different than the implementation we had for sinks.

Now that we have a way to transform the mouse stream, we can create something more useful than an object that follows the mouse. For example, we can set the y coordinate to be a fixed number, and pass the events to the `top_ruler` object.

```
auto flatline_x = [](const point_t &point) {
        return point_t(point.x(), 10);
    };

auto top_ruler_sink = sink([] (point_t position) {
        top_ruler->move_to(position.x, position.y)
    });

mouse_position | transform(flatline_x) | top_ruler_sink;
```

Now, we get a new object whose position is a projection of the mouse coordinate on the top of the window like in the Figure 3.7.

3.2.3 Sharing a stream

But now we have a problem. Only one object is able to use the `mouse_position` stream. We need to be able to share a single stream between multiple listeners. The current design does not allow this for two reasons:

- each stream iterator we created allows only one continuation function to be defined; and
- we designed each proxy iterator to be a single owner of its source stream.

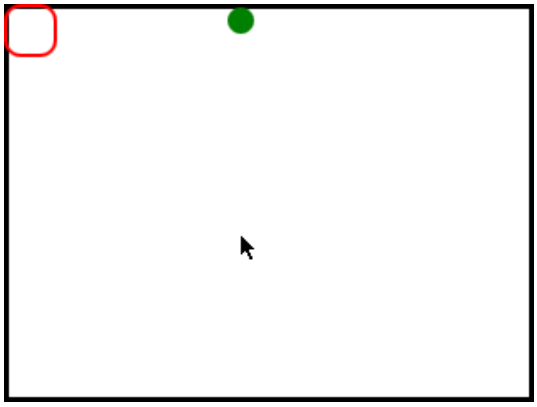


Figure 3.7: Top ruler projection

We can solve this problem in several different ways. We could reimplement all the previous stream iterators to store a list of continuation functions instead of limiting the number of continuations to be one. If we took that approach, we would also need to allow shared stream ownership – instead of stream iterators owning the source stream, they would need to allow shared ownership where the source stream is destroyed only if all stream iterators that are connected to it are destroyed.

The problem with this solution is that it would induce performance (atomic reference counting for shared ownership) and memory overhead (book-keeping data for a list of functions) even in the cases where there are no shared streams.

The alternative is just to create a special type of a stream – a stream that can be shared. Only shared streams would need to do reference counting and allow multiple listeners, which will have the performance overhead only in the places where sharing is needed.

This would allow us to use the same source stream of mouse coordinates to define positions of several objects – the mouse cursor indicator and its projections on the top and left edge of the window (Figure 3.8):

```
auto shared_mouse = shared_stream(mouse_position);

shared_mouse | mouse_cursor_sink;
shared_mouse | transform(flatline_x) | top_ruler_sink;
shared_mouse | transform(flatline_y) | left_ruler_sink;
```

3.2.4 Stateful function objects

Now a question arises. Is it safe to pass any callable to `transform`? If we are allowed to pass anything we want, we can also pass callables that have mutable state. Say, an

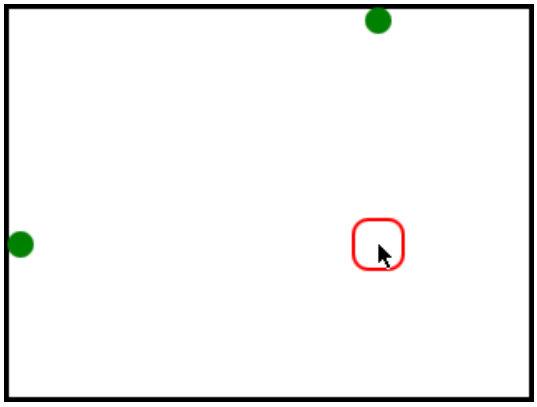


Figure 3.8: Both rulers projection, and the cursor

object that will try to move towards the mouse cursor, but slowly, without jumping (Figure 3.9). For this, it will need to store its previous position and use it when calculating the new one.

```
class inert_object {
public:
    inert_object()
    {
    }

    point_t operator() (const point_t &mouse_position)
    {
        current_position.x = current_position.x * .99
                           + mouse_position.x * .01;
        current_position.y = current_position.y * .99
                           + mouse_position.y * .01;

        return current_position;
    }

private:
    point_t current_position;
};
```

Would the following code be safe, or are we required to either pass only pure functions since we are working in a concurrent environment, or use a mutex to ensure all mutations performed by the transformation callable are atomic?

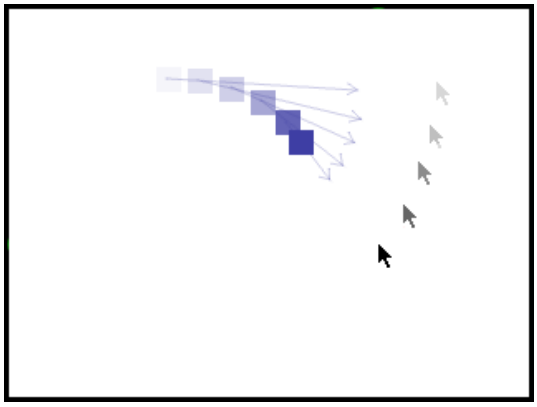


Figure 3.9: Rectangle that moves slowly towards the mouse

```
mouse_position | transform(inert_object()) | inert_marker_sink;
```

Usually, having mutable state in a concurrent system is ill-advised. It is one of the reasons why the pure functional programming languages like Haskell have been getting traction in recent years. There can be no data races if all data is immutable (persistent).

In the above code example, we have implemented a function object that has a mutable state, without mutexes or any synchronization at all. And we are using it to process the values from an incoming reactive stream. It is obvious that, in general, having mutable state can lead to concurrency problems. But the question is whether all mutable state is bad.

Having mutable state is a problem when several entities try to access and modify the same value at the same time – concurrently. Mutexes are used to force concurrent processes to wait for each other so that only one process can update a certain value at one point in time. But, if said value is not shared between multiple concurrent processes, then it can not be a problem. Mutable state is only problematic when it is shared (Figure 3.10). Proper design of reactive stream flows allows defining stream transformations that do not share any data.

It is worth noting that by passing `inert_object()` to the `transform` function, we are creating a new instance of that object, which none of other range iterators can access – it has just one owner, and only one process can access it.

3.2.5 Stream filtering

Another useful stream transformation is filtering. Sometimes we want to ignore some type of events. For example, a button in the UI usually does not care about mouse movements outside of its area.

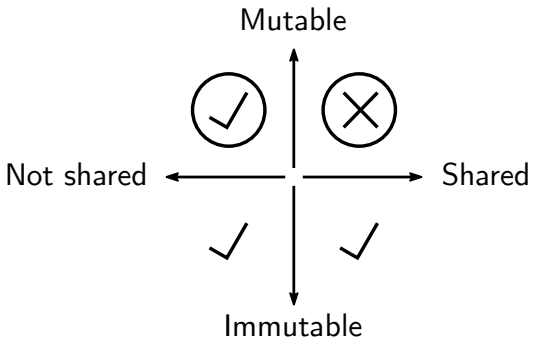


Figure 3.10: Mutable versus immutable state

Unlike `transform` which emitted one transformed value for each value it received, `filter` emits the values it received unmodified, but only if the value satisfies the given predicate:

```
template <typename Sender,
          typename Predicate,
          typename ValueType =                    | Filtering emits the same type
              typename Sender::value_type>        | of values it receives
class filter_impl {
public:
    using value_type = ValueType;

    …

    void process_value(ValueType&& value) const
    {
        if (std::invoke(m_predicate, value)) {    | Emits the value only
            std::invoke(m_continuation,           | if it satisfies the
                        (std::move(value));       | predicate
        }
    }

private:
    Sender m_sender;
    Predicate m_predicate;
    std::function<void(ValueType&&)> m_continuation;
};
```

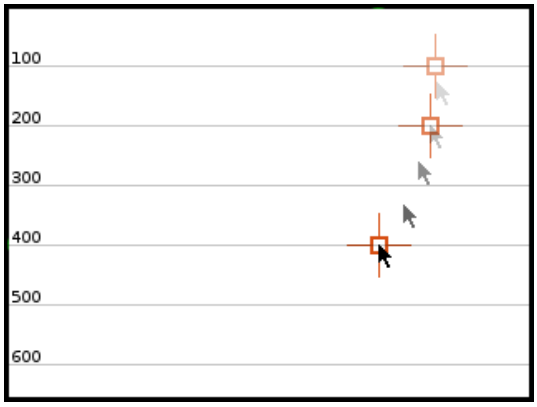


Figure 3.11: The cross-hair is snapping to guide-lines

Filtering can be useful in the previous example if we wanted to make the mouse cursor snap to positions where the y coordinate is divisible by 100.

```
mouse_position | filter([] (point_t point) { return point.y % 100 == 0; })
               | snapping_marker_sink;
```

This has the effect of snapping to guide-lines. In this case, the guide-lines are fixed to every 100 pixels, and are horizontal only (Figure 3.11).

3.2.6 Generating new stream events

There is a bug in the previous example. As can be seen in the Figure 3.11, one guide-line (at 300 pixels) has been skipped. This can happen because the mouse movement is not continuous, and there is a chance that while the mouse moves across the line, the mouse event for the exact $y = 300$ will not be emitted. The faster the mouse cursor moves, the greater probability that it will skip the guide-line.

In order to remedy this, we need to make the stream of coordinates continuous. In this case, since the coordinates are integers, continuous means that we want to create a stream in which the difference in coordinates for two consecutive events is not greater than 1 pixel per axis. The simplest way to achieve this is to remember the previous mouse coordinate, and generate the Manhattan path to the new one.

```
class more_precision {
public:
    using value_type = point_t;
```

```
    void process_value(point_t&& new_point)
    {
        // Going left (or right) until we reach new_point.x()
        int stepX = (m_previous_point.x() < new_point.x()) ? 1 : -1;
        for (int i = (int)m_previous_point.x();
                 i != (int)new_point.x(); i += stepX) {
             m_continuation(point_t(i, m_previous_point.y()));
        }

        // Going down (or up) until we reach new_point.y()
        int stepY = (m_previous_point.y() < new_point.y()) ? 1 : -1;
        for (int i = (int)m_previous_point.y();
                 i != (int)new_point.y(); i += stepY) {
             m_continuation(point_t(new_point.x(), i));
        }

        m_previous_point = new_point;
    }

    …

private:
    std::function<void(point_t&&)> m_continuation;
    point_t m_previous_point;

};
```

Unlike the previous stream iterators, the `more_precision` iterator is a highly specialized one – it can only transform streams of coordinates and nothing else. For each new coordinate it receives, it will generate all the mouse positions between the previously saved position and the new one.

In order to fix the bug from the previous program, we only need to add a single stream transformation (Figure 3.12):

```
mouse_position | more_precision()
               | filter([] (point_t point) { return point.y % 100 == 0; })
               | snapping_marker_sink;
```

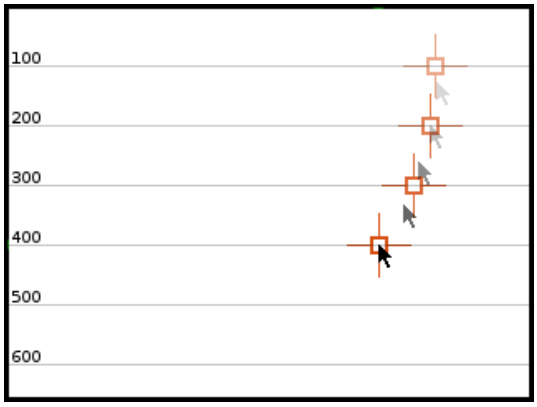


Figure 3.12: The cross-hair is snapping to guide-lines, fixed

3.2.7 Reactive streams as a monad

We have seen several examples of how a reactive stream can be transformed as if it was an input range. By defining the `transform` stream modifier, we have shown that reactive streams are functors. In order to show that reactive streams are also monads, we will have to define two functions more – a function that constructs a reactive stream from a given value, and a function that flattens out nested streams.

The former is easy, and we can even improve on the requirement a bit. Instead of creating a function that converts a single value to a reactive stream, we will implement a function that can convert a sequence of any number of values into a stream.

```
template <typename T>
class values {
public:
    using value_type = T;

    explicit values(std::initializer_list<T> values)
        : m_values(values)
    {
    }

    template <typename Cont>
    void set_continuation(Cont continuation)
    {
        for (auto&& value: m_values) {
```

```
            std::invoke(continuation, std::move(value));
        }

        m_values.clear();
    }

private:
    std::vector<T> m_values;
};
```

The `values` type implements a source stream that emits its values as soon as a continuation is registered. It does not need to store the continuation at all because it is not needed any more after the values are sent, and it can destroy all the stored values as soon as they are emitted.

Implementing the `join` operation that flattens out a stream of streams is a bit more intricate. It receives a stream of streams and it needs to listen to all of them and pass on the messages it receives. It also needs to become the owner of all streams it receives in order for them not to get destroyed.

```
template <
    typename Sender,
    typename NestedStreamType = typename Sender::value_type,
    typename ValueType = typename NestedStreamType::value_type>
class join_impl {
public:
    using value_type = ValueType;

    void process_value(NestedStreamType&& source)
    {
        m_sources.emplace_back(std::move(source));          ❶
        m_sources.back().set_continuation(m_continuation);
    }
    …
private:
    Sender m_sender;
    std::function<void(ValueType&&)> m_continuation;
    std::list<NestedStreamType> m_sources;
};
```

The `join` transformation will receive new streams as values to process. Each stream an instance of `join_imple` receives will be appended to the `m_sources` list to prolong that stream's lifetime, and it will set its continuation function to also be the continuation of all received streams ❶.

With `join` and `transform`, we can easily create the monadic bind transformation for reactive streams.

3.2.8 Data lifetime

An important thing to note regarding the design of reactive streams presented in this dissertation is that each node in the stream connects to its parent only when there is a node registered to listen for the values coming out of said node. The only exceptions are the sink nodes as they connect to their parent immediately.

This means that, while the node connection operator (the pipe operator) is left-associative, the connections (continuation function registration) happen from right to left when a sink is added to the pipeline. Until a sink is registered, no connections are performed. After the whole pipeline is connected, each value that a source node emits will be pushed through the nodes from left to right until it reaches the sink.

A side-effect of this design is that the source nodes must be implemented in such a way that they are not allowed to emit any values until their continuation function is set. This was demonstrated in the previously implemented `service` source node which started accepting client connections only after its continuation function has been set.

If a source node is not implemented correctly, and it tries to emit a value before it gets connected to a child node, the system will crash. While this might seem like a downside of the proposed design of reactive streams, it follows the C++ mantra of exposing the errors in the software system design as early as possible – in this case as soon as the first value is emitted.

A properly designed software system has to define the lifetime of a value precisely. As far as the proposed design of reactive streams is concerned, the value should be constructed by the source node, moved through different transformations and consumed and destructed by the sink node. It should not be allowed for a value to get lost somewhere in the pipeline unless it is explicitly ignored by a node that chooses not to pass it on as it is the case with the `filter` transformation.

This design allows for efficient value passing through the transformation pipeline without the need for garbage collection and unnecessary copying of values which are often present in other functional reactive systems.

3.2.9 Data-flow software design

In order to write functional reactive systems, one must start to think about them as data-flows [Bai+13]. More specifically, to analyze what is the input to the system, what is the input and output of different system components, and how the data should be transformed between them.

The input streams represent the data that is coming into the system from the outside world, while receivers represent either internal system components that react to the requests, or the data that is emitted back to the outside world.

The transformations that happen in-between, can be either simple ones described in the previous sections, or complete system components that receive some message types, and react to them by emitting different ones.

Let us demonstrate this with an example of a simple bookmarking web service that listens for connections on several different ports at the same time. The web service will listen for JSON-encoded messages containing the bookmark information, and will send the client a response whether the bookmark is valid or not after writing it to the standard output.

When designing a system like this, we first need to choose what are the inputs. In this case, it would be the easiest to say that input is a reactive stream of string messages that are coming from the client. For that, we need to implement a source stream that accepts client connections and emits all messages that the client sends. The service can be implemented like this:

```
class service {
public:
    using value_type = std::string;

    explicit service(
            boost::asio::io_service& service,
            unsigned short port = 42042)
        : m_acceptor(service,
            tcp::endpoint(tcp::v4(), port))
        , m_socket(service)
    {
    }

    service(const service& other) = delete;
    service(service&& other) = default;
```

The service listens to the specified port (by default 42042)

```
    template <typename Cont>
    void set_continuation(Cont cont)
    {
        m_continuation = cont;
        do_accept();
    }

private:
    void do_accept()
    {
        m_acceptor.async_accept(
            m_socket,
            [this](const error_code& error) {
                if (!error) {
                    make_shared_session(
                            std::move(m_socket),
                            m_continuation
                        )->start();

                } else {
                    std::cerr << error.message() << std::endl;

                }

                do_accept();
            });
    }

    tcp::acceptor m_acceptor;
    tcp::socket m_socket;
    std::function<void(std::string&&)> m_continuation;
};
```

The service starts accepting connections only when the continuation that processes the incoming messages is defined

Accepting the next client

The `service` class implements the interface we required from all source streams to implement. Since it listens only to a single port, we will have to create a several instances if we want it to listen to several different ports. We have already created all the functions we need for this.

```
auto data_flow =
    values { 42042, 42043, 42044 }   The ports the service listens on

    | transform([&] (auto port) {          Gives us three streams
        return service(event_loop, port)   of messages - one for each
    })                                     specified port

    | join()   Merges the messages from all service instances
               into one single stream

    …
```

Now that we have a merged stream of messages coming from the clients connected to the service, we can consider what transformations on it we need to perform – what the data flow should look like:

- First, we need to normalize the messages – to perform the initial cleanup. We can trim the whitespace from the messages, since it carries no information, and skip all the empty messages;
- Next, we need to parse the JSON data;
- And then to read the bookmark information from it;
- For each parsed bookmark, we write it to the standard output.

This is the initial data flow. We have a source, and several transformations to perform on the messages that source emits.

```
auto sink_to_cout = sink([] (auto&& value) {
    std::cout << value << std::endl;
});
auto data_flow =
    …
    | transform(trim)                                  Normalize the input
    | filter(std::not_fn(&std::string::empty))         messages

    | transform(parse_json)
    | transform(json_to_bookmark)

    | sink_to_cout;
```

There is a problem with the previously defined flow. Parsing JSON-formatted strings and extracting the data from a parsed JSON object can fail – the input string might not be a valid JSON, or the JSON object might not contain necessary bookmark data. If we allowed these functions to throw exceptions when they encounter an error, this would stop our program because the exception mechanism in C++ can not work with reactive data flows. The exception mechanism can only unwind the function call stack and search for a function that has the try-catch block to catch the exception. In the data flow, every transformation is a separate function, and there is no traditional function call stack which means that one function in the flow can not catch an exception thrown by another.

Instead of using exceptions, the `parse_json` and `json_to_bookmark` functions will need to do the error handling in a functional way – they need to return the error as the result of the function. We can use `expected<T>` for this. This means that we no longer can compose two ordinary transforms since the second transformation function expects a JSON object, and we are passing an instance of `expected<json>`. We can fix this simply by replacing the `json_to_bookmark` function with its monadically-bound variant:

```
…
| transform(parse_json)
| transform(
    [] (const auto& optional_json) {
        return optional_json | mbind(json_to_bookmark)
    }
)
| sink_to_cout;
```

This composition of transformations will return an empty optional value if either parsing or data extraction failed. Otherwise, it will return an instance of the bookmark object. It behaves just like the usual C++ exceptions mechanism, but it works with reactive streams.

The only piece that is missing is replying to the client. The problem is that with the current design we are not able to reply to the client because the data flow does not know that the client even exists. The source stream emits messages without any information about who sent the message – it just contains the information about the message itself.

Instead of the service object sending messages containing only strings, it needs to send messages that contain strings, and information about the client we can use when replying:

```
template <typename MessageType>
struct with_client {
    MessageType value;
    tcp::socket* socket;

    void reply(const std::string& message) const
    {
        auto sptr = std::make_shared<std::string>(message);
        boost::asio::async_write(
            *socket,
            boost::asio::buffer(*sptr, sptr->length()),
            [sptr](auto&&, auto&&) {
                // We don't care about when the message
                // is sent to the client
            });
    }
};
```

The `with_client` is a generic type parametrized on one template parameter. We can easily define the `transform` function for it (it would transform the value while keeping the socket intact). This means that `with_client` is a functor, and we can leverage that in our web service.

We can change the service class to emit `with_client<std::string>` instances instead of plain strings. The transformations that we previously had in our data flow do not need to change – we still need to process the messages in the same way we did before. We only need to be able to make the previous data flow work with strings wrapped in the `with_client` functor. As explained in Section 2.5.2, if we have a function that works on a type, and we need a function that works on a wrapped type, we just need to lift it.

In this case, instead of lifting all functions one by one, we can just make `transform` and `filter` do it for us automatically. We can replace these functions with our own implementations:

```
auto transform = [] (auto transformation) {
    return ::transform(
```

```
        [=] (auto value_with_client) {
            return with_client {
                std::invoke(transformation,
                            value_with_client.value),
                value_with_client.client
            };
        }
    );
};

auto filter = [] (auto transformation) {
    return ::filter(
        [=] (auto value_with_client) {
            return std::invoke(transformation,
                               value_with_client.value);
        }
    );
};
```

This will allow us to keep the same data flow we had earlier, without a single line of the code changed in it, while allowing us to pass some extra information through the flow that is only needed in the sink:

```
auto data_flow =
    values { 42042, 42043, 42044 }

    | transform([&] (auto port) {
        return service(event_loop, port)
    })

    | join()
    | transform(trim)
    | filter(std::not_fn(&std::string::empty))

    | transform(json_parse)
    | transform([] (const auto& optional_json) {
          return optional_json | mbind(json_to_bookmark)
    })
```

```
    | sink([](const auto& bookmark_with_client) {
          const auto exp_bookmark = bookmark_with_client.value;

          if (!exp_bookmark) {
              message.reply("ERROR: Request not understood\n");
              return;
          }

          message.reply("OK: Bookmark accepted");
    });
```

This shows the power of abstraction that reactive streams have – we can think about the data flow for our software system, and model that flow using stream transformations. It gives us a system that is easy to reason about as the data flow diagram is directly mapped to the code, and it is easy to extend either by adding new transformations, or by composing the reactive stream monad with other monads like `expected<T>` or other.

4 Imperative reactive programming

4.1 Introduction

We saw the complexity that arises from the proper handling of asynchronous tasks in Chapter 2. We have demonstrated how asynchronous execution can be handled in the functional style to look like ordinary sequence processing in Chapter 3. But this is not enough because most C++ developers are not accustomed to thinking about software systems in terms of data flows, but rather in terms of algorithm steps – i.e. in the imperative style.

The purpose of activity diagrams is to graphically represent these algorithm steps – the algorithm itself, and organization of the subroutines in it. With asynchronous tasks and the inversion of control (IoC), we get code that has almost nothing in common with the corresponding diagram. The diagram that corresponds to the actual implementation becomes more like the activity diagram in the Figure 4.1, which represents only a part of the whole process).

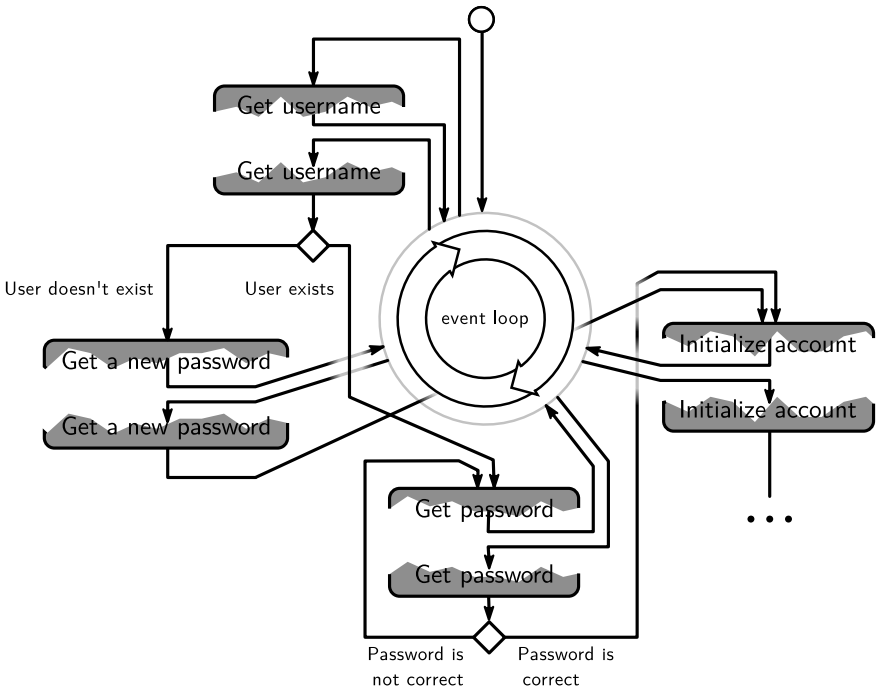


Figure 4.1: User authentication algorithm with IoC

Our aim is to make the necessary abstractions that will allow the programmer to write the code in an imperative manner – code that will implement an algorithm with a "1:1" correspondence between the steps in the algorithm's design and tasks in the actual code. The compiler should then be able to do all the heavy lifting

for us, and generate the equivalent IoC code behind the scenes. Taking the original problem, we want to be able to write something similar to the following, and get a proper asynchronous implementation:

```
function login:
    username = get_username(username)

    new_user = check_if_user_exists(username)

    if new_user:
        password = get_password()
        initialize_account(username, password)

    else:
        do:
            password = get_password()
        while (!check_if_user(username, password))

    initialize_environment()

    if new_user:
        show_welcome_message()
```

In a nutshell, we want to create the following:

- A single abstraction on top of different task types (both synchronous and asynchronous) that will be able to invoke a task and know when it has finished;
- Flow control structures that will use this abstraction to emulate the flow control structures of the host programming language, along with a few additional ones that are convenient for asynchronous programming;
- An embedded DSL (Domain-Specific Language) whose expressiveness on asynchronous code is comparable to that of the host language on regular code, but with lowered programmer-facing complexity.

Additionally, we want our solution:

- To make it easier to reason about asynchronous programs, and to make them easier to test and validate;
- To be efficient – to have no performance penalties at runtime.

4.2 Flow control structures

Just as the functional reactive programming approach to dealing with asynchrony presented in Chapter 3 was based on the continuation monad and its more relaxed variant – reactive streams, so the proposed imperative reactive programming (IRP) solution will be.

In the FRP approach, we used continuations and reactive streams to pass values from one stream or a stream transformation, to the next transformation. When a value appears in the source stream, it is then pushed via the continuation function to whichever object listens to values from that source stream.

In the imperative reactive programming approach, we can use continuations to build more complex chains of tasks and different kinds of flow control structures.

Instead of values being passed on from one asynchronous function to another directly (or, in the case of FRP, streams which generalize over asynchronous functions), the value returned by an asynchronous task will be handled and processed by task's executor which will be able to use the returned value by itself, or pass it on to another function.

This way we can build the flow control structures analogous to those provided by the programming language, but which can handle asynchronous execution without an explicit IoC. We can also build more advanced flow control structures that are more specialized for asynchronous execution. We will refer to the mechanisms that perform the task execution and result handling as *schedulers*. We will define those in a similar manner to "Lambda: The Ultimate Imperative" [SS76].

It is important to note that the schedulers can also be seen as tasks since they fulfil all the aforementioned requirements. This feature will allow us to nest schedulers into each other [Čuk13; Čuk14].

4.2.1 Serial execution scheduler

The simplest form of execution is the serial execution. In essence, it is analogous to a series of commands written one after another. With an exception that it implements the short-circuit logic like the *conjunction operator* (`&&`) in C++ (and other C-like languages) – it stops execution after the first task that has failed. In that case, it is considered that the scheduler task also failed. Otherwise, the task has succeeded.

Let us demonstrate this using the famous coffee preparing algorithm. It compares the synchronous (blocking) C++ implementation with the equivalent asynchronous (non-blocking) implementation using the imperative reactive programming (IRP for

short) approach. The only difference, apart from a bit changed syntax, is that the ordinary C++ version needs to employ exception handling in order to stop the execution if a task fails:

```
C++                                     C++/IRP

// Definition                           // Definition
void make_coffee()                      auto make_coffee = serial_
{                                       (
    try {
        fill_the_pot_with_water();          fill_the_pot_with_water(),
        turn_on_the_cooker();               turn_on_the_cooker(),
        place_the_pot_on_the_cooker();      place_the_pot_on_the_cooker(),
        wait_for_the_water_to_boil();       wait_for_the_water_to_boil(),
        put_coffee_in_the_water();          put_coffee_in_the_water(),
        mix_a_bit();                        mix_a_bit(),
        pour_into_the_cup();                pour_into_the_cup()
    } catch (...) {}
}                                       )

// Invocation                           // Invocation
make_coffee();                          make_coffee();
```

Implementing this behaviour is straight-forward. The scheduler needs to execute the first task in the list, and schedule the continuation function for that task to execute the next task, and so on. More precisely, it needs to perform the following steps:

1. If the list is empty, returns `success` ($\top$)
2. Execute the head of the list
3. When it finishes:
    - if the task failed: end execution and return `failure` ($\bot$)
    - if the task succeeded: recursively call the algorithm for the tail of the list – executing the rest of the provided tasks

4.2.2 Logic operator schedulers

The serial execution scheduler behaves like logical conjunction, so we can introduce an alias `and_` and conjunction operator (`&&`) for it, for convenience when used in

logical expressions. Disjunction can be similarly implemented. The only difference is that instead of exiting when a task fails, it would exit when the first task succeeds:

1. If the list is empty, returns `failure` ($\bot$)
2. Execute the head of the list
3. When it finishes:
   - if the task succeeded: end execution and return `success` ($\top$)
   - if the task failed: recursively call the algorithm for the tail of the list

Apart from conjunction and disjunction, to provide a full expressiveness of propositional logic, we need to add the schedulers which correspond to the constant logical operators (that always return `success`, or `failure`) and negation (which returns `success` iff the task passed to it failed). All these schedulers get one task as argument, which is always executed.

`true_`

1. Execute the task
2. When it finishes:
   - return `success` ($\top$)

`false_`

1. Execute the task
2. When it finishes:
   - return `failure` ($\bot$)

`not_`

1. Execute the task
2. When it finishes:
   - if the task succeeded: return `failure` ($\bot$)
   - if the task failed: return `success` ($\top$)

Figure 4.2: Logical constants and negation

These can be particularly useful when reporting warnings and errors to the user, or when we want to ignore an error that occurred in a specific task. Let us get back to the coffee preparing example. Imagine that the task `turn_on_the_cooker` failed – it would be good if we could show an error message to the user, since we are unable to continue after that error:

```
fill_the_pot_with_water(),
or_(turn_on_the_cooker(), false_(show_message("No electricity"))),
place_the_pot_on_the_cooker()
```

The `show_message` task will succeed, but we will ignore its result and pretend that it has failed because the `false_` scheduler always returns `failure` ($\bot$).

If we make a dedicated task `die_` that shows an error message and always returns `failure` ($\bot$) (a composition of `false_` and `show_message`), and use the disjunction operator (`||`) instead of `or_`, we can even simulate constructs from some other popular imperative programming languages like Perl or PHP:

```
fill_the_pot_with_water(),
turn_on_the_cooker() || die_("No electricity"),
place_the_pot_on_the_cooker()
```

On the other hand, if we do not have a spoon, and the task `mix_a_bit` fails, we do not need to kill the whole process. We can just specify that we want to ignore that task by wrapping it inside the `true_` scheduler:

```
put_coffee_in_the_water(),
true_(mix_a_bit()),
pour_into_the_cup()
```

Modelling `true_` and `false_` to be functions and to accept a task they need to execute before returning `success` ($\top$) and `failure` ($\bot$) respectively gives us greater expressiveness compared to modelling them as simple constants. If the need for proper constants arises at some point, they are easily implemented by passing an empty `serial_` scheduler to the already defined `true_` and `false_` schedulers.

4.2.3 Branching scheduler

By using `serial_` and `or_` we can only simulate the one-branch `if` statements (those that do not have both then and else branches). In order to create a full replacement, we need to create a new scheduler that receives three subtasks – one which will be used as the predicate, one for the consequent, and one for the alternative branch:

```
if_(predicate)
(
    consequent
).else_(
    alternative
)
```

The behaviour of `if_` can be written as follows:

1. Execute the `predicate` task
2. When it finishes:
    - if the task succeeded: Execute the `consequent` task
    - if the task failed: Execute the `alternative` task. If it does not exist, return `failure` ($\bot$);
    - When any of above two finishes, end execution and return the status it returned.

This is the same behaviour of if-then-else statements in functional programming languages, or the ternary operator `(?:)` from the C++ language.

Looking at our starting example of user authentication (Figure 2.3), we have different code paths handling a new user compared to an already existing one. It can be implemented like so:

```
if_(new_user) (
    // Creating a new user
    password = get_password(),
    initialize_account(username, password)
).else_(
    // Authenticating an existing user
    …
)
```

4.2.4 Loop schedulers

While and do-while loops can also be easily defined. The `while_` loop scheduler gets a list of tasks – the first is used as the predicate, and the rest are serially executed. If any task, except for the predicate, fails, the loop ends with the `failure` ($\bot$) status. If the predicate fails, the status of the loop is reported as `success` ($\top$).

The do-while (`do_`) is essentially the same, except that the condition is the last task in the list, and thus the last to be executed. Again, the loop returns `success` ($\top$) iff the condition task fails.

By allowing the tasks to stop the loop, we are giving the user a mechanism similar to that of the `break` keyword of C++.

In the example of user authentication, we have a part where the user is asked for the password until the correct one is entered. This fits the do-while scheduler

perfectly. It is worth noting that if the user does not provide a password, but cancels the process, the `get_password` task can simply return `failure` ($\perp$) and the loop will exit:

```
do_(
    password = get_password(),
).while_(
    !check_if_user(username, password)
)
```

The `for_(init, condition, increment)` loop can be easily implemented on top of `while_` and `serial_` schedulers.

While this approach does not deal with the `continue` statement, only `break`, it can be easily implemented either by assigning different meanings to different task return values (instead of just 0 for success, $\neq 0$ for failure), or by using a more nuanced error reporting mechanism based on the exception monad (Section 2.5.3.3) where we would use different error types for `break` and `continue`. This would also allow for defining a complete exception-like mechanism and error propagating, but that is out of the focus of this dissertation since the principle of defining schedulers would remain the same.

4.2.5 Chaining scheduler

As we have mentioned in Chapter 2, C++ does not allow overloading the dot operator (as in `a.b().c()`), so it is not possible to write a scheduler that will be able to chain the tasks with that syntax. This can be remedied in a similar manner to how we defined transformation chaining on reactive streams in Chapter 3 – by using the pipe syntax:

```
formatted_name = get_username() | get_full_name() | to_html();
```

This example shows that the result of one asynchronous operation can be passed as an argument to another one (from `get_username` to `get_full_name`) or from an asynchronous to the synchronous one (from `get_full_name` to `to_html`).

There is an important difference between the tasks that can be chained and ones that can be passed to other schedulers. Other schedulers require the task to return the status whether they succeeded or not, whereas a task that is chaining needs to return an actual value because that value will be passed to the next task as

an argument. These tasks are delimited continuations [Fel88], they can be invoked multiple times, and the resulting values used later.

The behaviour of chaining `head` and `tail` can be described simply as:

1. Execute the `head` task
2. When it finishes, execute `tail` and pass it the result returned by the `head` task
3. When `tail` finishes, return whether it succeeded or not

This behaves the same as a monadic bind function of the continuation monad when composing asynchronous tasks, and like the `transform` function when composing the synchronous tasks.

One nuance about the chaining scheduler that might not be as obvious is that in the previous code snippet, the `operator=` is also a form of a chaining scheduler – just a specialized one. Namely, the `operator=` scheduler just passes the result of the chained asynchronous operation (when it is finished) to the `operator=` function of the variable on its left-hand side.

If error handling is required for chaining as well, it can easily be done with the tasks returning `optional<T>` or even `expected<T>`.

4.2.6 Basic schedulers demonstration

The schedulers that we defined so far allow us to write the asynchronous code using the same principles and workflows as used in writing the ordinary imperative or functional code that uses the ordinary C++ flow control structures. It makes a full mapping between the programming concepts commonly used when designing an algorithm or a data flow, and a concrete implementation without the need for explicitly dealing with the inversion of control.

We have seen how reactive programming can make implementing asynchronous systems in the functional style in Chapter 3 easier. We are now going to demonstrate how we can benefit from the imperative style of reactive programming presented in this dissertation. Now we will show it on the example from the Section 4.1. With the schedulers we previously defined, we can implement this example like so:

```
auto login = serial_(
    username = get_username(),

    new_user = check_if_user_exists(username),
```

```
    if_(new_user) (
        password = get_password(),
        initialize_account(username, password)
    ).else_(
        do_(
            password = get_password(),
        ).while_(
            !check_if_user(username, password)
        )
    ),

    initialize_environment(),

    if_(new_user) (
       show_welcome_message()
    )
);
```

This is a line-by-line equivalent of the pseudo-code example we started with where the only difference is the different syntax. C++ imposes a lot of restrictions regarding operator overloading and definitions compared to some other languages like Haskell or Scala. This limits the ability to create a domain-specific language without any undesired artefacts. In this case, the artefacts are limited to using normal parentheses instead of braces, keywords with a trailing underscore (because the original ones are reserved), and having the second part of compound statements like `do_/while_` and `if_/else_` prefixed with a dot (because they need to be implemented as methods of the first one).

4.3 Advanced schedulers

These were rather rudimentary schedulers which mimic the flow control structures that the imperative languages provide for invoking synchronous tasks. We can also define more complex schedulers to match some of the common asynchronous programming patterns and good practices.

4.3.1 Transaction scheduler

There are many systems where a batch of commands needs to be executed successfully from the start to the end, or not be executed at all. This means that if any of the commands fail, the program and the data need to be restored to the same state as before the first command was run.

We can use a variant of the command pattern [Gam+95; Fre+04] where the command (a task in our terminology) also knows how to undo itself (has `do()` and `undo()` member functions). The transaction scheduler receives a list of tasks that should be executed sequentially, same as the serial scheduler. The behaviour can be written as follows:

1. Execute the list head
2. When it finishes:
    - if the task succeeded: Execute the tail recursively
        - if the tail execution succeeded: return `success` ($\top$);
        - if the tail execution failed: Execute the `undo` member function of the list head, and return `failure` ($\bot$).
    - if the task failed: return `failure` ($\bot$).

In order to demonstrate the transaction scheduler, let us consider an example of file copying. The requirement is that at the end of the operation, the file system should not be changed unless the file has been completely copied:

```
transaction_(
    temp = generate_temporary(destination),
    copy(source, temp),
    rename(temp, destination)
)
```

The first call creates a new temporary file (we can not copy directly to destination since it can already exist, and we would overwrite it even in the case copying fails). Then we copy the data, and rename the temporary file to destination.

Let us analyze the failure cases, and make sure that the original `destination` file is untouched, and the temporary is deleted in all of them.

- If `generate_temporary` fails to create a temporary file, we are exiting the transaction – nothing has been done, we have not changed anything;

- If `copy` fails, we start undoing previous commands – and the temporary file gets deleted by calling the `undo` member function of `generate_temporary`;
- If renaming fails, we are calling `undo` on `copy`, which does not need to do anything, and then on `generate_temporary` which deletes the temporary file.

There are two requirements that the tasks passed to the transaction scheduler must fulfil in order for it to be able to guarantee execution correctness:

1. If the task execution fails, it needs to clean up after itself before returning `failure` ($\bot$). This means that we do not need to call undo on a failed task.
2. The task needs to be designed so that the `undo` member function must always succeed (in the previous example, the temporary file deletion must never fail). If the `undo` member function fails, it is an unsolvable scenario[Ale12] so we are not even trying to handle it.

It is trivial to prove that the scheduler itself fulfils these requirements if the contained tasks fulfil them.

4.3.2 Fork execution scheduler

The main purpose of asynchronous programming is the ability to execute multiple tasks at the same time. The previous schedulers did execute the tasks in an asynchronous manner, but they did not execute the tasks that belong to them in parallel. For that, we need a new scheduler type – `fork_`. It takes a list of tasks, and performs the following:

1. Execute the list head
2. Execute the tail, in parallel using the same algorithm
3. When the list head finishes: register whether it succeeded or not
    - if the tail execution has already finished: Return `success` ($\top$) if both the tail and the head finished successfully. Otherwise return `failure` ($\bot$).
    - if not: Register that the head has finished, and register whether it succeeded or not.
4. When the tail finishes:
    - if the head execution has already finished: Return `success` ($\top$) if both the tail and the head finished successfully. Otherwise return `failure` ($\bot$).
    - if not: Register that the tail has finished, and register whether it succeeded or not.

This way, the fork execution scheduler waits for all the child tasks to finish, and returns `success` ($\top$) only if all tasks succeeded. The behaviour can be optimized by remembering only the total number of running child tasks, which will be shown in the Chapter 5.

In order to demonstrate the fork scheduler, we will use a slightly modified example of the user authentication problem. In the case where the user did not exist previously, we are calling the `initialize_account` function. We might break that function into three different parts – adding the user to the system, creating a directory for the user's data directory and assigning ownership to that directory. The first two can be performed in parallel, while the last one needs to wait for them to be completed:

```
if_(new_user) (
    password = get_password(),
    fork_(
        create_user(username),
        directory = create_directory_for_user(username)
    ),
    change_ownership(username, directory)
)
```

4.3.3 Detaching execution scheduler

The alternative approach to parallelism can be taken in the cases where we do not need to know when the child tasks have finished (such as logging or displaying messages to the user). For that case, we can define a simple scheduler `detach_` that starts all the tasks in the list and immediately returns `success` ($\top$) without waiting for the children tasks to finish, similar to `ft_thread_unlink` from the FairThreads library [Bou06].

A fitting example to demonstrate detaching would be a web service that accepts client connections and processes each client separately:

```
while_(client = asio::server::accept(server)) (
    detach_(process_client(client))
)
```

4.4 Data lifetime

There are a few popular methods of communication between parallel processes. Most notably, message passing and tuple spaces [Car+94]. We take a new, mostly orthogonal approach. One reason is to avoid duplication – there are existing implementations for the above mentioned solutions that can be used with the presented IRP schedulers (like CppLINDA[1]). The second, and more important reason, is that using a foreign syntax and a totally different paradigm would spoil the idea of having the code as similar as possible to the normal imperative code.

We are going to use the same model of scoped variables that the C++ language uses [ISO17, basic.scope]. A variable exists only in the scope where it is defined. It is created when the execution enters that scope, and it is destroyed when the execution exits the scope. In the following example, which does some calculations on a large data object, the variable `data` exists only while the function is being executed:

```
void process()
{
    sound_wave data;
    amplify_sound(&data); // we are passing a reference to the structure
                          // because it is inefficient to copy it
}
```

Although we will follow the same model, we can not rely on ordinary variables. Imagine that the `amplify_sound` function from the example is asynchronous. This means that the caller will get the control immediately after calling it and the variable (along with the data) will likely be destroyed before `amplify_sound` ends. This would lead to memory corruption. If we allocate `data` dynamically, the functions we call on it would need to know which one should free it which would complicate the APIs.

We will have to think of scopes in a different way. The variable is created when the execution enters the scope. But it is destroyed only when all its users finish executing. For this to work properly, we will have to use the `std::shared_ptr` introduced with C++11 [Str13, p. 5.2.1]:

```
auto process_task()
{
    auto data = std::make_shared<sound_wave>();
```

[1] Linda for C++, `http://sourceforge.net/projects/cpplinda/`

```
    return serial_(
        data = load_audio_task(),
        amplify_sound_task(data),
        highpass_filter_task(data)
    );
}
```

The additional advantage of this approach compared to the tuple-spaces is the lack of the need for garbage collection. Variables exist only while they are needed and are automatically disposed of afterwards.

5 Implementation

5.1 The event loop

The usual rule in C++ is not to introduce any runtime performance penalties by default. Introduce them only when the programmer uses a specific feature that requires them. The classic examples are exceptions and virtual member functions. If you do not use them, your code will be at least as performant as a C program. The implementation of reactive programming scheduler follows this principle. When possible, the compiler needs to be able to optimize out all the used abstractions.

Thanks to Alexandrescu and his Loki library [Ale01] the C++ compilers are now dealing efficiently with the generic code and template meta-programming (TMP) structures. The TMP is a Turing-complete language [Cza02] which is executed by the compiler itself, while it compiles the code. As mentioned in Section 2.4, we will use TMP mainly for code generation and static introspection.

The system that we want to create is internally based on message passing which will allow us to be notified when an asynchronous task is finished. In particular, on the centralized event loop where the client can register a handler for a specified type of event. The message will then be processed, and the task scheduler will be able to continue the execution with the next task.

The event loop, or message dispatcher, is a sub-system that listens for and dispatches events or messages. It usually resides in the main thread and serves as a universal communication mechanism between objects in a process. It is often used for synchronization, even in multi-threaded systems.

Popular C and C++ frameworks (for example GLib, Qt or Boost) provide their own implementations of signal handling (with Boost requiring more work to allow signals across threads by using `boost::asio::io_service`). In order to implement the scheduler system in a generic way, we need to abstract out the event loop so that the system can be used with any of the aforementioned libraries.

The abstract interface for the event loop contains only one type definition and four functions:

```
// Alias type for a generic function container
// that takes a pointer to the task that sent the signal,
// along with the return status of its execution
// (0 for success and any other value for failure)
using signal_handler = std::function<void (void *, int)>;
```

```
// Defines a function that should handle when
// the task (sender) has finished execution
void connect(void * sender, signal_handler h);

// Disconnects a task from its handlers
void disconnect(void * sender);

// Adds a new event that needs to be handled to the queue
// the arguments are the task that sent the event
// and the execution status
void emit(void * object, int status);

// Processes collected events
void process_events();

// Starts the event loop
[[noreturn]] void loop();
```

This API is quite simple. The following example shows how a connection is established. It connects the sender with a lambda function:

```
signal::connect(sender,
    [this] (void * task, int error) {
        // process event
    }
);
```

Using void pointers is highly discouraged in C++ because it breaks the type safety. It is not a problem in this case since we are using them only as unique identifiers of senders. There is no type-casting from void pointers to some more specific pointer type.

This is a simple interface that can easily be implemented on top of the Qt or GLib libraries. For example, in the case of Qt, it would be sufficient to create a new event type (`IRPEmitEvent`) and register the `process_events` function as its handler. The `emit` function would just send a `IRPEmitEvent` event via `QApplication::sendEvent()` member function. The `loop` function could be left empty (a no-op), because the application would have started Qt's event loop anyway. Any time the `emit` function is called, the `process_events` function will process any unprocessed events in the queue.

5.2 Tasks and task factories

As previously specified, a task in our nomenclature covers any callable piece of code for which we know when it has finished. In this case, "callable" means that it has the nullary function call operator (the `operator()` with no arguments) [Str13, p. 3.4.3] or a `start()` member function. If the task is synchronous, the task should return an integer value that represents the execution status. In the case of an asynchronous task, the task object needs to have a member function `set_on_end` which takes one `signal_handler` argument. This member function registers a callback function that should be invoked when the task finishes execution. Examples of these can be seen here:

```
int function()
{
    // do something
    …

    // we have finished
    return EXIT_SUCCESS;
}


class SyncFunctionObject {
public:
    int start()
    {
        // do something
        …

        // we have finished
        return EXIT_SUCCESS;
    }
};
```

```
class AsyncFunctionObject {
public:
    void operator() ()
    {
        // do something
        …

        // we have finished
        m_end_handler(this,
                EXIT_SUCCESS);
    }

    void set_on_end(
        signal_handler end_handler)
    {
         m_end_handler = end_handler;
    }

private:
    handler m_end_handler;
};
```

5.2.1 Introspection

In order to provide the most generic interface, we need to use the compile-time type introspection to examine the type of the callable object passed to the scheduler at

the time of compilation, instead of using the inheritance-based polymorphism which would require the users to subclass a common super-class and would have runtime performance penalties.

We can define a convenient macro which instantiates a SFINAE-based (Section 2.4.2.2) structure which is able to test for member function existence during compilation time:

```
#define declare_memfun_tester(Name, MemFun, Parameters)           \
    template<typename T>                                          \
    struct Name {                                                 \
        template<typename U, void (U::*)Parameters>               \
        struct test_struct {};                                    \
                                                                  \
        template<typename U>                                      \
        static std::true_type test(test_struct<U, &U::MemFun> *); \
                                                                  \
        template<typename U>                                      \
        static std::false_type test(...);                         \
                                                                  \
        using type = decltype(test<T>(0));                        \
        static const bool value =                                 \
            std::is_same<type, std::true_type>::value;            \
    }
```

5.2.2 Conversion to continuations

In order to implement continuations, we need to be able to differentiate between various types of callables. We need to be able to tell whether the passed callable is a synchronous function, an asynchronous task, or it might be even a factory function that returns an asynchronous task.

For this, we will implement a simple compile-time test which returns whether a type has the needed requirements to be an asynchronous task. The test returns true if the passed object is a function object (it has a call operator), and has the `set_on_end` member function:

```
declare_memfun_tester(has_apply, operator(), ());
declare_memfun_tester(has_start, start, ());
declare_memfun_tester(has_set_on_end, set_on_end, (signal_handler));
```

```
template<typename T>
struct is_async_task {
    static const bool value =
        (has_apply<T>::value || has_start<T>::value)
        && has_set_on_end<T>::value;
    using type = typename std::integral_constant<bool, value>;
};
```

Beside ordinary tasks, we will also support lazy creation of tasks by using task factories. This can be useful when the task object takes a significant amount of memory, and is rarely invoked. If a scheduler receives a function whose return type fulfils the requirements for a task (`is_async_task<T>::value` returns `true`), it first calls the factory function and then executes the task.

5.2.3 Serial scheduler

Now that we know the task types during compilation, we can start implementing the control structures – schedulers. The schedulers differ mainly in what they do when the task finishes execution, so we will cover only `serial_` in detail. As previously specified, it takes a variable number of tasks to execute. For this to work, we need to use variadic templates – templates with variable number of arguments (Section 2.4.3).

Dealing with variadic templates in C++ is similar to defining a function over a list in FPLs. We start by defining the behaviour for the empty list, and then cover the case where we have a head and a (possibly empty) tail. In order to keep the code simpler, we will require tasks to have the function call operator and not the `start()` member function:

```
template<>
class serial_scheduler<> {
public:
    serial_scheduler() {}

    void operator () ()
    {
        on_end_handler(this, EXIT_SUCCESS);
    }
```

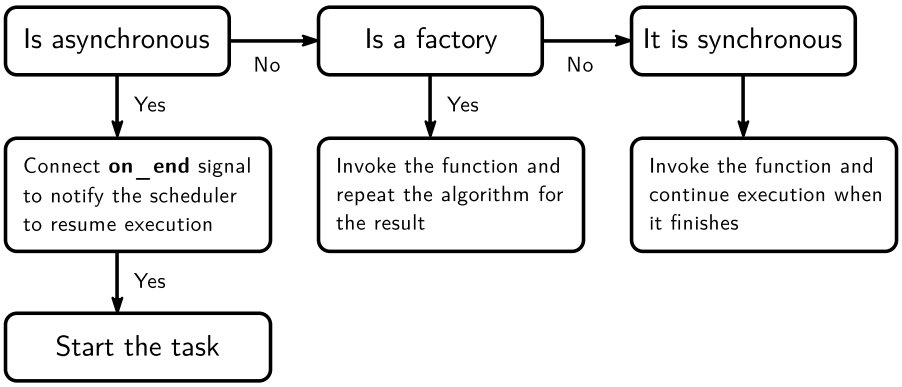


Figure 5.1: Task execution process

```
    void set_on_end(signal_handler handler) {
        // Nothing to do - no tasks to execute
    }
};
```

The algorithm in the Figure 5.1 represents the logic behind the execution of a task. It detects the task type and executes the task. This logic is used in most task schedulers in the library, so we are going to present it separately.

The main difference between schedulers is what they do when the previously started task has finished executing. The `serial_` scheduler just continues executing the rest of the task list. In this case, it is equivalent to invoking the function call operator (`operator()`) of the super-type (remember that processing variadic template arguments is done in the same manner as recursive list processing – process the head, and then invoke the algorithm on the tail of the list:

```
template<typename _Task, typename ... _OtherTasks>
class serial_scheduler<_Task, _OtherTasks ...>
    : public serial_scheduler<_OtherTasks ...> {

private:
    using this_type  = serial_scheduler<_Task, _OtherTasks ...>;
    using super_type = serial_scheduler<_OtherTasks ...>;

public:
    serial_scheduler(_Task task, _OtherTasks ... others)
        : super_type(others...), m_task(task) {}
```

```
    void operator () ()
    {
        // we are testing whether the head of the list
        // is an asynchronous task and we start it
        using is_async_task =
                typename traits::is_async_task<_Task>::type;
        _start(is_async_task());
    }

private:
    // Overloaded version that can execute synchronous tasks
    void _start(std::false_type) {
        // extraction of the function's return type
        using stripped_type =
            typename std::remove_pointer<
                typename std::remove_reference<
                    decltype(m_task())>::type
            >::type;

        // checking whether the return type is an asynchronous
        // task or a function
        using is_task_class =
            typename traits::is_task_class<stripped_type>::type;

        _start_factory_or_function(is_task_class());
    }

    // Overloaded version of the function that can start
    // asynchronous tasks
    void _start(std::true_type) {
        _start_task(m_task);
    }

    // Overloaded function that handles starting
    // of a task from the factory
    void _start_factory_or_function(std::true_type)
```

```
{
    using result_type   = decltype(m_task());
    using stripped_type = typename std::remove_pointer<
            typename std::remove_reference<
                decltype(m_task())>::type
        >::type;

    stripped_type task =
        dereference_if_pointer<result_type>::deref(m_task());

    _start_task(task);
}

// Overloaded function that executes the task function
// and immediately continues execution of other
// tasks in the scheduler
void _start_factory_or_function(std::false_type)
{
    if (m_task() == EXIT_SUCCESS) {
        super_type::start();
    } else {
        on_end_handler(this, EXIT_FAILURE);
    }
}

// Function that connects the task's on_end signal
// and starts the task
template<typename T>
void _start_task(T && task)
{
    signal::connect(&task, [this] (void * task, int status) {
        if (status == EXIT_SUCCESS) {
            this->super_type::start();
        } else {
            this->on_end_handler(this, status);
        }
```

```
        });

        task.set_on_end([] (void * task, int error) {
            signal::emit(task, error);
        });

        task();
    }

    … // on_end_handler related member functions

private:
    _Task m_task;
};

// Convenience member function which forces the compiler
// to automatically deduce the template parameters
template < typename ... _Jobs >
serial_scheduler < _Jobs ... >
serial_ (_Jobs ... args) {
    return serial_scheduler < _Jobs ... > (args ...);
}
```

5.3 Implementation efficiency

The system needs to have no runtime overhead compared to the regular call-callback C++ code. It can not outperform the former, but it should be as efficient. We are going to show that the compiler is able to optimize-out the introduced abstractions and generate the same code as it would in the case of call-callback approach.

The following is a side-by-side display of original C++ code with the generated assembly code[1] when invoking synchronous functions (Listing 5.2). It shows that the `serial_` scheduler (and the continuation-continnuator abstraction with it) generates no overhead compared to the normal C++ code when calling functions. Even more, if functions are declared to be `inline`, the compiler is able to inline them even when using the scheduler. A similar output is created when testing the other schedulers.

[1] generated by GCC version 4.7 with the following flags: `-std=c++0x -g -Wa,-a,-ad -O3`

```
function1();                call _Z9function1v
function2();                call _Z9function2v
inlineable_function();      … // no call instruction

serial_(
    function1,              call _Z9function1v
    function2,              call _Z9function2v
    inlineable_function     … // no call instruction
)();
```

Figure 5.2: Assembly code generated for synchronous functions

When calling the asynchronous functions from the scheduler, the compiler also generates the same assembly code as if we implemented the program using the call-callback method. It is interesting to see that, when we pass a function that has a callback function as its argument to the scheduler, that executing the scheduler calls only the former function, and automatically passes the continuation function to iy. This is demonstrated by the following example in which the `calculate_async` function performs some calculation asynchronously, and calls its argument when it finishes:

```
calculate_async(continuation);   mov  edi, OFFSET FLAT:_Z12continuationv
                                 call _Z15calculate_asyncPFvvE

serial_(                         mov  edi, OFFSET FLAT:_Z12continuationv
    calculate_async,             call _Z15calculate_asyncPFvvE
    continuation
)();
```

The fact that the assembly code is the same, both when calling ordinary functions and asynchronous ones, implies that our abstractions do not introduce any performance overhead by themselves. The only possible overhead of the system is due to the usage of the event loop for dealing with asynchronous calls which is anyhow a common element in all modern systems.

In order to test this statement in practice, we have tested the throughput of a single-threaded web-socket echo server implemented with Boost.Asio, with analogous version in which the IRP approach was used. The second version was also based on Boost.Asio, but with its functions wrapped into tasks that can be passed to the IRP schedulers.

The Web Socket protocol has a limit of 125 bytes for a base message which can be sent in a single frame. If a message is larger, it is split into multiple frames. In order to test the system in the real-world scenario, we have sent messages of different sizes – from small JSON documents, to larger images going up to 4MiB.

The benchmarking process went as follows:

- Start the server;
- Start a sufficient number of clients and load the data without connecting to the server (sufficient number to contain at least 500MiB of data);
- Trigger the clients to connect to the server;
- Measure the amount of data sent by the server in a single second.

By pre-starting and pre-loading the clients, we have removed the possibility of slow clients tainting the results of the benchmark.

The tests were performed both with servers compiled with GCC 5.2 and Clang 3.6 compilers (without client recompilation). The initial results with GCC were 154 MiB/s for the pure Boost.Asio version, and 147 MiB/s for the IRP version. While this is still a good result, the performance hit is noticeable (even more so with Clang, see table 5.1).

After profiling the code, it was shown that the main performance issue that exists only in the IRP version is due to its usage of `std::shared_pointer` to pass the data around (see Section 4.4). The shared pointer keeps a reference count which needs to be incremented and decremented atomically whenever a new copy is created, or an instance is destroyed. The pure Boost.Asio version used global variables and references to them. When the IRP variant of the server was altered to use global variables and pass references to them to the tasks, it reached similar performance to the pure Boost.Asio version when compiled with GCC. The Clang version also improved, but was still slower than the pure Boost.Asio version. This implies that Clang misses some optimization opportunities, and that the performance of the IRP solution depends significantly on the compiler used which is true for any template-heavy code.

Service version	GCC 5.2	Clang 3.6
Pure Boost.Asio version	154 MiB	137 MiB
Initial IRP version	147 MiB	128 MiB
IRP without shared pointers	154 MiB	130 MiB

Table 5.1: Echo server throughput

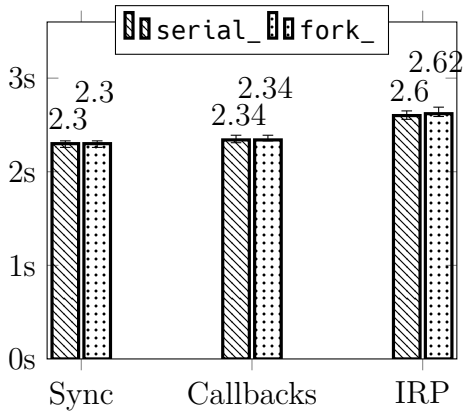
This was exactly our aim – to have a nicer way to write code, but to still have programs that are as efficient as the usual call-callback code.

5.4 Compilation times

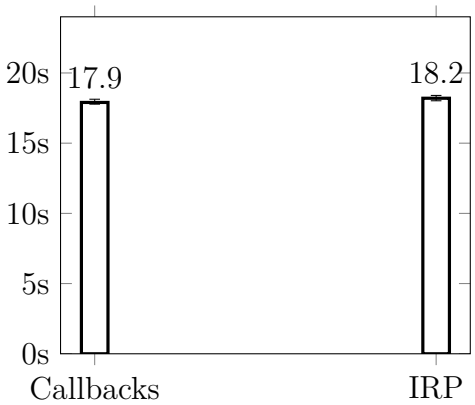
Due to the heavy usage of template meta-programming, the scheduler mechanism has a negative impact on the compilation times. In order to measure how much, we have performed a few benchmarks.

We have measured the speed of compiling `serial_` and `fork_` schedulers separately, compared that to the compilation speed of normal synchronous code, and asynchronous code that implements the same logic using calls and callbacks. In order to get the clearest picture of how big is the performance hit of using the schedulers, and we did not compare against the `std::future<T>` and similar classes since they are also template-based. These tests have shown around 11% hit for the `serial_` scheduler, and 12% for the `fork_` (Figure 5.3 (a)).

The second thing we measured is the performance impact on a full build of a real project – a simple web-socket server. While the previous tests were meant to show how much time the compiler needs to compile the IRP control structures, this one is meant to test what is the impact on the full build (compilation + linking) of a project that uses real-world libraries like Boost.Asio. In this case, the slowdown was 2% (Figure 5.3 (b)).



(a) Compilation of `serial_` and `fork_` schedulers



(b) Full build time of the echo web-socket project

Figure 5.3: Compilation times (in seconds)

6 Discussion and conclusion

6.1 Case studies

6.1.1 A simple echo server

As the first example, we will cover a single-threaded implementation of a simple echo server with a WebSocket-like handshake. The server should at all times be listening to incoming connections. It means that we can not wait for one client to be disconnected before we are ready to serve the next one. We will branch out and detach processing the connection as soon as we receive the request. The implementation uses the Boost.Asio [Koh] library wrapped so that its functions can be used as tasks:

```
asio::server server;
asio::client_ptr client;

// We want to wait until we get a connection.
while_(client = asio::server::accept(server)) (
    detach_(
        // Start a detached execution path to process the client.
        [] {
            // Defining local variables that are needed
            // for communication with the client
            asio::client_header_ptr header;
            asio::message_ptr message;
            asio::key_ptr server_key;

            serial_(
                // WebSocket handshake
                asio::client::get_header(header),
                server_key = asio::server::create_key(items),
                asio::client::send_header(client, server_key),

                // Sending the initial greeting message
                asio::client::message_write(client, "Hello, I'm Echo"),

                // Connection established
                while_(message = asio::client::message_read(client)) (
```

```
                    // echoing the input
                    asio::client::message_write(client, message)
                )
            )();
        }
    )
)
```

It is important to note that the existence of `detach_` does not necessarily imply creating a separate thread for each connection. In fact, it does not in the default implementation. The whole service is able to handle multiple connections in a single thread. Due to the level of abstraction, it is easy to alter the behaviour of `detach_` to spread the load on multiple threads, if the need arises. Our example will continue working without changing a single line of code – after all, that change is an implementation detail and not a part of our algorithm.

6.1.2 Secure project management service

In this example, which is based on the KActivities[1] project, but simplified, we will dive deeper into the comparison between the explicit IoC and the IRP approach. Imagine the following scenario – we have a system to manage a set of projects. The user can work on only one project at a time and some projects are private and encrypted with a password. When switching to a private project (Figure 6.1), the user is asked for the password. If the project was successfully decrypted, it is loaded and the environment adapts to it. If the decryption failed, the user is repeatedly asked for the password until she enters the correct one or cancels the procedure. As before, all the steps in the algorithm should be non-blocking.

When using the standard call-callback approach for asynchronous programming, we need to break apart the algorithm into a set of small methods that are not logical sub-parts of it. We also need to move all the variables that we do not want to carry from call to call into the global namespace:

```
void request_switch_to(string project)
{
    // Setting a global variable that holds
    // the project to which we want to switch
    global::new_project = project;
```

[1]KDE Activities Framework `https://projects.kde.org/projects/frameworks/kactivities`

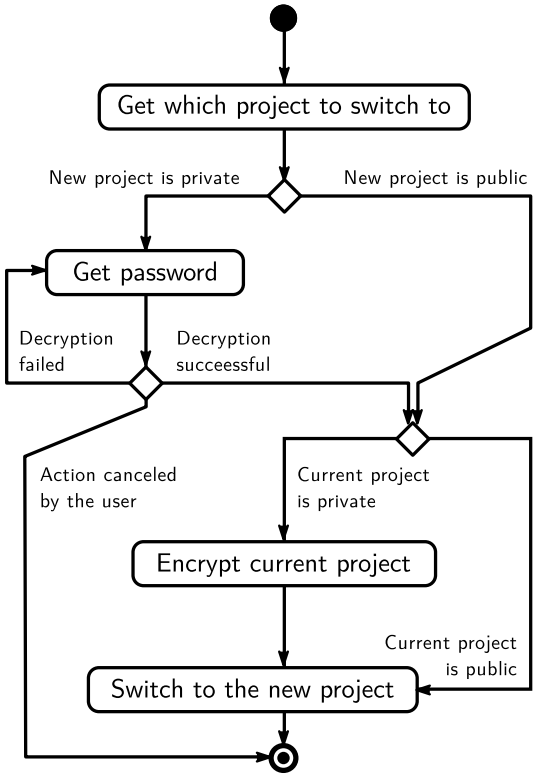


Figure 6.1: Switching projects algorithm

```
    is_private(global::new_project);
}

void is_private_callback(bool result)
{
    if (!result) {
        // Not a private project, we can switch
        // to it without problems
        switch_project();

    } else {
        // Requesting the service to ask the
        // user for the password and to try
        // to decrypt the project
        request_decryption(request_decryption_callback);
    }
}
```

```
void request_decryption_callback(bool success, bool canceled)
{
    if (canceled) {
        // If the request is canceled, cancel
        // the whole procedure
        global::new_project.clear();

    } else if (success) {
        // If the decryption is successful,
        // we can switch to the project
        switch_project();

    } else {
        // If the decryption failed,
        // we need to ask the user for the
        // password again
        is_private_callback(true);
    }
}

void switch_project()
{
    // First, we need to encrypt the current project
    // if it is private
    is_private(global::current_project,
            is_current_project_private_callback);
}

void is_current_project_private_callback(value bool)
{
    if (value) {
        // We will send the request to encrypt
        // the current project, but we will not
        // wait for it to be finished
        encrypt(global::current_project);
    }
```

```
    // We can now actually change the current project
    global::current_project = global::new_project;
}
```

The implementation is hard to understand even with all comments placed in the code. It is impossible to see the flow of the program. Even more, the rule that a method's name should explain what it does can not be followed at all, because none of the methods can really stand by itself. Any naming scheme we choose will fail to explain the purpose of the method, who should call it and when.

Let us look at the equivalent implementation using the IRP method proposed in this dissertation:

```
void request_switch_to(var<string> new_project)
{
    // Decryption status that can be Success,
    // Cancelled or Failed
    var<DecryptionStatus> decryption_status;

    serial_(
        // If the project is private, keep trying
        // to decrypt while the decryption_status
        // is not Success or Cancelled.
        if_(is_private(new_project)) (
            do_(
                decryption_status =
                    request_decryption(new_project)
            ).while_(
                decryption_status == Failed,
            )
        ),

        // Asserting that the user did not cancel
        // the procedure
        decryption_status == Success,

        // If the current project is private,
        // encrypt it.
        if_(is_private(current_project)) (
```

```
            detach_(request_encryption(current_project))
        ),

        // Set the new project to be the current
        current_project = new_project
    )();
}
```

The code is easy to understand, as well as the logic behind it. It can be argued that it is even simpler to understand than the diagram in the Figure 6.1. It also has the benefit of being easy to refactor and change. If we decide that we want to wait for the previously selected project to be encrypted before we change to the new one, we can just remove the `detach_` statement. The same modification would introduce at least one additional method in the original implementation. We will leave the modification of the original code up to the reader.

6.2 Comparison with other techniques

6.2.1 Comparison with coroutines

The concept of coroutines was introduced in the early 1960s [Con63]. It describes a cooperative task management where one routine can yield control to another when it needs to retrieve a result that is not immediately available, and get the control back when available. There is no common definition of the concept, but the general consensus is that coroutines should follow these notions:

- the values of data local to a coroutine persist between successive calls;
- the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

Although a very old concept, and quite covered in the computer-science community and some FPLs, it started to get more popular in practice a few years ago, mostly in the form of generators (see Python generators [SPH]). While they can be used to cover more general use-cases, such is ours, coroutines fail on two important points.

The first is the implementation efficiency. Coroutines, as usually implemented (Python, boost.coroutine), need to save the states of all routines that are started, with special care that needs to be taken regarding the recursive coroutines. This

leads to overuse of memory, inhibition of CPU's branch prediction, and excessive context switching. Furthermore, since coroutine lifetime is not tied to the caller like it is with the normal subroutines, the memory can not be used as a stack. It mandates a more complex memory management system which brings additional performance hits.

The second is a greatly lowered usability. Reasoning about processes where the control is able to jump from one step to another is even harder than thinking in a call-callback paradigm. The flow of the program is completely broken which also makes it very difficult, if not impossible, to do the step-by-step debugging on it. Debugging programs based on the presented IRP solution is the same as debugging callback-based programs as it is how the compiler *sees* the schedulers we defined.

6.2.2 Comparison with resumable functions

As previously shown, general coroutines lack in usability and versatility. One of the possible approaches to fix this problem is to introduce resumable functions and the `co_await` keyword to the language [Nis14] (as proposed for C++20 under the name *coroutines*, even though they do not provide the general for of coroutines).

A resumable function, similarly to a coroutine, is a function that is capable of split-phase execution, meaning that the function may be observed to return from an invocation without producing its final logical result or all of its side-effects [Gus+13]. A resumable function would, when it first suspends, return a placeholder for the result (`std::future<T>`) representing the return value of a function that will eventually be calculated. After suspending, this function will be resumed by the scheduler and will eventually complete its logic, and set the function's result value in the placeholder.

The `co_await` keyword would instruct the compiler that the caller function should be suspended until the `future<T>` returned by the invoked interruptible subroutine gets its value. In the following example, `parent_routine` will execute the first statement, and then become suspended until the result of `complex_subroutine` is calculated. The runtime system needs to know when the calculation is ready in order to be able to continue executing the parent routine from the point it was suspended in.

With this approach, the code becomes straightforward and hides the inversion of control and the asynchrony in a similar manner to that of the proposed imperative reactive programming approach (Listing 6.2). The only point where the asynchrony is exposed is in the need to mark the asynchronous call with `co_await`. This means

```
future<double> complex_subroutine(double parameter)
{
    …
}

future<void> parent_routine( )
{
    // Executes when the function is invoked
    double parameter = 1.0;

    // Defining a break/continue point
    double result = co_await complex_subroutine(parameter);

    // Executes when the complex_subroutine's
    // result is ready
    std::cout << result;

    co_return result;
}
```

Figure 6.2: Resumable function example

that the user needs to know whether a method is asynchronous or not in order to invoke it. It implies the need to change the caller code if the called method's type of execution is changed, which is not a requirement of our approach.

The second, and more significant issue with the resumable functions concept is the need for a more complex programming language runtime. The runtime system needs to do the book-keeping for all futures that are being waited for and to keep the function call contexts in a similar manner to that of coroutines. The approach taken in this dissertation requires no changes to the language nor the runtime system – it uses only the abstraction mechanisms already provided.

The resumable functions also lack the level of abstraction needed to allow the creation of more advanced schedulers like the transaction scheduler (4.3.1).

6.2.3 Comparison with Boost.Asio

Boost.Asio is a great library for asynchronous network and low-level I/O programming, but is applicable also to other domains that require asynchronous execution. Boost.Asio provides the tools to manage asynchronous operations, without requiring programs to use concurrency models based on threads and explicit locking (mutexes

and alike). It is based on the serialized event dispatch system which guarantees that two event handlers will not be invoked concurrently.

The library API is callback-based. Every asynchronous operation requires a handler function to be provided. When the operation is finished, the handler will be called with the result, or with an error code in the case of failure.

Boost.Asio also provides a library-level coroutine implementation that allows writing programs in a similar manner to the resumable functions, but without the need for extending the core language. To implement a coroutine, the developer just needs to create a class that inherits `boost::asio::coroutine`. Since these are stackless coroutines, the state needs to be stored in the class itself. The actual implementation of the coroutine logic goes into the call operator of the class, as it is shown in the Listing 6.3.

The first invocation of the call operator will execute the code after the first `yield` keyword. The second invocation will execute the code after the second `yield` and so on. This, along with passing the reference to the coroutine itself (`*this`) as the callback of the asynchronous operation allows writing localised imperative code that will be unwrapped into calls and callbacks at runtime.

This approach does not require changes to the core language like the resumable functions do, but it requires the user to write boiler-plate code to wrap the coroutine logic in. Compared to the presented solution, its advantage is that it uses mostly standard C++ syntax (apart from the boiler-plate and a few special keywords like `yield`). The disadvantage is that it does not directly support the creation of more advanced schedulers, and that unwrapping the logic into the call-callback pairs happens at runtime instead of it being done during compilation like it is the case with our implementation.

6.2.4 Comparison with actor systems

The actor model formalism was first published by Hewitt, Bishop and Steiger in 1973 [HBS73]. They wanted to create a model for formulating programs for their artificial intelligence research. Shortly thereafter, Ericsson started developing a new functional programming language called Erlang for telecommunication systems implementation where the concept reached its full potential.

The basic idea is to model a system to impersonate a group of people performing a task in collaboration with each other. One person knows how to do specific tasks, and can do it independently of others. He gets a request, performs a task, and sends the result or a new request to another person. A person, in this sense, is an

```
class session: boost::asio::coroutine {
public:
    session(boost::shared_ptr<tcp::socket> socket)
      : m_socket(socket),
        m_buffer(new std::vector<char>(1024))
    {
    }

    void operator() (boost::system::error_code ec =
                         boost::system::error_code(),
                     std::size_t n = 0)
    {
        if (!ec) reenter (this) {
            while (true) {
                yield m_socket->async_read_some(
                        boost::asio::buffer(*m_buffer), *this);
                yield boost::asio::async_write(
                        *m_socket,
                        boost::asio::buffer(*m_buffer, n),
                        *this);
            }
        }
    }

private:
    boost::shared_ptr<tcp::socket> m_socket;
    boost::shared_ptr<std::vector<char>> m_buffer;
};
```

Figure 6.3: Echo server using stackless coroutines in Boost.Asio

independent and isolated entity, and his internal logic and data are not available to the outer world.

The main benefits of actor systems and message passing are loose coupling between components and relieving the user of the need to manually handle threads and distributed computation. Actor systems are generally very scalable in the terms of distributing the workload across a large number of separate physical nodes.

On the other hand, the loose coupling and communication solely through the message queues reduces the usability and composability even more than the call-callback approach. Let us look at an example of requesting a user name in an asynchronous manner. The example (Listing 6.4) is implemented with Akka [2].

[2]Akka – an actor system for the Scala programming language – `http://akka.io`

```
class LoginActor extends Actor {
    def receive = {
        case Login =>
            usernameProvider ! GetUsername()

        case ReturnedUsername(username) =>
            // …
    }
}
```

Figure 6.4: Actor implementation example

At first, the example looks suspiciously like the call-callback example, as shown in the Section 4.1. It has something that looks like our `login` method, a request for the username, and a callback handler. But, if we dive more into it, it shows some additional problems.

When sending a message to the `usernameProvider` actor, we have no indication of where the execution will continue from after the username retrieval task has finished. If we had another case in the above code for a message of type `OnGotUsername(String)`, it would be impossible to tell which of those is the continuation just by reading the code. With the call-callback approach we had this information because it was necessary to provide a specific callback method.

The other problem arises in the case where more than one code path can lead to the invocation of `GetUsername`. In that case, we somehow need to remember (in-actor mutable state, or by creating different actor behaviours) where we asked for the username to be able to process the response correctly.

With our solution, we know that the continuation is what looks like the next statement in the code, and we don't have the limitation that we can have only one handler for a specific asynchronous method.

6.2.5 Testing and validation

So far, we have shown that the proposed approach makes the code easier to write and reason about. We will also briefly demonstrate that it also improves the ability to do automated testing and static code analysis.

6.2.5.1 Automated testing

Unit testing is one of the more popular testing methods for the projects where it is hard or impossible to do formal verification of software. The primary goal of unit testing is to take the smallest piece of code, isolate it from the remainder, and test whether it behaves as desired [HK07].

Testing asynchronous systems is not a solved topic. The need for all previously defined task types (Section 2.1.2) usually comes from the need to integrate more separate agents, be it a user, a separate process or thread, or an external service. While Multi-Agent Systems (MAS) are becoming prevalent in the modern software design, not enough research was focused on how they can be tested [CCZ05].

The Agile methodology for MAS proposes a testing phase based on JUnit which required reimplementing significant parts of the underlying agent platform [Knu02]. An alternative approach is to create mock agents and aspects [Coe+06] which focus on testing the actual system agents in a more granular way. The tests are organized in a few levels – from testing of the specific agents, to testing the integration between them. All this requires a substantial amount of work on the testing infrastructure itself, which increases the probability of errors. It also adds the need for extensive testing of the testing infrastructure itself, which is a clear overhead of the process.

Because of the increased complexity required to create proper MAS tests, in practice, developers tend to write ordinary unit tests with sleep intervals which are meant to allow the system to settle when all the agents in it finish processing their messages. This is widely used, but is not efficient and can be risky because the sleeps can potentially block some messages from arriving, or interfere with the communication in some other manner.

Our approach allows for a cleaner separation of unit and program logic tests. It can ease up integration testing in the casees where the task interactions are fully described by the IRP schedulers – that is, if there is no *behind the scenes* interoperation between tasks. While this assumption is not applicable to all real-world problems, it does cover enough cases to be useful.

Concretely, the clean multi-layered separation comes from the code separation into asynchronous calls and the routines invoking them, which, in turn, can be used as asynchronous calls in other routines. The testing is performed much like in the case of synchronous programs by first testing the smallest non-divisible chunks of code, and then proceeding to test the code that uses the former.

For the program logic tests, instead of the need to reimplement significant parts of the underlying agent platform in the testing framework itself, IRP allows a sim-

pler approach. By creating a common abstraction over both synchronous and asynchronous calls, it allows the same code to be executed against both with no alterations whatsoever.

This means that, in order to test the logic of a routine, one needs only to provide it with the synchronous versions of the used functions. We can demonstrate this on the `login` example. Let us consider just the following:

```
auto login = serial_(
    username = get_username,

    …
);
```

The code is the same regardless of whether the invoked method is asynchronous or not. This means that we can pass it an asynchronous `get_username` method, as well as a blocking one based on `std::getline` or similar. This switch is as simple as changing a compilation flag and does not require any changes to the code itself. This allows the build system to use the same code to generate the final product, the logic/integration tests over synchronous versions of the functions, and tests of the actual asynchronous functions, as presented in the Figure 6.5.

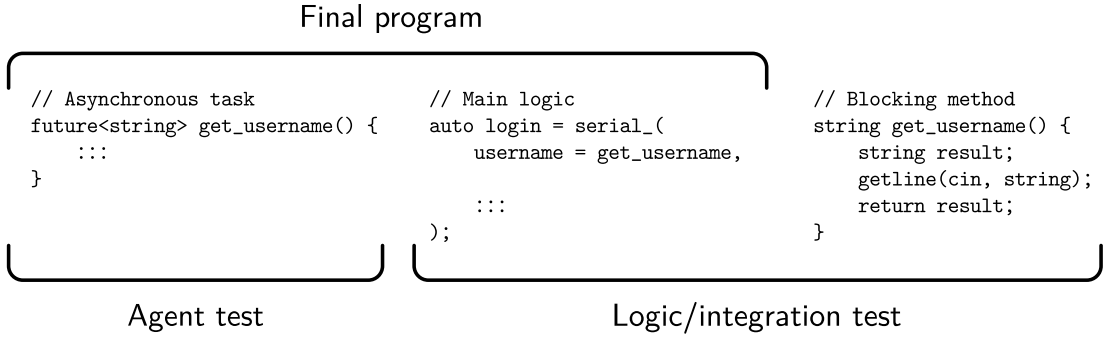


Figure 6.5: Testing components

It is worth noting that the methods based on the IRP schedulers do not need rewriting in order to make them synchronous – they become synchronous automatically when the tasks they invoke are synchronous. In the previous example, the `login` method will be asynchronous in the final program, while being blocking in the logic/integration tests – without any changes to the method nor the scheduler itself.

This way, we are able to catch a class of bugs before the complete integration tests, and in the case it is found, it is more easily located.

6.2.6 Static analysis

We are now going to demonstrate that this approach is also beneficial in aiding the static analysis tools such as `clang-analyze`.

In the same manner as with the unit testing, we are performing the static analysis on the algorithm with tasks replaced by their synchronous equivalents, thus telling the analyser that the variables will not be altered by some other function called by the event loop.

Let us demonstrate this with a simple program that uses a task to initialize a pointer. The task factory function just returns a lambda that will actually set the pointer to point to the object specified by the passed index. The program that uses the function will have a small bug that will allow it to leave the pointer uninitialised:

```
auto assignValue(Value * &vp, int i) {
    return [&] { vp = getObject(i); }
}
```

We will first test the analyser by manually calling the task factory and the function it generates. This will show us what is the best analysis we can get for this example. It should be noted that this basic program would not work with the asynchronous version of the `assignValue` function – it is used only as an edge case for comparing the quality of static analysis output:

```
Value* value = nullptr;
int i = index();

if (i >= 0) {
    assignValue(value, i)();
}

value->write();
```

The bug in the above code is obvious – if the index function returns a negative value, the pointer will be null and we will get a crash at runtime when invoking `value->write()`.

When running the analyser on this code, we get a detailed explanation of the error, as shown in the Figure 6.6 (a). It clearly states that if `i` becomes negative, the pointer will be null in the last line of the `main` function.

```
int main(int argc, char* argv[])
{
    Value* value = nullptr;
```
1 **'value'** intialized to a null pointer value

```
    int i = index();
    if (i >= 0) {
```
2 Assuming **'i'** is < 0

```
        assignValue(value, i)();
```
3 Taking false branch

```
    }

    value->write();
```
4 Called C++ object pointer is null

```
}
```

(a) Ordinary `if`

```
int main(int argc, char* argv[])
{
    Value* value = nullptr;
```
1 **'value'** intialized to a null pointer value

```
    int i = index();
    if_ (i >= 0) (
        assignValue(value, i)();
    )();

    value->write();
```
2 Called C++ object pointer is null

```
}
```

(b) IRP `if_`

Figure 6.6: Clang analyser output

The next step is to do the same for the IRP version where we replace the `if` construct with the IRP equivalent (Listing 6.7). As previously stated, we are still testing with the synchronous version of the `assignValue` function, but the relevant part of the code would work with the asynchronous version without any changes.

```
Value * value = nullptr;
int i = index();

if_ (i >= 0) (
    assignValue(value, i)
);

value->write();
```

Figure 6.7: Function invocation with the `it_` scheduler

The output generated by the analyser, as shown in the Figure 6.6 (b). is not as detailed as the first one because the analyser does not show the insides of the `if_`

call, but it still tells the user that there is a problem of calling a method on a null pointer. The analyser could be easily taught what the `if_` means thus allowing it to create even better diagnostics, but for most purposes, this is sufficient.

The last step is to actually convert the starting example to its call-callback equivalent (Listing 6.8). This approach requires moving the `value` variable to the global scope and splitting the main function into two separate parts.

```
Value * value = nullptr;

auto assignValueIfIndexPositive(Value * & vp, int i,
                                void (*callback)(void))
{
    if (i >= 0) {
        vp = getValue(i);
    }

    callback();
}

void onFinished()
{
    value->write();
}

int main(int argc, char *argv[])
{
    int i = index();

    assignValueIfIndexPositive(value, i, onFinished);
}
```

Figure 6.8: Call-callback version of the code

When running the analyser on this program, it just returns a 'No bugs found.' message. This shows that, while the static analysis of a IRP scheduler-based program is not as detailed as the one performed on an ordinary synchronous program, it is still much better than the one performed on a call-callback version. The fact that the analysis does not work for the approach that has been ubiquitous for a long time and does on a new one speaks for itself.

6.2.7 Related work

Lauer and Needham [LN79] have shown that the message passing and threads are dual to each other, and that anything that is modeled using one of the approaches, can be implemented with the other. Both systems have their advantages and disadvantages, and while plain threads are usually considered bad as explained by Ousterhout, and more recently by Keiser [Ous96; Kei14], and under-performing as shown by von Behren [BCB03], they are usually prevalent in most modern web servers [Men03] and other software systems mainly because event-driven programs are difficult to write and maintain [BCB03].

Lately, it has become apparent that an abstraction of different asynchronous models is needed that will make the software easier to manage while not losing performance.

The approach taken by Haller and Odersky on a managed platform [HO06] and Charousset et al. for the native code [Cha+13] is to implement concurrent systems using the actor model. This allows for a very horizontally scalable system. But the actors are not easy to compose, and additional patterns need to be devised to truly avoid IoC. Examples include the ask pattern [Hun14] or more intricate constructs like async-finish [IS12].

Another approach is to try to make threads more usable by creating task-based systems which can schedule threads by creating different types of executors, providing thread pools and work stealing semantics [Koh13].

Most of the recent research tends to focus on improving and developing the lower-level parts of the system, with only a small number of new approaches being taken for defining the higher-level abstractions. Approaches like the one taken in the Responders library [CM06], provide a way of defining a control flow through manipulation of shared state. While this allows the code to become more organized and readable, the shared state requires synchronization mechanisms which induce performance penalties. And it still does not provide the level of expressiveness that our approach does.

In 1999, Claessen proposed using the continuation monad for modeling asynchronous systems [Cla99]. The continuation monad is used by Li and Zdancewic to create a system that unifies events and threads for Haskell [LZ07]. The continuation monad is further abstracted to reactive streams [WH00; MRO10] which allow using the common FPL patterns of list transformations to the event streams and asynchronously calculated data.

This idea is valuable even for the non-functional programming languages. It

just needs to be further expanded and adapted to imperative and object-oriented paradigms like it is presented in this dissertation.

Since the solution proposed in this dissertation is based on the continuation monad, it can abstract out any underlying model for asynchronous programming and therefore automatically benefit from further developments in both actor and message-passing systems, and task and thread-based models. It also allows easy integration with the reactive streams thus making it a library that allows writing asynchronous programs in a natural way to both functional and imperative programmers. Another approach that uses continuations is JavaTasks [FMM07] which has some similarities to our approach. It defines a new programming language that compiles to Java and implements resumable functions using switch constructs which define which part of the routine should be executed. The main similarity to our solution is that it is scheduler-agnostic, and that it can work with different event systems.

6.3 Conclusion

We have defined an abstraction that is able to handle different types of tasks, and used it for creating the control structures to mimic those from the host language, along with a few that are not available in the host language (Sections 4.3.1 and 4.3.2) With this, we have created an embedded DSL inside C++ that allows the user to write the program logic that deals with asynchronous tasks in a more natural manner.

We have demonstrated that this approach makes it easier to write and reason about the programs that use asynchronous APIs along with the synchronous ones. The program becomes a simple set of tasks that can run asynchronously and still be deterministic.

The defined abstractions allow for swapping asynchronous calls with synchronous ones without any code changes. This allows for easier automatic testing and static code analysis.

We have also shown that the task abstraction introduces no performance overheads compared to the usual code.

The future work will focus on a few additional integration points between the existing system and various asynchronous models, and host programming languages, and potential inclusion of collective barrier inclusion and other synchronization constructs [IS14]. While C++ provides unmatched performance, a system like this could

be beneficial to Scala and its Akka library [HO06], as well as the Cloud Haskell library [EBP11].

It is also worth researching the different types of error handling in asynchronous environments and multi-agent systems, and how they can be adapted to the concept of C++'s exceptions and its current techniques with cross-thread exception propagation.

Another area of interest is the usefulness of the created schedulers on the other types of monads beside continuations. The system should be applicable to standard collections like lists and vectors, as well as the IO, Parser and similar monads. It needs to be researched whether it would bring significant benefits to those areas as well.

A Relation to KDE Plasma

The initial idea for creating an embedded DSL (domain-specific language) for asynchronous task scheduling was born during the development of the *activities* system of KDE Plasma Workspace. The activities system provides the user with a way to manage projects, and how the Plasma Workspace behaves in each of the projects.

Activities provide the user a way to set up encrypted password-protected projects where only one project can be activated at some point in time. The project can get activated only if the user provided the correct password and the project decryption was successful. The actual encryption and decryption is handled by a trusted 3rd party system which means that the activities system can not check the password that the user entered, it can just wait for the 3rd party system to report whether the data is accessible or not.

Due to the interconnections between project switching and the encryption system, significant parts of the code were intermingled between the project management class and the encryption management class. This was not a clean design since the shared responsibility for the same task broke the encapsulation, and a lot of the jobs of the project management class were performed by the encryption management class.

Apart from that, the code was split into overly many callbacks, and had a few global variables that needed special handling because different asynchronous calls were both reading and writing them. User input, project opening and closing, communicating with the encryption system, communicating with the Plasma Desktop, etc. all had to be modelled as asynchronous tasks to avoid blocking the user interface. The further complication came from the fact that any of the operations can fail and most of the failures required further synchronization with the rest of the system. Because of all of this, the number of contributors willing to work on this system decreased significantly, and those who remained failed to fully understand the inner-workings of the system which resulted in an increased bug count.

After a particularly dubious commit made against the main service of the activities system, it was decided that it needs to be rewritten to handle asynchronocity much better. The solution was to create asynchronous tasks and schedulers much like the ones shown in this dissertation, although they were not as elegant at first.

After the switch to the implementation based on the continuation monad and schedulers[1] the need for a dedicated encryption management class went away. The logic for project switching, which was formerly interspersed into two classes and a

[1] commit 5c85863dab2c2 from March 2012

few dozen member functions became contained in a single one, without any global variables that needed any special handling.

The concepts behind the schedulers were further developed for this dissertation, along with the presented approaches that solve similar problems in the functional style. Different implementations of the proposed approach ware successfully applied in various systems – both in open source systems used by millions of people (in different parts of the KDE software which is deployed in schools[2], film studios[3], control room of the CMS experiment of the Large Hadron Collider in CERN[4], etc.) and in proprietary systems developed by third parties.

[2]KDE to Serve 52 Million Brazilian Students – https://dot.kde.org/2008/04/25/kde-serve-52-million-brazilian-students

[3]KDE Plasma Used in The Hobbit – http://news.softpedia.com/news/KDE-Plasma-Used-in-The-Hobbit-The-Desolation-of-Smaug-Production-Video-461707.shtml

[4]Bugs in the Fabric of Reality – http://cukic.co/2016/03/15/bugs-in-the-fabric-of-reality/

B Community survey

In order to test the professional community opinion on the IRP schedulers, we had set up a small survey with 29 participants. The survey was open for two days for all willing participants, and the invitation to participate was posted to a few social channels mainly targeting the open source developers community. None of them had any previous contact with this particular solution.

The survey compared the IRP appreoach to the three most common approaches for programming asynchronous systems: call-callback approach; actors and message passing; and coroutines. The questions presented in the survey, along with the collected data can be found online[1].

The introductory questions covered the general programming experience of participants, and their previous experience with asynchronous programming.

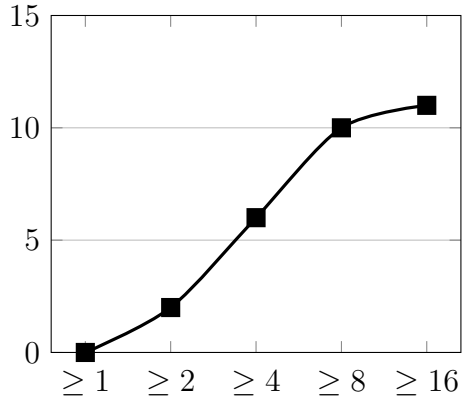


Figure B.1: Programming experience of the survey participants (x-axis – years of programming experience, y-axis – number of participants)

These have shown that the survey was filled in mostly by experienced developers (majority had more than 8 years of programming experience) as shown in the Figure B.1. The most popular programming language amongst the surveyed was C++ (which is the targeted audience), followed by C, Java and similar languages. Script languages like JavaScript and Python were moderately represented, while the participants had mostly no experience with any functional programming language (Figure B.2).

The survey showed that the different techniques of asynchronous programming are not well known, even amongst experienced developers. The most used ones are the call-callback and signal-slots approaches. This is understandable considering that the participants are mostly focussed on C and C++ and other object-oriented

[1]Survey questions and collected data: `http://poincare.math.rs/~ivan/papers/causeway/survey/`

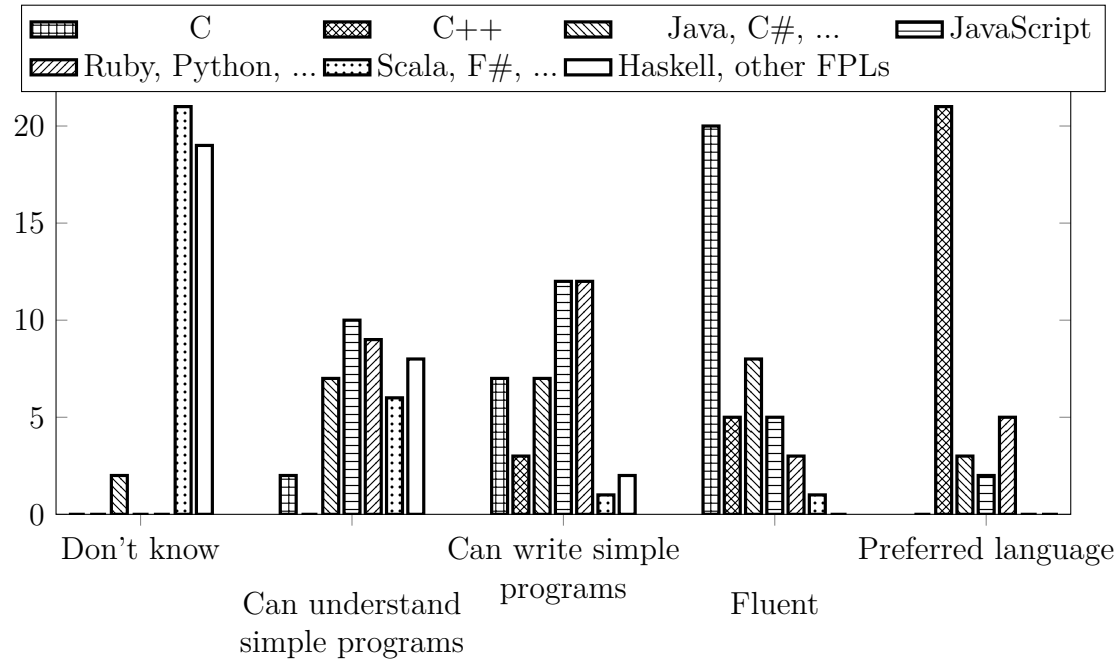


Figure B.2: Expertise in specific programming languages (x-axis – level of expertise, y-axis – number of participants)

languages, where these two are prevalent. The actor systems took the third place (Figure B.3).

B.1 Basic code example

The participants were first presented with a simple example written with all four approaches. The example implemented a loop that shows a message to the user every two seconds, where the waiting it asynchronous, that is, the main process is not blocked between the messages.

The participants were asked to rate the solutions based on readability and how easy it is to comprehend the code. The highest score was given to the solution based on IRP schedulers (4.79) with 25 out of 29 people giving it the highest score possible (Figure B.4 (a)). The second one were coroutines (3.07), followed by callbacks (2.79) and actors (1.69).

The second question was about how many additional lines of code do they think are needed to modify the program to show an additional message to the user and sleep for three more seconds afterwards. The participants recognized the IRP-based solution as the easiest to modify. The close second were coroutines for which the most chosen option was also $1 - 2$ lines, but chosen only by 15 voters, compared to

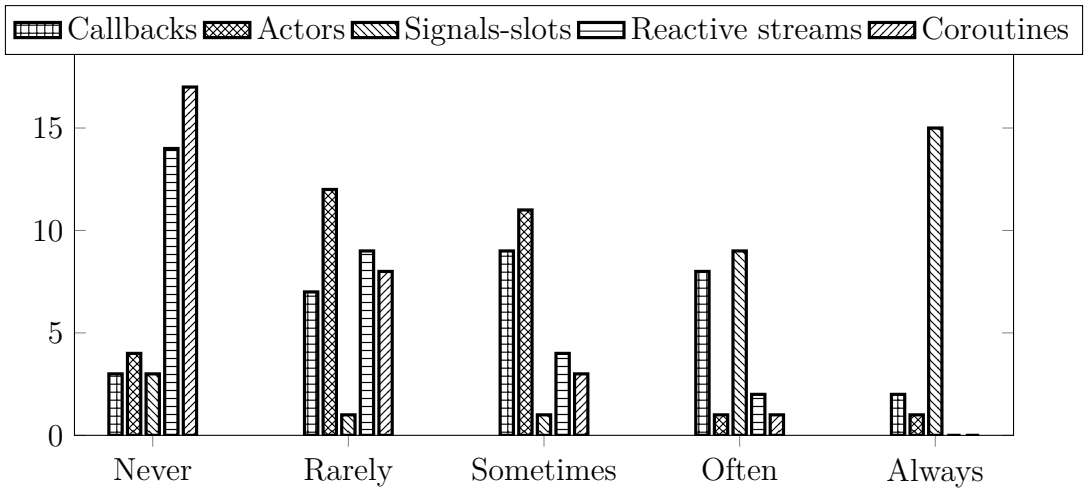


Figure B.3: Experience with asynchronous programming techniques (x-axis – how often a particular technique is used, y-axis – number of participants)

26 for IRP. Additionally, the low dispersion of answers for the IRP-based solution have shown that it leaves almost no doubt as to how to change the code. Coroutines, as the second best solution, had the "I don't know" answer (represented as '?' in the graph) chosen by 6 people (Figure B.4 (b)).

B.2 Asynchronous method invocation

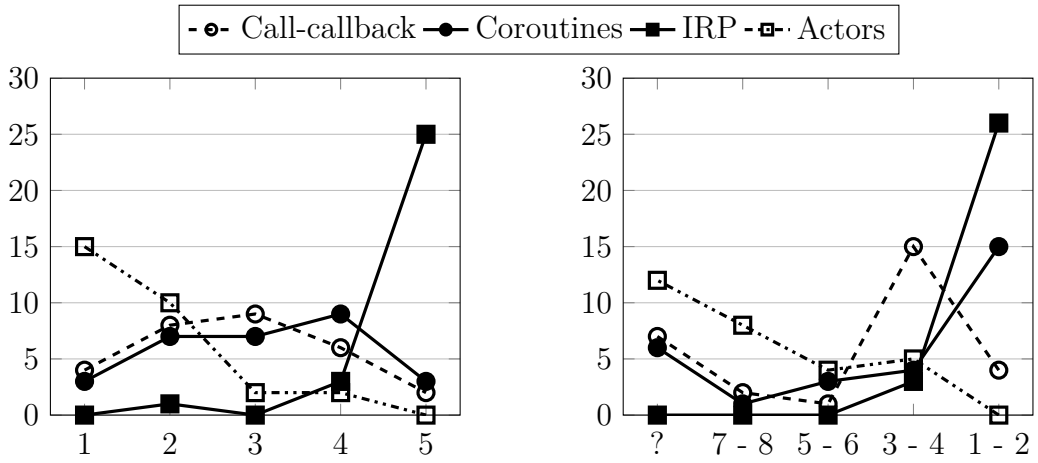
The next part was about ranking the readability of code snippets that invoke an asynchronous method, and comparing the intrusiveness of changing it to perform invocation of a synchronous method.

For readability (Figure B.5 (a)), the IRP-based solution got the highest score of 3.76, followed closely by callbacks (3.48) and coroutines (3.34), while actors got the lowest score (1.76).

With regard to intrusiveness of changing the code to be synchronous, the IRP-based solution was ranked as the least intrusive (score of 4.62) with other options ranking the same as in the previous question (Figure B.5 (b)).

B.3 Producer-consumer example

The last example was demonstrating the implementation of producer-consumer pattern in the form of a simple echo server (Figure B.6). The procedure is taking the clients from the producer one by one, asks the client for the message, and replies with the same message. The participants were asked to rank the solutions for read-



(a) How easy it is to understand (x-axis – 1 meaning it is difficult to understand, and 5 meaning that it is easy to understand, y-axis – number of participants)

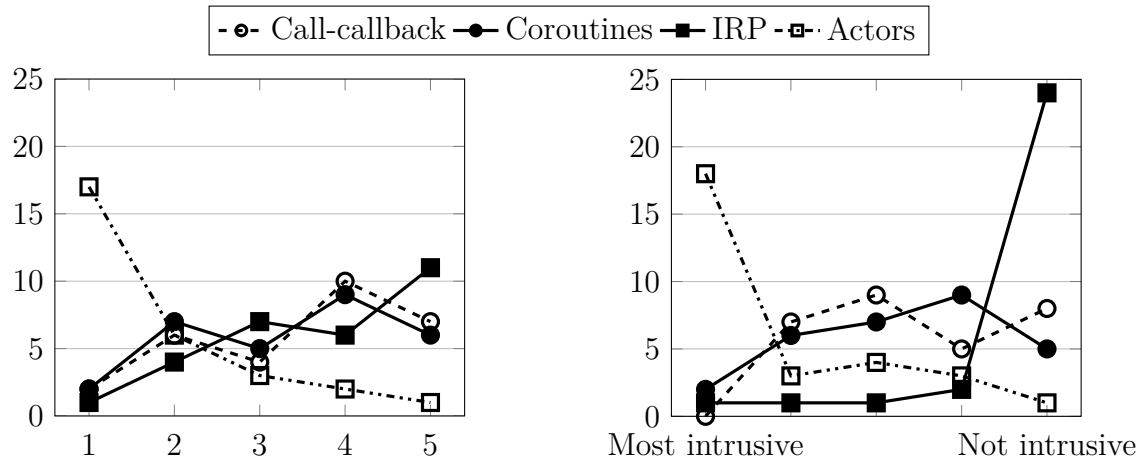
(b) How easy it is to modify (x-axis – how many lines of code need to be changed to achieve the specified goal, y-axis – number of participants)

Figure B.4: Basic example

ability and intrusiveness of changing to synchronous execution, like in the previous example.

The IRP-based solution got the highest scores on both (4.72 for readability, 4.75 for non-intrusiveness), followed by coroutines (3.59, 3.17), actors (2.18, 1.97) and callbacks (1.45, 1.83).

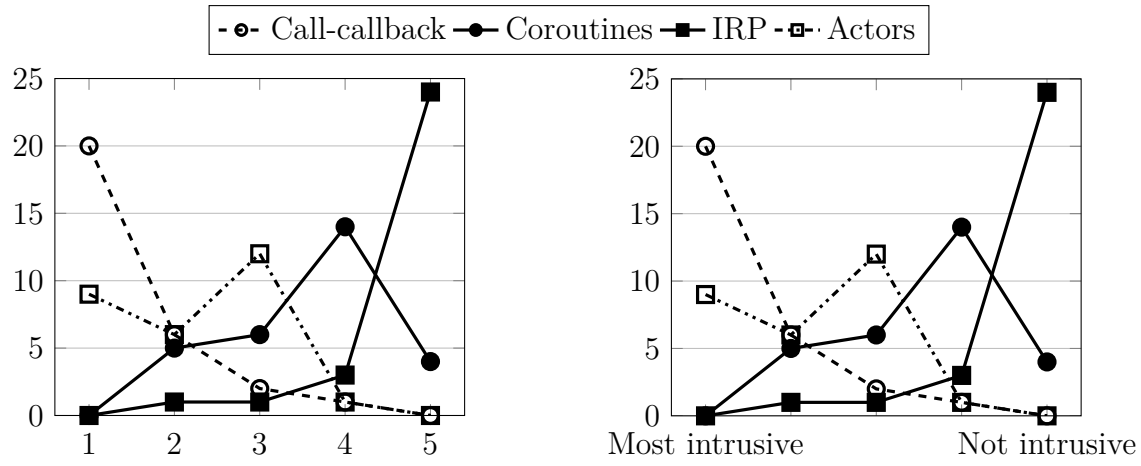
The results clearly show that the participants consider the code based on the IRP schedulers proposed in this dissertation to be easiest to understand and modify.



(a) Code readability and comprehensiveness (x-axis – 1 meaning low readability, and 5 meaning high readability, y-axis – number of participants)

(b) Intrusiveness of switching to synchronous method invocation (x-axis – score ranging from the most intrusive, to not intrusive at all, y-axis – number of participants)

Figure B.5: Asynchronous calculation



(a) Code readability and comprehensiveness (x-axis – 1 meaning low readability, and 5 meaning high readability, y-axis – number of participants)

(b) Intrusiveness of switching to synchronous method invocation (x-axis – score ranging from the most intrusive, to not intrusive at all, y-axis – number of participants)

Figure B.6: Producer-consumer pattern implementation

C Extending the QFuture

In Chapter 3, we mentioned that the `QFuture` type was a combination of the exception and continuation monads like other covered future types, that the main difference was the syntax for specifying the continuation function.

In reality, the `QFuture` class template is somewhere between a future and a reactive stream. Namely, the `QFuture` can return multiple values over time just like reactive streams, the only limitation is that the values are stored inside of the future object instance forever which means that a future can optimally return only a limited number of values contrary to streams which were infinite.

While the `QFuture` is the main user-facing type, all the work happens in the `QFutureInterface` type. It implements the logic of the asynchronous operation and pushes the resulting values into the `QFuture` instance. We can demonstrate this with a simple function which creates a future object that has already finished, and contains a value:

```
template <typename T>
QFuture<std::decay_t<T>> makeReadyFuture(T&& value)
{
    QFutureInterface<T> interface;
    auto future = interface.future();

    interface.reportStarted();
    interface.reportResult(
        std::forward<T>(value));
    interface.reportFinished();

    return future;
}
```

The interface defines when the calculation has started, the resulting value, and when it has finished. The future object is only a handler which can be used to retrieve the resulting value or values.

The `makeReadyFuture` function is one of the possible constructor functions for the `QFuture` monad. It takes a value of type `T` and produces an instance of `QFuture<T>` that contains said value. A possible alternative constructor would be a function that creates a future whose value is not calculated yet, but will appear later.

We can also create a function that creates a canceled future – a future that has finished and contains an exception instead of a normal value:

```
template <typename T>
QFuture<T> makeCanceledFuture(const QException &exception)
{
    QFutureInterface<T> interface;
    auto future = interface.future();

    interface.reportStarted();
    interface.reportException(exception);
    interface.reportFinished();

    return future;
}
```

Both `makeReadyFuture` and `makeCanceledFuture` create an already completed future. Implementing futures which are not finished is more involved. We need to subclass the `QFutureInterface` class template and implement the logic for the value generation manually. Let us demonstrate this with a delayed future – a future that emits a given value after a predefined period of time.

```
template <typename T>
class DelayedFutureInterface : public QObject
                             , public QFutureInterface<T> {

public:
   DelayedFutureInterface(T value, int milliseconds)
        : m_value(value)
        , m_milliseconds(milliseconds)
    {
    }

    QFuture<T> start()
    {
        auto future = this->future();

        this->reportStarted();
```

```
        QTimer::singleShot(m_milliseconds, [this] {
            this->reportResult(m_value);
            this->reportFinished();
            deleteLater();
        });

        return future;
    }

private:
    T m_value;
    int m_milliseconds;
};
```

When this future interface is started, it will schedule a timer that will set the future value after a predefined number of milliseconds, and report that the future has finished. We can wrap this in a convenient constructor function that creates the delayed future:

```
template <typename T>
QFuture<std::decay_t<T>> makeDelayedFuture(T&& value, int milliseconds)
{
    return (new DelayedFutureInterface<std::decay_t<T>>(
            std::forward<T>(result), milliseconds)
        )->start();
}
```

C.1 Transform

Now that we have seen several constructor functions for the `QFuture`, we can move on to implement other functions required for the continuation monad. We can start with `transform`. We have to subclass the `QFutureInterface` class template again. This class will have to store the `QFuture` instance it is meant to transform, the transformation function and the `QFutureWatcher` object to monitor the events on the source future object[1].

[1]The used type traits are omitted for readability. The full implementation is available at `https://cgit.kde.org/asynqt.git/`

```
template <typename _In, typename _Transformation>
class TransformFutureInterface
    : public QObject
    , public QFutureInterface<…> {

    …

private:
    QFuture<_In> m_future;
    _Transformation m_trafo;
    std::unique_ptr<QFutureWatcher<_In>> m_futureWatcher;
};
```

Just like with the `DelayedFutureInterface`, the function where the main logic is implemented is the `start` member function. It needs to react to all events that the source future can emit – it needs to listen for the generated values, transform them, and pass them on; and it needs to pass on the events that mark the future completion. The `start` member function will look like this:

```
QFuture<result_type> start()
{
    m_futureWatcher.reset(new QFutureWatcher<_In>());

    onFinished(m_futureWatcher, [this] { this->reportFinished(); });
    onCanceled(m_futureWatcher, [this] { this->reportCanceled(); });

    onResultReadyAt(m_futureWatcher,
        [this] (int index) {
            this->reportResult(
                std::invoke(m_trafo, m_future.resultAt(index)));
        });

    this->reportStarted();
    m_futureWatcher->setFuture(m_future);
    return this->future();
}
```

C.2 Flattening out the nested futures

Now that we have the constructor for the `QFuture` and the `transform` modifier, we just need to implement the `join` function that flattens out nested futures. Since each `QFuture` instance can generate multiple values, this means that if we are given a `QFuture<QFuture<T>>`, we have a future that can generate several `QFuture<T>` objects and that we need to listen to them all.

Ordinary reactive streams are infinite, which means that the only possible implementation of a `join` transformation is to listen for all source streams at once and emit the values as soon as they appear. Since the `QFuture` can only emit a limited number of values, we get a few more options. We can use the Round Robin algorithm [Sil09] and pick one value at a time from each of the futures we are listening to. We can also perform ordered joining where we emit all the values generated by the first future before moving on to the next one.

For this, we will need to create a class that will hold a single instance of a nested future, along with a corresponding watcher object and a queue of ordinary future objects we got from the nested one. Since we want to implement ordered joining, we only need to listen to one future at a time which means that we only need a single additional watcher object.

```
template <typename _Result>
class JoinFutureInterface
    : public QObject
    , public QFutureInterface<_Result> {

    …

private:
    QFuture<QFuture<_Result>> m_outerFuture;
    std::unique_ptr<QFutureWatcher<QFuture<_Result>>>
            m_outerFutureWatcher;

    QQueue<QFuture<_Result>> m_innerFutures;
    QFuture<_Result> m_currentInnerFuture;
    std::unique_ptr<QFutureWatcher<_Result>>
            m_innerFutureWatcher;
};
```

The `start` member function will be more complicated compared to the one in `TransformFutureInterface` as it has two levels of futures to deal with. The future completion events need to be passed on only if the queue `m_innerFutures` queue is empty. When a new value appears, it needs to add it to the queue.

```
QFuture<_Result> start()
{
    m_outerFutureWatcher.reset(new QFutureWatcher<QFuture<_Result>>());

    onFinished(m_outerFutureWatcher, [this] {
        if (m_innerFutures.isEmpty()) {
            this->reportFinished();
        }
    });

    onCanceled(m_outerFutureWatcher, [this] {
        if (m_innerFutures.isEmpty()) {
            this->reportCanceled();
        }
    });

    onResultReadyAt(m_outerFutureWatcher, [this] (int index) {
        m_innerFutures.enqueue(m_outerFuture.resultAt(index));
        processNextInnerFuture();
    });

    m_outerFutureWatcher->setFuture(m_outerFuture);
    this->reportStarted();
    return this->future();
}
```

The main job of the `start` member function is to handle the queue of futures we need to listen to. The logic that does flattening is in the `processNextInnerFuture` function. This function needs to handle the events coming from the first future in the queue. If the current future has finished, the next one from the queue is scheduled for processing. If the future has been canceled, it propagates the cancellation to the `JoinFutureInterface`.

```
void processNextInnerFuture()
{
    // Already processing something
    if (m_innerFutureWatcher) return;

    m_innerFutureWatcher.reset(new QFutureWatcher<_Result>());
    m_currentInnerFuture = m_innerFutures.head();

    onFinished(m_innerFutureWatcher, [this] {
        m_innerFutureWatcher.reset();
        m_innerFutures.dequeue();

        if (m_innerFutures.size() == 0) {
            if (m_outerFuture.isCanceled()) {
                this->reportCanceled();
            }

            if (m_outerFuture.isFinished()) {
                this->reportFinished();
            }
        } else {
            processNextInnerFuture();
        }
    });

    onCanceled(m_innerFutureWatcher, [this] {
        this->reportCanceled();
    });

    onResultReadyAt(m_innerFutureWatcher, [this] (int index) {
        this->reportResult(m_currentInnerFuture.resultAt(index));
    });

    m_innerFutureWatcher->setFuture(m_currentInnerFuture);
    this->reportStarted();
}
```

These are all the functions required for the `QFuture` to be a monad. Other useful functions that we defined for the reactive streams like filtering can also be easily implemented in a similar manner.

C.3 Void future

A special type of a future is a future that does not return any value but only communicates that an asynchronous operation has finished – a *void* future. All the future implementations we mentioned in Chapter 3 including `QFuture` allow the type parameter to be `void`. While reactive streams can not be defined as streams of `void` values since the `void` type does not have a value, void futures are often used in practice to denote that a process has been finished.

All the previously implemented transformations of the `QFuture` require the future to have regular values. In C++, `void` is not a regular type [Cal16] which means that the previously defined constructs will not work on `QFuture<void>`. In order to support void futures, we need to create specializations for all of the previous functions and classes we have created.

For the constructor function, the specialization is fairly trivial – we just need to create a `QFutureInterface` and immediately report that it is finished without previously setting a value:

```
QFuture<void> makeReadyFuture()
{
    QFutureInterface<void> interface;
    auto future = interface.future();

    interface.reportStarted();
    interface.reportFinished();

    return future;
}
```

Some specializations will be straight-forward like the `makeReadyFuture`, but most will not. In some cases, the specializations will require some thinking. Consider the `transform` stream modifier. When we look at the types involved, we have the `QFuture<T>` instance that we need to transform, and we have the resulting `QFuture<R>`, where `R` is the return type of the transformation function.

With regard to void futures, we have four separate cases that need to be covered:

- Both `T` and `R` are regular types
- Only `T` is regular, `R` is `void`
- `T` is `void` and `R` is regular
- Both `T` and `R` are `void`

The first case is already covered. For other cases, we need to cover what *transform* means. For example, when `R` is `void`, we can not emit the transformed values. Does that mean that we do not need to call the transformation function at all? Similarly, if `T` is `void`, we can never get a value to transform. Does *that* mean that the transformation function needs not to be called?

These questions do not have a single correct answer, but we can choose the ones that fit the most common use-cases.

If the return type of the transformation is `void`, the most common use-case is probably that it is a *sink* function (see Section 3.2.1.2) and that the developer is only interested in when all the asynchronous operations have completed. In this case, the obvious choice is to call the transformation on every value coming from the source `QFuture` instance.

If the source future is a void future, then we will never receive an event that a new value appeared, but the developer will still expect the transformation function to be executed. The only choice we have is to execute the transformation function when the source future emits that it has finished.

This means that we need to create specializations for both the function that reports new values and for the function that handles the finished event of the source future.

```
void setFutureResultOnFinished(std::true_type, /* T is void */
                               std::true_type  /* R is void */)
{
    if (!m_future.isCanceled()) {
        m_transformation();
    }
}

void setFutureResultOnFinished(std::true_type, /* T is void */
                               std::false_type /* R is not void */)
{
```

no value, no result to create, but it should still call the transformation function

```
    if (!m_future.isCanceled()) {
        this->reportResult(
            m_transformation());
    }
}

template <typename Other>
void setFutureResultOnFinished(
    std::false_type, Other)
{
}

void setFutureResultAt(int index,
                       std::false_type, /* T is not void */
                       std::true_type   /* R is not void */)
{
    m_transformation(
        m_future.resultAt(index));
}

void setFutureResultAt(int index,
                       std::false_type, /* T is not void */
                       std::false_type  /* R is not void */)
{
    this->reportResult(
            applyTransformation(m_future.resultAt(index)));
}

template <typename Other>
void setFutureResultAt(int, std::true_type, Other) {}

QFuture<result_type> start()
{
    m_futureWatcher.reset(new QFutureWatcher<_In>());
```

pretending that the completion of the source future instance actually emits a void value

If the source future is not a void future, it needs no special handling of the completion event

nothing to do with the value, but we still want to call the transformation function

```
    onFinished(m_futureWatcher, [this] {
        setFutureResultOnFinished(in_type_is_void(),
                                  result_type_is_void());
        this->reportFinished();
    });

    onCanceled(m_futureWatcher, [this] { this->reportCanceled(); });

    onResultReadyAt(m_futureWatcher, [this] (int index) {
        setFutureResultAt(index, in_type_is_void(),
                                 result_type_is_void());
    });

    m_futureWatcher->setFuture(m_future);
    this->reportStarted();
    return this->future();
}
```

As this example shows, in order to support void futures, a significant effort is required, but the work is mostly straight-forward. Once we decide what a specific transformation means in the context of void futures, the code can easily be written.

C.4 The AsynQt library

This was a short demonstration of how concepts presented in this dissertation can be applied to an independent software library and a non-standard design of a future value concept. A more complete implementation of various extensions for the `QFuture` type can be found in the `AsynQt` library[2] developed as a part of this dissertation.

[2]KDE AsynQt library –`https://cgit.kde.org/asynqt.git/`

Bibliography

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70431-5.

[Ale12] Andrei Alexandrescu. "Systematic Error Handling". In: *C++ and Beyond 2012*. 2012.

[Ale13] Alvin Alexander. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media, Inc., 2013. ISBN: 9781449339616.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. 2003.

[Awo06] Steve Awodey. *Category Theory*. Oxford University Press, 2006. ISBN: 0198568614.

[Bai+13] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[BCB03] J Robert von Behren, Jeremy Condit, and Eric A Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers)." In: *HotOS*. 2003, pp. 19–24.

[Bou06] Frédéric Boussinot. "FairThreads: Mixing Cooperative and Preemptive Threads in C: Research Articles". In: *Concurrency and Computation: Practice and Experience* 18.5 (Apr. 2006), pp. 445–469. ISSN: 1532-0626. DOI: 10.1002/cpe.v18:5.

[But] David Butenhof. *Recursive mutexes*. 2005 (accessed on 24. 8. 2015.) URL: http://www.zaval.org/resources/library/butenhof1.html.

[Cal16] Matt Calabrese. *P0146R1: Regular Void*. Tech. rep. Standard C++ Foundation, 2016.

[Car] John Carmack. *In-depth: Functional programming in C++*. 2012 (accessed on 24. 8. 2015). URL: http://gamasutra.com/view/news/169296/Indepth%5C_Functional%5C_programming%5C_in%5C_C.php.

[Car+94] Nicholas J Carriero et al. "The Linda alternative to message-passing systems". In: *Parallel Computing* 20.4 (1994). Message Passing Interfaces, pp. 633–655. ISSN: 0167-8191. DOI: 10.1016/0167-8191(94)90032-9.

[CCZ05] Luca Cernuzzi, Massimo Cossentino, and Franco Zambonelli. "Process Models for Agent-based Development". In: *Eng. Appl. Artif. Intell.* 18.2 (Mar. 2005), pp. 205–222. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2004.11.015.

[Cha+13] Dominik Charousset et al. "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments". In: *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013.

[Cla99] Koen Claessen. "Functional Pearls: A Poor Man's Concurrency Monad". In: *Journal of Functional Programming* 9.3 (1999). DOI: 10.1.1.39.8039.

[CM06] Brian Chin and Todd Millstein. "Responders: Language support for interactive applications". In: *ECOOP 2006–Object-Oriented Programming*. Springer, 2006, pp. 255–278.

[Coe+06] Roberta Coelho et al. "Unit testing in multi-agent systems using mock agents and aspects". In: *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems - SELMAS '06*. ACM Press, 2006. DOI: 10.1145/1138063.1138079.

[Con63] Melvin E. Conway. "Design of a separable transition-diagram compiler". In: *Communications of the ACM* 6.7 (July 1963), pp. 396–408. ISSN: 0001-0782. DOI: 10.1145/366663.366704.

[Čuk13] Ivan Čukić. "Natural task scheduling using futures and continuations". In: *Qt Developer Days 2013*. 2013. URL: https://devdays.kdab.com/?page_id=225#64.

[Čuk14] Ivan Čukić. "Monads in Chains". In: *Meeting C++*. 2014. URL: http://meetingcpp.com/index.php/tv14/items/22.html.

[Čuk15a] Ivan Čukić. "Functional Reactive Programming for C++". In: *Central-European Functional Programming Summer School*. Budapest, Hungary, 2015.

[Čuk15b] Ivan Čukić. "Keynote". In: *C++ Siberia*. Tomsk, Russia, 2015.

[Čuk15c] Ivan Čukić. "Task scheduling with the continuation monad". In: *C++ Russia*. 2015. URL: http://meetingcpp.ru/?page_id=597.

[Čuk16a] Ivan Čukić. "A continuation-based task programming model for C++: design of the Causeway library". In: *Software: Practice and Experience* 46.12 (Feb. 2016), pp. 1617–1656. DOI: 10.1002/spe.2395.

[Čuk16b] Ivan Čukić. "For a brighter QFuture". In: *QtCon*. 2016.

[Čuk18] Ivan Čukić. *Functional Programming in C++*. Manning Publications, 2018. ISBN: 9781617293818.

[Cza02] Krzysztof Czarnecki. "Generative Programming: Methods, Techniques, and Applications Tutorial Abstract". In: *Software Reuse: Methods, Techniques, and Tools*. Ed. by Cristina Gacek. Vol. 2319. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 477–503. ISBN: 978-3-540-43483-2. DOI: 10.1007/3-540-46020-9_38.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Vol. 8. A.P.I.C. Studies in Data Processing. Academic Press, 1972.

[Dea16] Ben Deane. "std::accumulate: Exploring an Algorithmic Empire". In: *CppCon*. 2016. URL: https://www.youtube.com/watch?v=B6twozNPUoA.

[EBP11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. "Towards Haskell in the Cloud". In: *SIGPLAN Not.* 46.12 (Sept. 2011), pp. 118–129. ISSN: 0362-1340. DOI: 10.1145/2096148.2034690.

[Esc17] Vicente J. Botet Escribá. *P0323R3: Utility class to represent expected object*. ISO/IEC JTC1 SC22 WG21 Programming Language C++, 2017.

[Fel88] Mattias Felleisen. "The Theory and Practice of First-class Prompts". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 180–190. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73576.

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. "Tasks: Language Support for Event-driven Programming". In: *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '07. Nice, France: ACM, 2007, pp. 134–143. ISBN: 978-1-59593-620-2. DOI: 10.1145/1244381.1244403.

[Fre+04] Elisabeth Freeman et al. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. ISBN: 0596007124.

[Fug] Hans Fugal. *Futures for C++11 at Facebook*. 2015 (accessed on 12. 10. 2016.) URL: https://code.facebook.com/posts/1661982097368498/futures-for-c-11-at-facebook/.

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Gus+13] Niklas Gustafsson et al. *N3564: Resumable Functions*. Tech. rep. Standard C++ Foundation, 2013.

[Hal13] Pablo Halpern. *N3557: Considering a Fork-Join Parallelism Library*. Tech. rep. Standard C++ Foundation, 2013.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[HK07] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley, 2007. ISBN: 9780470165164. URL: http://books.google.ca/books?id=PhnoE90CmdIC.

[HO06] Philipp Haller and Martin Odersky. "Event-based programming without inversion of control". In: *Modular Programming Languages*. Springer, 2006, pp. 4–22.

[Hud+07] Paul Hudak et al. "A history of Haskell: being lazy with class." In: *HOPL*. Ed. by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856. URL: http://dblp.uni-trier.de/db/conf/hopl/hopl2007.html#HudakHJW07.

[Hun14] John Hunt. "Further Akka Actors". In: *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer, 2014, pp. 399–412.

[Hut99] Graham Hutton. "A Tutorial on the Universality and Expressiveness of Fold". In: *J. Funct. Program.* 9.4 (July 1999), pp. 355–372. ISSN: 0956-7968. DOI: 10.1017/S0956796899003500.

[IS12] Shams Imam and Vivek Sarkar. "Habanero-Scala: Async-Finish Programming in Scala". In: *The Third Scala Workshop (Scala Days 2012)*. 2012.

[IS14] Shams Imam and Vivek Sarkar. "Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns". In: *ECOOP 2014 – Object-Oriented Programming*. Springer Science - Business Media, 2014, pp. 618–643. DOI: 10.1007/978-3-662-44202-9_25.

[ISO15] ISO/IEC 14882:2011. *P0159R0: Technical Specification for C++ Extensions for Concurrency*. Tech. rep. Standard C++ Foundation, 2015.

[ISO17] ISO/IEC 14882:2017. *Information technology – Programming languages – C++*. ISO, Geneva, Switzerland, 2017.

[Kei14] Hartmut Keiser. "Plain Threads are the GOTO of today's Computing". In: *Meeting C++*. Berlin, Germany. 2014.

[Knu02] Holger Knublauch. "Extreme programming of multi-agent systems". In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems part 2 - AAMAS '02*. ACM Press, 2002. DOI: 10.1145/544862.544912.

[Koh] Christopher Kohlhoff. *Boost.Asio*. 2003 (accessed on 14. 6. 2011.) URL: http://www.boost.org/doc/libs/1.

[Koh13] Christopher Kohlhoff. *N3747: A Universal Model for Asynchronous Operations*. Tech. rep. Standard C++ Foundation, 2013.

[Lan98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1998. ISBN: 0387984038.

[Lee06] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (May 2006), pp. 33–42. ISSN: 0018-9162. DOI: 10.1109/MC.2006.180.

[LN79] Hugh C Lauer and Roger M Needham. "On the duality of operating system structures". In: *ACM SIGOPS Operating Systems Review* 13.2 (1979), pp. 3–19.

[LP16] Guillaume Lazar and Robin Penea. *Mastering Qt 5*. Packt Publishing Ltd., 2016. ISBN: 978-1-78646-712-6.

[LZ07] Peng Li and Steve Zdancewic. "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives". In: *SIGPLAN Notices*. Vol. 42. ACM. 2007, pp. 189–199.

[Mal10] Saša N. Malkov. "Customizing a functional programming language for web development." In: *Computer Languages, Systems & Structures* 36.4 (2010), pp. 345–351. DOI: 10.1016/j.cl.2010.04.001. URL: http://dblp.uni-trier.de/db/journals/cl/cl36.html#Malkov10.

[McC11] Robert Ryan McCune. "Node.js paradigms and benchmarks". In: *STRIEGEL, GRAD OS F* 11 (2011).

[Men03] Daniel A Menasce. "Web server software architectures". In: *IEEE Internet Computing* 7.6 (2003), pp. 78–81.

[Mey15] Scott Meyers. *Effective modern C++*. O'Reilly, 2015, pp. I–XV, 1–315. ISBN: 9781491903995.

[Mey97] Scott Meyers. *Effective C++*. Second. Addison-Wesley Professional Computing Series. Reading, Massachusetts: Addison-Wesley Professional, Sept. 1997. ISBN: 0201924889.

[Mog91] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (July 1991), pp. 55–92. URL: http://dblp.uni-trier.de/db/journals/iandc/iandc93.html#Moggi91.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. *Deprecating the observer pattern*. Tech. rep. 2010.

[Mus97] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Software: Practice and Experience* 27 (8 1997). DOI: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.

[Nis14] Gor Nishanov. "await 2.0: Stackless Resumable Functions". In: *CppCon*. 2014.

[NPS14] Eric Niebler, Sean Parent, and Andrew Sutton. *N4128: Ranges for the Standard Library*. Tech. rep. Standard C++ Foundation, 2014.

[Ous96] John Ousterhout. "Why Threads are a Bad Idea (for most purposes)". In: *USENIX Winter Technical Conference*. Jan. 1996. URL: http://www.cs.utah.edu/~regehr/research/ouster.pdf.

[Par15] Sean Parent. “Better Code: Concurrency”. In: *C++ Russia*. 2015.

[Par17] Sean Parent. “Better Code: Human interface”. In: *Meeting C++*. 2017. URL: https://www.meetingcpp.com/2017/Schedule.html.

[Rod85] David P. Rodgers. “Improvements in Multiprocessor System Design”. In: *SIGARCH Comput. Archit. News* 13.3 (June 1985), pp. 225–231. ISSN: 0163-5964. DOI: 10.1145/327070.327215.

[Rom] Vittorio Romeo. *Capturing perfectly forwarded objects in lambdas*. 2016 (accessed on 14. 3. 2017.) URL: https://vittorioromeo.info/index/blog/capturing_perfectly_forwarded_objects_in_lambdas.html.

[Sch11] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[Sil09] Abraham Silberschatz. *Operating system concepts*. Hoboken, NJ: J. Wiley & Sons, 2009. ISBN: 9780470233993.

[SPH] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. *PEP255: Simple Generators*. 2009, (accessed on 14. 6. 2011.) URL: http://www.python.org/dev/peps/pep-0255/.

[SS76] Guy L. Steele and Gerald J. Sussman. *Lambda: The Ultimate Imperative*. Tech. rep. 1976. URL: http://portal.acm.org/citation.cfm?id=889232.

[Str13] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addisson-Wesley, 2013. ISBN: 0321563840.

[VJ03] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003. ISBN: 9780201734843.

[WH00] Zhanyong Wan and Paul Hudak. “Functional reactive programming from first principles”. In: *ACM SIGPLAN Notices*. Vol. 35. ACM. 2000, pp. 242–252.

Биографија кандидата

Иван Чукић је рођен 14. 3. 1983. године у Београду. Завршио је основну школу "Н.Х. Влада Аксентијевић" са просечном оценом 5.00. Математичку гимназију у Београду је завршио са просечном оценом 4.91. Завршио је основне студије на Математичком факултету, Универзитета у Београду, смер "Рачунарство и информатика" са просечном оценом 9.50. Докторске студије на смеру "Информатика" је уписао 2009. године.

Од 2009. до 2011. је био запослен као сарадник у настави на Математичком факултету, Универзитета у Београду, а 2011. године је унапређен у асистента за ужу научну област "Рачунарство и информатика". Држао је вежбе из следећих предмета: Рачунарска графика, Развој софтвера, Обрада дигиталних слика, Конструкција и анализа алгоритама, Објектно-оријентисано програмирање и Функционално програмирање.

Током основних академских и докторских студија је радио на пројекту *KDE* и постао један од главних програмера овог пројекта. Такође, био је пет пута учесник програма *Google Summer of Code*, два пута као студент и три пута као ментор. Софтвер који је написан у оквиру ових пројеката користе стотине милиона корисника широм света.

Током докторских студија је радио и на Математичком институту САНУ као истраживач сарадник (до 2016. године) и учествовао на пројектима *Cendari* у оквиру *FP7* програма Европске Уније и "III044006" Министарства науке, просвете и технолошког развоја Републике Србије.

Аутор је књиге *Functional programming in C++* у издању *Manning Publishing*, и био је пленарни предавач на конференцији *C++ Siberia 2017*. Поред овог пленарног предавања, био је позван предавач на још седам конференција (у тренутку писања).

Прилог 1.

Изјава о ауторству

Потписани-а Иван Чукић

број уписа 2029/2009

Изјављујем

да је докторска дисертација под насловом

Functional and imperative reactive programming based on the generalization of the continuation monad in the C++ programming language
(Функционално и императивно реактивно програмирање употребом генерализације монаде наставка у програмском језику C++)

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, _____

_____

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Иван Чукић

Број уписа 2029/2009

Студијски програм Информатика

Наслов рада Functional and imperative reactive programming based on the generalization of the continuation monad in the C++ programming language (Функционално и императивно реактивно програмирање употребом генерализације монаде наставка у програмском језику C++)

Ментор Др Ненад Митић

Потписани _____

изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, _____

_____

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Functional and imperative reactive programming based on the generalization of the continuation monad in the C++ programming language (Функционално и императивно реактивно програмирање употребом генерализације монаде наставка у програмском језику C++)

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, _____

_____

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.