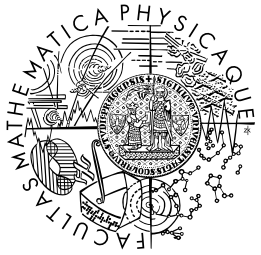

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Petr Kubát

Performance based adaptation of Scala programs

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Doc. RNDr. Tomáš Bureš, PhD.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, July 21, 2017

Title: Performance based adaptation of Scala programs

Author: Petr Kubát

Department: Department of Distributed and Dependable Systems

Supervisor: Doc. RNDr. Tomáš Bureš, PhD., Department of Distributed and Dependable Systems

Abstract: Dynamic adaptivity of a computer system is its ability to modify the behavior according to the environment in which it is executed. It allows the system to achieve better performance, but usually requires specialized architecture and brings more complexity.

The thesis presents an analysis and design of a framework that allows simple and fluent performance-based adaptive development at the level of functions and methods. It closely examines the API requirements and possibilities of integrating such a framework into the Scala programming language using its advanced syntactical constructs. On theoretical level, it deals with the problem of selecting the most appropriate function to execute with given input based on measurements of previous executions.

In the provided framework implementation, the main stress is laid on modularity and extensibility, as many possible future extensions are outlined. The solution is evaluated on a variety of development scenarios, ranging from input adaptation of algorithms to environment adaptations of complex distributed computations in Apache Spark.

Keywords: adaptive systems performance optimization run time prediction

I would like to thank my supervisor, Doc. RNDr. Tomáš Bureš, PhD., for his help, valuable suggestions and ideas. I would also like to thank prof. Ing. Petr Tůma, Dr. for his consultations and advice. My thanks go to Mgr. Vojtěch Horký for setting up the environment to perform Apache Spark tests. Last but not least, I would like to thank David Kuboň for providing useful feedback about the work.

Contents

Introduction **5**

1 Scala programming language **7**
1.1 Methods and functions in Scala 7
1.1.1 Function types 8
1.1.2 Eta-expansion 9
1.2 DSLs in Scala 9
1.2.1 Infix operators 10
1.3 Implicit conversions 11
1.3.1 Implicit arguments 11
1.4 Trait linearization and mixins 12
1.5 Def Macros 13
1.5.1 Building syntax trees 14

2 Solution overview **15**
2.1 Detailed goals 15
2.2 Key parts of the system 17

3 Adaptive development API **19**
3.1 Design of the API 19
3.1.1 Basic API requirements 19
3.1.2 Possible API drafts 19
3.1.3 Functional API requirements 22
3.1.4 Implementation 22
3.2 Usage of the API 23
3.2.1 Basic use cases 23
3.2.2 Covariance and contravariance 25
3.2.3 Generic methods 26
3.2.4 Implicit arguments 27
3.2.5 Extending traits 28
3.2.6 Delayed measuring 28
3.2.7 Usage from Java and other JVM based languages 29

4 Adaptive function selection **31**
4.1 Overview of the selection process 31
4.1.1 Problems of the prediction and selection process 31
4.1.2 Input in the prediction process 32
4.1.3 Input grouping 33
4.1.4 Limiting the historical data age 34
4.1.5 Solving the decision failures 34
4.1.6 Selecting from multiple functions 34
4.2 The invocation chain 35
4.2.1 Run time evaluation 36
4.2.2 Storing and retrieving the evaluation data 36
4.2.3 Selecting a function 37

4.3 Mean based selection strategies . . . 37
4.3.1 T-test for two functions . . . 38
4.3.2 T-test for multiple functions . . . 39
4.3.3 Mann–Whitney U-test . . . 41
4.4 Input based selection strategies . . . 42
4.4.1 Simple linear regression . . . 43
4.4.2 Window-bound linear regression . . . 45
4.4.3 Local regression . . . 47
4.4.4 Window-bound t-test . . . 47
4.4.5 Whitebox model construction . . . 49
4.5 Strategy comparison . . . 50
4.5.1 Input based strategies success rates . . . 50
4.5.2 Mean based strategies success rates . . . 53
4.5.3 Strategy performance . . . 54
4.6 Invocation policies . . . 55
4.6.1 Statistical data . . . 56
4.6.2 Implemented policies . . . 57
4.6.3 Policy builder . . . 61
4.6.4 Policies and groups . . . 62
4.6.5 Possible improvements . . . 62

5 Framework implementation 63
5.1 Goals of implementation . . . 63
5.2 Architecture overview . . . 64
5.2.1 API architecture . . . 64
5.2.2 Internal architecture . . . 65
5.3 Implementation options . . . 66
5.3.1 History storage location . . . 66
5.3.2 Function identifiers . . . 68
5.4 Extracting method name from eta-expansion AST . . . 69
5.4.1 Eta-expansion AST format . . . 70
5.4.2 Retrieving the target from the method call . . . 71
5.4.3 Generating the conversion . . . 72
5.4.4 Extracting method overloads . . . 73
5.4.5 The conversion demonstration . . . 74
5.4.6 Macros in the combining method . . . 74
5.4.7 Extraction evaluation . . . 75
5.5 Module implementation . . . 76
5.6 User interaction and customization . . . 80
5.6.1 Combined function setup . . . 80
5.6.2 Logging, analytics and control access . . . 82
5.6.3 Framework configuration . . . 83
5.6.4 Extending the framework . . . 85
5.7 Framework distribution and usage . . . 86

6 Framework evaluation **87**
6.1 Simple applications . . . 87
6.1.1 Sorting algorithms . . . 87
6.1.2 Matrix multiplication . . . 89
6.1.3 JSON parsing . . . 91
6.1.4 Load balancing . . . 93
6.2 Apache Spark . . . 96
6.2.1 Spark APIs . . . 96
6.2.2 ScalaAdaptive tests . . . 97
6.3 Problems with practical use of the framework . . . 102
6.3.1 Invocation overhead . . . 102
6.3.2 Maintainability . . . 102
6.3.3 Testing and verification . . . 103
6.3.4 Same API requirement . . . 104

7 Related work **105**

Conclusion **107**

Bibliography **111**

List of Figures **115**

List of Tables **117**

List of Abbreviations **119**

Attachments **121**
A The attached CD . . . 121
B Complete list of configuration blocks . . . 121

Introduction

In modern software engineering, performance awareness has become a first class concern. The systems deal with growing amounts of data, with higher emphasis on the real-time processing and with more limited environments for embedded solutions, while we are reaching hardware limitations. It is the responsibility of the developer to assure that his code will not only work correctly, but will also be optimized and fast enough to meet these expectations.

The actual performance-aware programming can often be difficult and problematic for the developer, who is forced to make an assumptions about the speed of his program. The performance in general is, however, a dynamic and not a static feature. Even though it is possible to analyze the algorithm complexity at design-time, the result will always be only an approximation, as it works with abstract elementary computational steps. In reality, the programs often behave in a different way due to the influence of CPU[1] cache, memory locality, I/O[2] operations, parallelism and many other factors.

The only way how to actually find out if given implementation is fast enough – possibly faster than some other variant – is through a thorough testing process covering all the possible inputs and executed in the actual production environment. This process is complicated, costly, and should be repeated after every change in the code, as it can have a previously unexpected impact. It can be partially automatized by introducing performance constraints to the classic unit tests. There is an active research in this area, e.g. [1, 2, 3].

Considering a common case where a programmer needs to decide which implementation option of some functionality to use, the discussed solution has some major downsides. First, he needs to design the tests, select test data, execute them and make decisions based on the results, which is a non-negligible effort, especially in a common software engineering process. Second, the decision that is made is final – once executed, the program will always have to use only that selected implementation. There are many cases where the performance of the implementations varies for different inputs or in different environments, and we would like our application to always use the best possible implementation, in other words, to adapt its execution to these conditions.

To address these problems and to simplify the performance-aware development in general, we propose a completely new approach in development, where the programmer identifies the implementation options of a function in the code using a programming language construct. At execution, the system tracks the performance of the implementations involved and makes a new decision for each run of the function based on the inputs and current trends in the run times.

The goal of this thesis is to design a framework that would allow this way of development and to implement its prototype. It will be tested in various scenarios common for many applications where the approach might be beneficial, and the actual advantages and problems of proposed solution will be evaluated. The target platform of both the design and the implementation is the Scala programming language ([4]), because it has a very flexible syntax suitable for creating

[1] Central Processing Unit

[2] Input / Output operation

language-like constructs while being statically typed with both object-oriented and functional foundations. In addition, it compiles to the Java bytecode and runs on the JVM[3], which makes it portable and open to the variety of Java-based libraries.

Many frameworks for data processing and similar tasks where the adaptation could be employed are either implemented or have interfaces in Scala, mainly for its high expressive powers. An example can be the Spark framework [5], which will be used in the evaluation process.

Structure of the text

In the first chapter of this work, the Scala language features that are not common among current object-oriented programming languages are introduced. Most of them will be mentioned and used in further chapters, so it serves as a brief introduction for a reader who knows basic object-oriented and functional languages, but does not have a deep knowledge of advanced Scala features.

The second chapter serves as a brief overview of the framework developed as a part of this thesis. More detailed goals are stated and a basic structure of the solution is outlined.

The third chapter is dedicated to the framework API[4]. It begins with the requirement analysis and most simple use cases, followed up by possible drafts and their problems. Then, the actual design is presented, along with its evaluation and more advanced scenarios and extensions. The whole chapter is written independently on the framework implementation, and is based only on the requirements and a basic notion of the functionality.

The fourth chapter is more theoretically focused. It introduces the functionality of the selection part of the framework on an abstract level. The whole process of deciding between multiple functions based on the historical observations of their runs is explained. Several algorithms are presented, tested and compared. In addition, more concrete improvements are suggested to increase the selection precision and to speed up the invocation process.

In the fifth chapter, the whole framework implementation is briefly presented. Concepts from the third and fourth chapters are incorporated into an actual Scala program. A couple of problems is mentioned and solved. Last but not least, the options that the framework user has of customizing and possibly extending the functionality are discussed.

The goal of the sixth chapter is to try out the framework in real-life scenarios and to evaluate the benefits that it brings. For that, a couple of different problems will be used, including basic selection between algorithms, JSON parsing or adapting to quickly changing network environment. A part of the chapter is devoted to the Spark framework for distributed data processing, which offers a great potential for adaptive execution.

In the last, seventh chapter, other work that is related to the problem of performance adaptation, measurement, prediction and similar topics is analyzed.

[3] Java Virtual Machine

[4] Application Programming Interface

1. Scala programming language

The adaptive framework introduced in this work was created in Scala [4], an advanced programming language that is built upon the JVM runtime. This chapter will present a brief overview of some of the language aspects that are important for the API design.

The basic paradigm of Scala is object-orientation – it is based on the same principles as Java and its language constructs get mapped to Java bytecode constructs upon compilation. In addition to this, Scala aims to enable full-scale functional development in the same context as well. It is quite a unique concept among the common and production-used languages, which tend to adopt some functional features (higher-order functions, lambdas), but stop at some point and do not add more advanced mechanisms (type classes, monads).

1.1 Methods and functions in Scala

There is an important difference between methods and functions in Scala, even though the names are often used interchangeably in the literature. Methods have the same concept as in Java and other object oriented languages – they are an integral, compile-time part of their class and the only context that they can refer to is their class instance, their arguments and static members. They cannot be passed around, so no additional context is needed. Every invocation has to be performed on a class instance, allowing the compiler to correctly assign the *this* reference. Simple example of a method can be the following:

```
1 class Class {
2   def method(arg: String): String =
3     s"Called method($arg) on $this"
4 }
5 val obj = new Class()
6 println(obj.method("Hello"))
```

Functions are first-class values that have a type and can be passed into other functions or methods, or be part of expressions. Internally, they are represented as closure objects, carrying around their context. The compiled code is stored as a method in the closure type. They are created using lambda expressions:

```
1 val function: (String) => Unit =
2   arg => println(obj.method(arg))
3 function("Hello again!")
4 List("One", "Two", "Three").foreach(function)
```

Each lambda expression is upon compilation converted into an automatically generated class, and its occurrences in the code are replaced by an instantiation of the class. The `obj` reference is *captured*[1] inside the closure, the function is not tied to any class.

The methods represent the basic building block of the object-oriented approach, being the single point that can access or modify the private internal

[1] Becomes an attribute of the class and is passed in using the constructor.

state of the object. The functions are a foundation of the functional part of the language, because they can be passed around as values.

We can create an illusion of functions behaving like methods by declaring them as a class fields:

```
1 class Class(val field: String) {
2   val print: () => Unit = () => println(field)
3 }
4 val instance = new Class("Test")
5 instance.print()
```

In this case, the class contains an immutable field `print` containing a function. The function references another field, which is stored in the function closure. The invocation is done by accessing the `print` field and invoking the result.

1.1.1 Function types

The first-class value functions have to be anchored somewhere in the Scala type system in order for type inference, type checking and other mechanisms to work correctly. The type of a function is unambiguously determined by its signature, i.e. the number and types of arguments and the return type. Consequentially, the function types have to be generic and able to accept different numbers of type arguments.

Scala as a language does not support variadic generic types (as opposed to languages like C++), so there has to be a different type with different number of type arguments for every possible number of arguments of a function. Currently, there are traits for functions accepting 0 to 22 arguments[2]:

```
1 trait Function0[+R]
2 ...
3 trait Function22[-T1, ..., -T22, +R]
```

These traits have syntactic aliases in the language:

```
1 () => R
2 ...
3 (T1, ..., T22) => R
```

Functions with more than 22 arguments cannot exist in Scala and have to be replaced by function accepting tupled[3] arguments. The necessity to represent functions with various argument numbers as multiple traits leads to code duplication wherever we want to interact directly with function types and support any argument number.

Function invocation is represented by an `apply` method declared in all the function trait types with the following signature:

```
1 def apply(v1: T1, ..., vN: TN): R
```

[2]In the code snippets, the type and argument lists will be shortened using the ... notation. The code would have to be expanded in order to be compilable. Analogically, N will be used in place of a number in the type or type argument names.

[3]Scala has generic tuple types and syntactic shortcuts for tuples: `(T1, ..., TN)`.

Scala automatically converts the `()` operator on any instance to an invocation of the `apply` method on the same instance with given arguments. This works for all the class instances, we can make any type callable in Scala.

The default function implementations in Scala are created using lambda expressions, which are compiled to a separate closure class that extends the corresponding function trait type and has the code in the `apply` method implementation. We can, however, create our own function trait implementations that will be treated by all Scala code and libraries in the same way as any function. The only thing required is to provide implementation of the `apply` method.

1.1.2 Eta-expansion

As the strengths of Scala lie in its functional features, it is often handy to be able to convert methods to functions. This process is called *eta-expansion* and is quite straightforward – we just need to create a function which delegates the call to the method:

```
1 val methodFunction: (String) => String = arg => method(arg)
```

Scala has a special operator for the conversion that generates the function automatically:

```
1 val methodFunction = method _
```

And in some expression contexts, this conversion becomes implicit:

```
1 List("One", "Two", "Three").map(method)
```

In this example, the conversion is implicitly applied to an argument of a method in order to meet the required type. The expansion in this case will be always implicit, unless one of the following conditions is met:

1. The method has overloads.
2. There is an implicit typecast required from the expanded type to the parameter type.

Unfortunately, the same kind of automatic expansion is not performed on the target of a method call. The following example will not compile:

```
1 val composed = method.andThen(method)
```

The explicit eta-expansion operator is needed for it in order to work.

```
1 val composed = (method _).andThen(method)
```

1.2 DSLs in Scala

DSL (Domain Specific Language) is a type of a programming language that is narrowly focused on solving a specific class of problems. It offers a higher level of expressiveness and therefore is easier to use than general-purpose languages. There is a large variety of areas where the DSLs are often employed, e.g. mathematical computations (R, MATLAB), database queries (SQL), build scripts (make, Ant), parsing (regular expressions) and many more.

In order to simplify the process of creating a new DSL, it is often implemented inside an existing general-purpose language using objects and references to substitute the keywords of the language. The DSL is represented by a set of class definitions and can be distributed as a library. This significantly speeds up the development process – there is no need to implement the syntax parser or the compiler. Additionally, the end user does not have to have any specialized tools to actually use the language. A program written in such a language will only get compiled (if its necessary) and executed. The DSL code will create a certain object structure in runtime, interpret it and carry out required tasks.

These DSLs became recently very popular for defining configurations, and build and deployment processes – an example could be the Gradle scripting language [6], which is actually a DSL in Groovy, or the Recipe DSL for Chef [7] built atop of the Ruby language.

DSLs can be efficiently implemented only in languages with flexible-enough syntax, which do not limit it too much. Scala belongs to this category of languages, along with for example Groovy. A code in one of the Scala DSLs, the ScalaTest framework [8], can look like:

```
1 an [Exception] should be thrownBy { s.charAt(-1) }
2 greeting should equal ("hi") (after being lowerCased)
```

The above is a valid Scala code that can be compiled and run, and is used by the framework to express unit test assertions.

1.2.1 Infix operators

One of the features that support both the functional flavor of Scala and the DSL building capabilities is the possibility to use methods as infix operators. Any method that has exactly one argument can be called using the following special syntax:

```
1 class RichInt {
2   def plus(other: RichInt) = ???
3 }
4 ...
5 val res = x plus y
```

This syntax is even more powerful if we take into consideration that Scala allows the methods to have non-alphanumeric names:

```
1 def +(other: RichInt) = ???
2 ...
3 val res = x + y
```

This approach enables us to use the infix syntax with existing methods and with methods that were not designed with the intention to be used that way. It has, however, a downside as well – it does not allow us to create an infix operator that would accept a certain type as its first argument without actually changing the definition of that type and adding the method to its definition. Scala does not directly support declaring extension methods[4] on types (unlike C#, Kotlin and

[4]Methods that can be added to existing types without changing them, they have access only to the public API of the type.

other languages), so the only way to solve this problem is to introduce implicit typecast to a custom type and to add the infix operator as a method to it.

1.3 Implicit conversions

Scala supports defining a custom set of implicit conversion methods that will be automatically called throughout the code. The conversion from type `T` to type `U` is a method or a function with one argument of type `T` and a return value of type `U` declared with the `implicit` attribute:

```
1 implicit def myConversion(t: T): U = ???
```

Now wherever an expression of type `T` is either used in a place where the type `U` is expected, or a selector (method or field name) unknown to type `T` but known to type `U` is applied to it, a call of our `myConversion` method is generated to convert the type. The implicit method has to be in scope at the moment of the typecast, otherwise the compiler will not be able to locate it. The common practice is to gather all the implicit conversion methods into an object called either `implicits` or `Implicits`, and let the user import all the methods of the object whenever he needs it:

```
1 import my.project.Implicits._
```

The most typical use case for the implicit conversions is adding new functionality to existing types. This can be used in DSLs as well, like in the following simple example:

```
1 class RichInt(val i: Int) {
2   def times[T](fun: () => T): Unit =
3     Seq.range(0, i).foreach(_ => fun())
4 }
5 implicit def intToRichInt(i : Int): RichInt = new RichInt(i)
6 def greet(): Unit = println("Hello!")
7 4 times greet
```

In other languages (C#, Kotlin), the same goal is achieved using extension methods, which is a less flexible, but more transparent solution.

Implicit conversions are considered a potentially dangerous feature, therefore they have to be manually enabled either by a Scala compiler switch, or by importing `scala.language.implicitConversions` in the scope.

1.3.1 Implicit arguments

In addition to being automatically called for conversions, the implicit methods and values can be automatically passed into method calls as arguments. If an argument of a method is marked with the implicit attribute, it does not have to be specified at invocation. It will be filled automatically by an identifier with matching type that is accessible at the point of the call and is marked implicit itself. The actual argument then, again, behaves as being marked implicit within the function. This can be used to propagate implicit values throughout a chain of calls.

The most common application of implicit arguments is enabling the usage of implicit typecasts with type arguments – a method can set bound on its type arguments by requiring implicit conversion to a different type. In the Scala terminology, the type has to be *viewable* as that type.

```
def bubbleSort[A](list: List[A])(implicit ord: A =>
   Ordered[A]): List[A] = ???
```

There used to be a specialized syntax called *View bounds* in earlier versions of Scala to support this usage, but it has been deprecated:

```
def bubbleSort[A <% Ordered[A]](list: List[A]): List[A] = ???
```

The typical use case of the described scenario is simulating *type classes* from functional languages like Haskell. A generic trait with required methods (representing the type class and its functions) is created:

```
trait Incrementable[A] {
 def increment: A
}
```

Adding a member to the type class is done by creating an implementation of the trait with the type fixed and an implicit method that provides the conversion from the original type:

```
implicit def intIsIncrementable(i: Int): Incrementable[Int] =
   new Incrementable[Int] {
 override def increment: Int = i + 1
}
```

Now, whenever this conversion method is accessible, `Int` can be used as `Incrementable[Int]` and, in addition, can be passed to any generic function that requires the argument to be *viewable* as `Incrementable`:

```
 def incrementValue[K, V](map: mutable.Map[K, V], key: K)
                         (implicit inc: V => Incrementable[V]) =
   data.update(key, data(key).increment)
```

Note that the value of an implicit argument can be specified manually at the time of the call, and it does not necessarily have to be implicit in the calling scope. In such a situation, it will be treated as implicit only inside the method scope.

1.4 Trait linearization and mixins

In Scala, the concept of *interfaces* known from common object-oriented languages is replaced by *traits*. The main difference is that traits can contain method definitions and data, thus being almost equivalent to abstract classes (except for not having parametrized constructors). The possibility to extend multiple types containing data members leads to the *diamond problem* known from C++ and other languages with multiple inheritance. The issue lies in the fact that one ancestor can exist in more inheritance branches, and therefore be extended multiple times.

Scala solves this problem using a concept of *linearization*, a set of rules that can determine a clear and unique single inheritance line for any type (for more details see [9]). A new chain is build specifically for each concrete type that can be instantiated. The `super` keyword gets bound dynamically to the parent in this chain (i.e. a `super` call in a trait might result in different calls for different usages of the trait). Consider the following example:

```
1 class Parent
2 trait TraitA extends Parent
3 trait TraitB extends Parent
4 class Child extends TraitA with TraitB
```

The *linearization* of type `Child` leads to the following inheritance chain:

```
Child <- TraitB <- TraitA <- Parent
```

In tis case, the `TraitB` will have the `super` reference bound to `TraitA`, even though at the point of definition of the `TraitB`, the `TraitA` does not even have to be known.

The multiple inheritance among traits leads to the possibility of creating *mix-ins* – traits containing methods with implementations that can be added to a type to enrich it. An example follows:

```
1  trait IntSequence {
2    val data: Seq[Int]
3  }
4  trait SequenceWithAverage extends IntSequence {
5    def average(): Double = data.sum / data.size
6  }
7  class IntList(val data: List[Int]) extends IntSequence
8
9  val listWithAverage =
10   new IntList(List(1, 5, 200)) with SequenceWithAverage
11 val average = listWithAverage.average()
```

1.5 Def Macros

Def Macros is quite a unique feature among all the widely used programming languages. It allows a library writer to include a code that will be executed upon compilation of the unit which references the library. This basically means that simple compiler extensions in Scala can behave like libraries and can be distributed and used similarly.

The API is very simple – there are methods marked as *macros* in the library code. Whenever a call of this method is being compiled, the compiler invokes the macro implementation, which is a normal Scala method. It will receive an AST[5] of the arguments passed to the original method and produce another AST, which will replace the call in the code.

The only limitation is that the macro cannot be used in the same compilation unit where its declaration and implementation are, as the Scala compiler needs to have access to the already compiled implementation when building the usages.

[5] Abstract Syntax Tree – the product of the syntax analysis of the source code

1.5.1 Building syntax trees

The macro usually has to be able to perform two steps:

1. Parse the argument AST
2. Build the replacement AST

The syntax trees are represented by tree node classes with `apply()` and `unapply()` methods accepting and returning their descendants. We need to deconstruct the input, preferably using pattern matching, and then build an output tree by creating a new chain of nodes. For both actions, it is essential to know the structure of valid syntax trees that correspond to the code.

Unfortunately, there is not a lot of documentation available for the syntax tree structure. The official Scala documentation has only a few examples and the types in the code lack the documentation as well (as of Scala 2.11). One of the most complete overviews of the syntax tree nodes and its building scheme can be found in [10].

One of the possible options to find out how certain expressions are parsed into ASTs is to create a simple macro that will accept any expression, print its AST and replace itself by the same expression again:

```
1 def printAst(arg: Any): Any = macro printAst_impl
2
3 def printAst_impl(c: Context)(arg: c.Expr[Any]): c.Expr[Any] =
     {
4   import c.universe._
5   println(showRaw(arg.tree))
6   arg
7 }
```

The output that is written to *stdout* by a macro implementation is displayed as a warning in the compilation process, so there is no need to run the program to see the results of `printAst()`, it is sufficient to just compile it.

For example, `printAst(x + 1)` produces the following compile-time warning:

```
Warning:scalac: Block(List(),
Function(List(ValDef(Modifiers(PARAM | SYNTHETIC),
TermName("i"), TypeTree(), EmptyTree)),
Apply(Select(This(TypeName("MacroTest")),
TermName("increment")), List(Ident(TermName("i"))))))
```

2. Solution overview

This chapter will provide a basic overview of the solution presented and of the tasks that had to be solved in the process and that will be described with more details in later sections.

2.1 Detailed goals

In the introduction, the basic goal of the thesis was mentioned in a very general manner – as a simple-to-use framework that allows the user to introduce variants into his program. Now, we will go a little deeper and present more detailed goals and the consequences they have for the implementation.

A simple and fluent API

We would like the programmer to be able to simply combine two or more functions or methods (in any combination), resulting into a new function that can be called again.

```
1 def impl1(in: T): U = ???
2 def impl2(in: T): U = ???
3 val impl3: (T) => U = ???
4
5 val function = impl1 _ or impl2 or impl3
```

In the rest of the text, such a function will be called *combined function*. We will also occasionally use the term *adaptive function*, as it will be the API name of the function.

In order to achieve this behavior, we are going to have to work with implicit typecasts from function types and with eta-expansion.

Transparent usage

The function that was created by the combining process (in the rest of the text referred to as *combined function*) should behave just like a normal function – the caller should not have to perform any special action and he should not be able to recognize that multiple implementations are involved.

```
1 val result1: U = impl1(arg)
2 val result2: U = function(arg)
```

If the user, however, decides to combine the function that is already combined from n simple functions one more time, the result should be a combination of $n+1$ functions, not a combination of 2 functions where one of them is the original.

```
1 val newFunction = function or impl4
```

Similar behavior can be achieved by introducing custom function trait implementations and reflecting it in the typecast.

Measuring and storing run times

Whenever the function is called and one of the implementations run, the run time (or other metric) has to be measured and stored. The storage should be statically accessible, because the function can be invoked from different contexts and we would like to share the history across the entire application. In order to identify the implementation in the static storage, we would need an identifier, preferably the name of the method (if a method is used). For that, we are going to have to use a compile-time macro.

Selecting the most appropriate function in different cases

As the key part of the framework, we want to select the function to run each time. The selection should have 2 basic outcomes:

- Select the function that is expected to have better performance on given input in current conditions if we are certain enough
- Select the function that we need to collect data for if we are not certain enough

The selection would be based on historical measurements of each function. We should be able to handle the following situations regarding the history data:

- Functions whose run time does not depend significantly on the input – we expect that if one function has better performance than the other, it will not change with different input or environment
- Functions whose run time depends on some characteristics of the input (e.g. if the sequence is sorted)
- Functions whose run time is expected to be a function of the input size or some similar measurable feature of the input (e.g. length of the sequence to sort)
- Functions whose run time depends on the execution environment and is expected to change over time

This leads to a few consequences. Multiple selection strategies meeting the requirements of these cases will be necessary. In addition, their decisions should reflect the certainty using methods of statistical testing with given significance level. The user will have a way to customize the function combination in order to identify the case and the input characteristics.

Controlling the invocation behavior

The invocation behavior should change in some situations – when we are sure enough that one option is in general better than the other, when we need to gather more data, when we need to save time and cannot perform the selection, or in other cases. Every combined function should have its state and some behavior plan, which would reflect some basic statistics and events on the functions.

We can achieve this by introducing a simple state machine logic and a language in form of a Scala DSL to create simple state machines.

Possibility to extend the framework

The framework should be modular, with its parts being easy to replace or extend. It should be possible for the user of the framework to incorporate his own implementations without having to change the framework code in any way, just by modifying the initial runtime construction.

The following extensions should be possible without changing the code of the framework, only by adding custom implementations:

- Change the metrics that is measured for the functions
- Change the history storage
- Add, replace or extend the selection strategies

For this to work, the framework runtime will be created by composition with no close-coupling between modules. All of the runtime parts will communicate using trait interfaces. A simple form of IoC[1] configuration will be used for the composition.

2.2 Key parts of the system

Based on the extended goals presented in section 2.1, key parts of the solution can be identified and will be described in detail in the rest of the text.

- **API and `CombinedFunction`** – a set of classes and methods that the user will be directly interacting with in order to invoke the combined functions
- **Invocation policies** – a system of simple state machines to manage the invocation process with almost no overhead, handled directly in the *CombinedFunction*
- **Selection strategies** – simple, stateless and isolated strategies that can decide about the most appropriate function to run given the historical measurements of all the functions
- **Function selector and invoker with the history storage** – module that will be responsible for selecting the method using a given strategy, invoking it, measuring the performance metrics and storing and retrieving the historical data; it will be called by the *CombinedFunction*
- **Configuration** – a static description of the function selector and invoker; can be changed by the user

[1] Inversion of Control

3. Adaptive development API

API (Application Programming Interface) is a key component of every framework or library. It defines how a stand-alone piece of code (a function, class, module, or even an entire running application) interacts with its environment. This includes the possible calls or requests, format of the data that are passed in as arguments and the data that are received as the result of the action. In our case, the API will be used to access functionality of our library from the user's own code.

One of the key tasks of this thesis is to research the potential possibilities of adaptive development in the user code using the API of the adaptation framework, to design it to be as fluent and expressive as possible using the Scala programming language features described in chapter 1 and to analyze the problems and limitations that it has. Some key design decisions about the functionality of the entire framework that are tightly connected to the API design have to be discussed as well.

3.1 Design of the API

3.1.1 Basic API requirements

The framework itself should not require much interaction from the programmer. The typical use case of our API could be described in the following way:

> User has a class with two methods, `method1()` and `method2()`, and uses the framework simple API to create a third method in the same class, `method()`, that adaptively uses one of the other two methods. If he wants to change the behavior of the adaptation (method selection) process, he can do it directly in the definition of the `method()`.

The important point is that we want the result of combining the methods to behave exactly like a normal function or method. The caller should not know and should not need to know that he is using an object that is somehow special. This allows introducing adaptivity to existing project just by changing the method implementation, and no new dependency will be introduced in the majority of the code.

We can make a list of the **basic functional requirements**:

1. Create a method by combining two or more different methods (stating that they can be called interchangeably)
2. Define some basic adaptation behavior for the method
3. Make some more complex configuration changes for the whole framework

3.1.2 Possible API drafts

The core part of the whole API and the mechanism that will determine the usability of the framework is the method combination definition. JVM classes in general have the limitation that once they are loaded by a *classloader*, they

cannot be modified, and neither do their methods. Generating new methods at runtime is therefore extremely difficult. An alternative to that would be to create functions instead, which behave like normal Java objects (see section 1.1). From the point of view of the potential caller, a function can be used almost identically as a method.

This leaves us with two basic approaches to the API.

API based on methods

This variant would have to rely at least partially on compile-time actions and would have to work with annotations (in case we do not want to create custom language extensions).

The combination definition could work roughly in the following way:

```
1 @ClassWithCombinations
2 class Functions {
3   @CombineInto("combinedMethod")
4   def method1(arg: Int) = ???
5
6   @CombineInto("combinedMethod")
7   def method2(arg: Int) = ???
8 }
```

There are multiple ways of implementing such an API. We are going to discuss the following two:

1. Using Scala compile-time macros
2. Using aspect weaving

Scala macros would be able to modify the AST of the `Functions` class to contain three methods (`method1`, `method2` and `combinedMethod`) with the same signature. The typecheck would be done at compile time in the macro execution. Notice a few problems and limitation that this solution has:

- The `@ClassWithCombinations` annotation would be necessary for each class that would contain the combined function, because the AST modification of the class can be done only when processing an annotation of the entire class
- Consequentially, it would be possible to combine only methods within one class
- The IDEs do not process the macros upon code inspections and code highlighting, and would mark the combined methods as non existing, which would be extremely inconvenient (this could potentially be fixed by adding a dummy implementation of the combined method, which would impose more work on the user)
- The string-based name of the combined method is error prone – methods could be excluded from the combination due to typos in the name, or an existing method could be overwritten by mistake

The second option of using an aspect weaver[1] would lead to a very similar API with different limitations. The weaver does not allow us to modify the class and to generate a method, but can intercept calls of annotated methods using an annotation pointcut. We could either intercept all calls on the methods combined into the same group and use the selection mechanism every time (which would not allow the user to invoke any of the combined methods without the selection process), or have a dummy method with empty implementation and intercept its calls. Looking for other methods with annotation from the pointcut would require us to use reflection.

API based on functions

This option would work with functions as first-class values (and the objects representing them) entirely at runtime[2]. It might look the following way:

```
1 val function1 = ???
2 val function2 = ???
3 val combinedFunction = /* Expression to combine function1 and
     function2 */
4 val result = combinedFunction(data)
```

The `combinedFunction` would be an object extending the `Function1` trait (see 1.1.1) and thus usable as a normal function.

The main inconvenience of the design is that it works only with functions. Existing methods can be treated as functions using *eta-expansion* (see section 1.1.2), so methods can be combined as well with almost no effort. The other way around, i.e. creating an adaptive method, is a little more complicated. Both will be discussed later.

Apart from these complications, this approach has a lot of advantages. It can be used at runtime in any context to create a temporary function. The type checking and IDE tooling work on it, it can be fluent and flexible. And, in general, this approach fits the functional concept of Scala.

The function combination expression technically has to be a higher-order function (or method). We would like it to conform to the functional Scala design patterns. As an inspiration, lets have a look at a different case where we generate one function from two different ones – function composition. This can be done in Scala using `andThen` method defined on the function type traits (see 1.1.1):

```
1 val timesTwo = (x: Int) => { x * 2 }
2 val plusOne = (x: Int) => { x + 1 }
3 val timesTwoPlusOne = timesTwo.andThen(plusOne)
```

We can achieve even more readable and natural looking code if we take advantage of the infix operator syntax (see 1.2.1):

```
1 val timesTwoPlusOne = timesTwo andThen plusOne
```

Chaining of the methods / operators is possible as well:

[1]For more information about JVM-based language aspect weaving see e.g. the AspectJ project [11].

[2]With a small exception – a compile-time macro is actually used to fetch the method names, as will be explained in 5.4

```
val manyOperations = timesTwo andThen plusOne andThen plusOne
    andThen timesTwo
```

Inspired by this readable and very simple syntax to compose functions which relies only on the basic syntactic features of Scala, we can try to design the combining expression exactly in the same way:

```
val combinedFunction = function1 or function2
```

3.1.3 Functional API requirements

From the two basic drafts presented in section 3.1.2, the function-based API will be implemented in this work. The reason is primarily a more fluent integration into the Scala programming style and functional development in general. Based on this decision, we can make the basic outlines of the API for the adaptive framework to be able to examine its usage patterns and possibilities.

The `or` method, in order to be used as an infix operator, has to be defined on the type that represents the first argument, in this case a function. More specifically, all of the function types mentioned in 1.1.1. Because Scala does not support extension methods, we need to introduce a new custom type for functions containing this behavior, and an implicit conversion from the function traits. We will use the same type as for the resulting combined function – an `AdaptiveFunctionN` type[3]. The following is required:

- `FunctionN` is implicitly convertible into `AdaptiveFunctionN`, creating a default with just one implementation
- `AdaptiveFunctionN` has an operator `or` that will combine it with a different `AdaptiveFunctionN`, creating a new object with concatenation of the implementations
- `AdaptiveFunctionN` is a subtype of `FunctionN` and performs the implementation selection and invocation upon calling `apply()`

3.1.4 Implementation

The type `AdaptiveFunctionN`, which will be the main part of the interface for the user, will be just a trait hiding a concrete implementation. The user should never come in contact with the actual class, as we do not want him to create instances of it manually or to interact with its API. The implementation itself should wrap a list of functions from which to choose, along with some basic configuration, which will be discussed later.

There are two ways for the user how to get a new `AdaptiveFunctionN` instance:

- By using the implicit conversion function

```
implicit def toAdaptiveFunction1[T1, R](fun: (T1) => R):
    AdaptiveFunction1[T1, R]
```

[3]Actually a set of types, one for each N, in this thesis only from 0 to 5

- By calling the `or` method defined on `AdaptiveFunctionN`

```
1 def or(fun: (T1) => R): AdaptiveFunction1[T1, R]
```

The implicit conversions have to be defined for all the function types separately as publicly accessible methods on an object[4]. For this purpose, a special object `Implicits` was introduced – it can be imported into any other Scala source file and the implicit conversions become active in the scope (for more details see 1.3).

The conversion of `FunctionN` to `AdaptiveFunctionN` should check every time, whether the `FunctionN` instance is not already a `AdaptiveFunctionN` instance as well, and if so, just cast it instead of creating a new wrapper. The same should be done by the `or` function, which can just manually call the conversion on its argument.

3.2 Usage of the API

After having designed the basic look of our API, we can examine the usage patterns and go through strengths and weaknesses of the solution.

3.2.1 Basic use cases

Combining two functions

The most simple and straightforward use case is to combine two functions previously stored in variables. We use the implicit type conversion and the `or` method.

```
1 import scalaadaptive.api.Implicits._
2 val combinedFunc = function1 or function2
```

Note that the import of the `Implicits` object is necessary every time and will be omitted in the code examples from now on.

We suppose that `function1` and `function2` have the same type `FunctionN` with the same type arguments. The `combinedFunc` variable will have a type of `AdaptiveFunctionN`. When exposing the resulting function in a public API, it might be better to enforce the `FunctionN` type for the variable or attribute:

```
1 val combinedFunc: (Int) => Int = function1 or function2
```

This will ensure that the `AdaptiveFunctionN` specific API will remain hidden unless having the `Implicits` explicitly imported again.

Combining multiple functions

The `or` calls can be chained and used to combine more than two functions in one expression.

```
1 val combinedFunc = function1 or function2 or function3 or
    function4
```

[4] A Scala concept for singleton.

Combining function twice

A combined function can be combined again, no matter whether treated as `FunctionN` or `AdaptiveFunctionN` in the meantime. The resulting combination will be the same as if the combination was done in one expression[5].

```
1 val combinedFunc1: (T) => R = function1 or function2
2 ...
3 combinedFunct1(...)
4 ...
5 val combinedFunc2 = combinedFunc1 or function3 or function4
```

Combining lambda expressions

Just like named function variables, lambda expressions can be combined directly.

```
1 val combinedFunc = { () => obj1.call(...) } or { () =>
    obj2.call(...) }
```

Combining methods

This is expected to be the most common use case – we have multiple methods and we want to combine them into a single combined function. As mentioned in 1.1.2, the methods can be easily converted in functions, but unfortunately, in case of the implicit typecast and then a method call, the expansion is not done automatically on the first method. The second method, which is an argument, gets expanded.

```
1 val combinedFunc = obj.method1 _ or obj.method2
```

Note that if the `or` method accepted the `FunctionN` argument and relied on the implicit conversion of the argument as well (just like the call target does), its implicit *eta-expansion* would get blocked and an operator _ would be needed as well:

```
1 val combinedFunc = obj.method1 _ or obj.method2 _
```

Creating a method from combined function

Even though functions are so common in Scala, there might be situations where we would need to create a method from our combined function. Thanks to the simplified syntax that allows us to define the method body using a one-line expression, quite an idiomatic way to create this combined method exists:

```
1 def combinedMethod(arg: T) = (function1 or function2)(arg)
```

The method body consists of creating the combined function object and then immediately applying it. Although this does look elegant, it has a major problem – the `AdaptiveFunctionN` instance is created upon every invocation, which leads to an unnecessary overhead and to loss of all the data locally stored in the

[5] Actually, the `or` call chain is exactly the same case, the `AdaptiveFunctionN` instances are created one by one.

instance. In addition, there is no way to access the `AdaptiveFunctionN` internal API.

The preferred, a little less convenient way is to create a private field containing the combined function initialized in constructor and to delegate the method calls to it[6].

```
1  private val combinedInner = function1 or function2
2  def combinedMethod(arg: T) = combinedInner(arg)
```

3.2.2 Covariance and contravariance

So far, all mentioned usages of the `or` method were limited to functions with the same signatures. Quite common case, however, might be combining multiple functions with slightly different argument and return value types, typically one being a specialized version of the other, for example:

```
1 def sort1(data: Iterable[Int]): Iterable[Int] = ???
2 def sort2(data: List[Int]): List[Int] = ???
```

We will examine how this can be done on a simplified case with more combinations:

```
1 val fun1: (Any) => String = ???
2 val fun2: (String) => Any = ???
3 val fun3: (String) => String = ???
4
5 val fun4 = fun3 or fun2
6 val fun5 = fun2 or fun3
7 val fun6 = fun3 or fun1
8 val fun7 = fun1 or fun3
```

We receive compilation error on lines 5 and 8, in the definition of `fun4` and `fun7`. Lines 6 and 7 compile correctly. These are the cases where either:

- Return type of the function passed in as an argument is a subtype of the return type of the target function
- Argument type of the function passed in as an argument is a supertype of the argument type of the target function

The function types in Scala are defined in the following way:

```
1 trait Function1[-T1, +R] extends AnyRef
```

The type arguments representing the function arguments are defined as contravariant and the type argument representing the return value of the function is defined as covariant. This leads to `(String) => String` being a subtype of `(String) => Any` and to `(Any) => String` being a subtype of `(String) => String`. So the `or` method is working flawlessly for the `fun5` and `fun6`, as implicit conversions from subtype to supertype exist for the functions.

In general, the target of the `or` call (i.e. the `AdaptiveFunctionN` instance created by the implicit conversion) determines the type of the resulting function,

[6] It can be thought of as an analogy to the concept of *property* and its *backing field*.

and only its subtypes can be passed in as arguments. This can solve majority of the cases where it would be enough to put the more general function on the left side of the `or` method. In the case where we wanted the combined function to have more general signature, we can perform the typecast on the first function passed in:

```
def sort1(data: Iterable[Int]): Iterable[Int] = ???
def sort2(data: List[Int]): List[Int] = ???
val sortCombined = (sort2 _).asInstanceOf[(List[Int]) =>
   Iterable[Int]] or sort1
```

Potentially, this problem could be solved by having the `or` function generic:

```
def or[J1 <: T1, S >: R](fun: (J1) => S):
   AdaptiveFunction1[J1, S]
```

This, however, leads to problems when being invoked as a macro, so this solution was not used in the final version.

3.2.3 Generic methods

Functions have one major limitation in Scala – they can't have any generic arguments. The signature of a function always has all the argument and return value types specified at compile-time. If we perform the *eta-expansion* on a generic method, we have to specify the type arguments. Otherwise, they will be fixed as the most general achievable within the limits of the covariance and contravariance of the function type (see 3.2.2).

In some cases, this might make the function impossible to call:

```
def makeTuple[A, B](a: A, b: B): (A, B) = (a, b)
val fun = makeTuple _
// fun: (Nothing, Nothing) => (Nothing, Nothing)
```

In other cases, it might lead just to losing the compile-time type check:

```
def defaultCount[A](list: List[A], item: A): Int =
 list.count(_ == item)
val fun = defaultCount _
// fun: (List[Any], Any) => Int
```

As the `or` method performs combination of functions into another function, the *eta-expansion* has to be done at the point of combination, and the arguments should be specified at that point to prevent the described effects. This effectively prevents us from achieving genericity of the combined function in the same way as with methods.

We can, however, take advantage of the fact that the type arguments being explicitly set in the process of eta-expansion can be generic type parameters of an enclosing structure (generic class or generic method). With this approach, there are two possible patterns for the workaround:

1. Defining a generic method by creating the combination and then calling it immediately:

```
def count[A](list: List[A], item: A) = (defaultCount[A] _
   or customCount[A])(list, item)
```

This approach was already mentioned in 3.2.1 and is discouraged from as it leads to repeated creation of the `AdaptiveFunctionN` implementation.

2. Defining a function field inside a generic class:

```
def defaultCount[A](list: List[A], item: A) =
   list.count(_ == item)
def customCount[A](list: List[A], item: A) =
   list.filter(_ == item).map((i) => 1).sum

class ListTools[A] {
  val count = defaultCount[A] _ or customCount[A]
}
```

3.2.4 Implicit arguments

As with a lot of other features, the implicit arguments (described in detail in section 1.3.1) are supported only by methods, not by functions. Therefore, all of them have to be resolved and fixed when performing *eta-expansion*, along with all the type arguments.

Lets consider the case where the implicit arguments depend on the value of a type argument. The implicit arguments can be provided manually, or using an implicit variable from the scope where they are being expanded. If the type arguments are being fixed to concrete types, we usually have the implicit implementations at our disposal:

```
def radixSort(list: List[Int]): List[Int] = ???
def bubbleSort[A <% Ordered[A]](list: List[A]): List[A] = ???

val sort = radixSort _ or bubbleSort[Int]
```

The implicit argument might, however, often be used to set requirements about the type argument (in the sense of type classes from other languages), like in this example:

```
def sort[A](list: List[A])(implicit ord: A => Ordered[A]):
   List[A] = ???
```

As explained in 1.3.1, the goal is to propagate the implicit arguments through several different scopes to the point where the type argument is fixed and the implicit value can be provided. The solution from section 3.2.3 to expand the type argument to a type argument of an enclosing scope can be extended to the implicit arguments, as the principle is exactly the same – the implicit argument can just be repeated in the method scope:

```
def sort[A](list: List[A])(implicit ord: A => Ordered[A]):
   List[A] = (sort1[A] _ or sort2[A])(list)
```

Or, preferably, at the level of classes:

```
class Sorter[A](implicit ord: A => Ordered[A]) {
  val sort = sort1[A] _ or sort2[A]
}
```

3.2.5 Extending traits

Traits in the role of interfaces represent key element of object-oriented programming approach. Having a combined function exposed through a trait to the rest of the system is expected to be one of the most common usage patterns, as it would allow replacing the combined function with a fixed implementation and vice versa.

A trait in Scala can define both methods and properties. In case of properties, the implementations will override the getter and setter of the property (which can be automatically generated). This allows us to define functions as parts of traits as well, and the Scala syntax will make them almost unrecognizable from methods in the trait user's point of view.

```
1 trait TestTrait {
2   def testMethod(arg: List[Int]): List[String]
3   val testFunction: (List[Int]) => List[String]
4 }
```

If we design our own traits for the use in the application, we can intentionally expose the logic using only functions. In that case, implementing the trait using combined functions does not represent any problem. In case that we need to use an existing trait that contains methods, we have to implement them and delegate the call to the combined function using the techniques described in section 3.2.1.

```
1 class TestImpl extends TestTrait {
2   override val testFunction: (List[Int]) => List[String] =
      impl1 _ or impl2
3   override def testMethod(arg: List[Int]): List[String] =
      testFunction(arg)
4 }
```

3.2.6 Delayed measuring

In some specific cases it might be useful to delay the measurement, or, in general, to measure invocation of a different function than the one that has multiple implementation, as they can affect something else than their own runtime. Some example use cases might be:

- Configuration, or generation of configuration
- Expression tree building
- Query building
- Method chain building with lazy evaluation

It is obvious that our API, which was designed to remain as simple as possible, does not support this case. We need to slightly extend it.

Decision context

Supposing we already have a mechanism to mark the actual function to measure, we encounter another problem. Making a decision when executing the function with multiple implementations automatically generates a context of the decision – all measured function invocations that were affected by the decision. When measuring the invocation, the framework needs to know to which context it belongs to in order to be able to assign the measured data to the run history of the corresponding function.

The contexts can interleave and we cannot match the decision with the invocation by time or location. It is impossible to decide which invocation belongs to which decision automatically – we need support from the user. He will have to explicitly state the context whenever invoking the measured function. By doing this, he will also pinpoint the moment of decision that affected the invocation.

Practically, we need to introduce a special object – the *invocation token* – which will represent the context. The `AdaptiveFunctionN` instances will have a special, modified version of the apply method, which will not trigger the measure of the selected function, but will generate an invocation token that will be used to measure future function runs. We chose to make it accessible using the `^()` operator, to simulate the fluent application.

```
1 val getConfig = getFastConfig _ or getSlowConfig
2 val (config, measure) = getConfig^()
3 ...
4 measure(() => run(config))
```

This approach has one main disadvantage compared to the rest of the API – it breaks the transparency. The user has to know that he is working with the `AdaptiveFunctionN`, not just normal function. His assistance is, however, required by the nature of the problem.

3.2.7 Usage from Java and other JVM based languages

Even though Scala has a very high level of possible interoperability with Java and in general, Scala classes and methods can be called directly from Java code (being just a classes and methods in bytecode), the direct usage of described API from Java is not possible. The reason is that it depends heavily on the syntactic features that Scala provides, including implicit conversions and compile-time macros.

There is, however, a very simple way how to utilize the method combining in a Java or any JVM based language project. Supposing we have multiple methods in Java that are interchangeable and that we want to adaptively combine, we can introduce a simple Scala wrapper class. This wrapper will use the described framework API to create an adaptive function with said Java methods as the implementation options. The function should be exposed through a method, like described in section 3.2.1, as Java does not have the concept of functions and the invocation would have to be done using the `apply` method call.

4. Adaptive function selection

The key purpose of the adaptive framework is to always invoke a function that is expected to have the best performance possible in a given environment and with given inputs. By *function performance* we mean any measurable description of the qualities of the function. It can be the actual run time of the function, memory consumption, number of I/O operations, number of threads created, execution counts for various parts of the code, etc. All these factors can be valuable and be the goal of the system run optimization.

This thesis is focused on the task of optimizing the function run time or time complexity in general. The core of the framework, however, is designed to be extensible by modules that can analyze the function run in different ways. More on this topic will be covered in chapter 5.

Note we will suppose some basic knowledge of mathematical statistics in this chapter. A nice introduction covering all the concepts that this text works with can be found in [12].

4.1 Overview of the selection process

Suppose we have functions $f_1, \ldots, f_n$, all of them interchangeable, and an input in[1]. We need to decide which of the functions to select and invoke. To be able to do that, we need to formulate predictions about run times of the functions under given circumstances, and then use the function with the lowest prediction. As the internal structure of the functions is unknown to us, the only data we can base the prediction on is the information we collect during previous executions of the individual functions. The exact form of these *historical data* will be discussed later.

4.1.1 Problems of the prediction and selection process

There are three main factors we should take into account when predicting run times and using them to select a function:

- The run times may depend on the input in
- The run times may depend on the environment
- The predictions are never precise, they have certain probability based on the amount and the character of the historical data

The *input dependency* problem is being solved in many works about program execution time predictions. Certain features of the input are usually analyzed and a model is constructed describing the relation to the run time. The model can then be used to predict the times for new inputs. This approach was used in [13, 14]. The authors of [15] propose clustering the inputs prior to creating any models.

[1] If the original Scala function has multiple arguments `arg1, ..., argN`, we will consider them to be a tuple `(arg1, ..., argN)` to get a single input object.

The *environment dependency* is more complicated – there are no factors that we could watch and use to classify the runs or to include in the prediction model. The simplest solution is to give the history records a limited duration after which they expire. The environment changes are not expected to occur very often, which makes the most recent data the most suitable for modeling and predicting. This technique is suggested among the adaptive framework use cases in [16].

As for the *uncertainty* problem, works in this field tend to use statistical methods to express the confidence of the prediction. In [15], confidence intervals are used for this purpose. The comparison of predictions used for selection should aways take these information into account and reject to select any function in case of uncertainty to prevent potentially wrong decision. Such a case should be solved by executing the function with the least data to gather more observations.

4.1.2 Input in the prediction process

The input of the function can be a complex structure with multiple factors contributing to the run time[2]. Common prediction techniques ([13, 14]) use simplified model of the input *in* represented by a vector of *features* $(t_1, \ldots, t_n)$.

Based on this approach, we will suppose that for each Scala function exists g so that for each input *in* modeled by $(t_1, \ldots, t_n)$ holds $t = g(t_1, \ldots, t_n) + \varepsilon$, where t is the runtime of the function on input *in*, and ε is an error represented by a random variable with certain distribution, which we, for simplicity, assume to be normal.

Now, the *features* can be divided into 2 basic categories:

- *Features* with distance metric
- *Features* without distance metric

If the *feature* has a distance metric, i.e., is comparable (typically sizes of structures), we can construct models that use the it to analyze the trends and make predictions about inputs with feature values that were not processed before. The *features* without the distance metric (usually enumerations and discrete configuration values) cannot be treated this way, and we need previous observations for a given value before making any assumptions.

To integrate these concepts into the framework, two mechanisms were included. First, to replace the vector of *features* with distance metric, a single integer called *input descriptor* is used. The user can define a *descriptor function* that extracts the *input descriptor* from the input *in* for given function. Then, at the beginning of the selection process, the *descriptor function* is applied and the resulting value is used by the predicting model. In addition, the *input descriptors* are stored in the run history as well, so a model can be built upon it and used in the selection process. We will suppose that exists h so that for every input *in* with *input descriptor* x holds $t = h(x) + \varepsilon$.

If we wanted to be more precise in the dependency modeling, we should be using a tuple of all the distance-based features as our *input descriptor*. That

[2]Consider for example a general graph – for most of the algorithms, both number of vertices and edges have to be taken into account

would, however, lead to significantly more complex prediction models, and therefore we decided to keep the framework prototype simple by limiting ourselves to one factor. More complex models working with multiple features can be added to the framework in the future.

As for the *features* with no metric, the simplest way of addressing the issue is to treat different combinations of these *features* the same way as if they were different functions. They should have completely separate historical measurement records, and the strategies can build separate models for them, using the *input descriptor* on top of that. A possibility of grouping the inputs was therefore added and will be discussed in a separate section.

4.1.3 Input grouping

In [15], an automatic method of clustering the inputs is used before formulating any predictions about the run time. This is done because of the fact that inputs with common factors are in general expected to lead to similar run times, or at least to run times that can be described by the same, simpler model. In addition, it might be necessary to split the inputs using the values of *input features* with no distance metric, as explained in 4.1.2.

Due to the performance limitations and the overall complexity of the solution, we will not include the algorithmic clustering. The user will, however, be able to specify a method of assigning the inputs to groups. The historical records for inputs from different groups will be held separately, and upon selection, only the ones from the corresponding group will be used.

Group selection

The group selection function that determines a group for a given input is called a *group selector* and has to be provided by the user when creating the combined function. Groups are identified by a custom type that can be either an integer, or a special *NoGroup* value.

The grouping should be primarily based on the discrete values that affect the function behavior – enumerations, boolean flags and similar, as these cannot be included in the *input descriptor* based model of the selection strategies. It can, optionally, include the *features* covered by *input descriptor* as well, in order to divide the range of all its possible values into more smaller ranges, which can increase precisions of the models. Some of the most common ways how to group values for the *input descriptors* might be:

- Logarithmic – orders of magnitude
- Linear – tens, hundreds, thousands
- Fixed – predefined finite number of groups

The groups in general should not be too small, as the selection process relies only on the data from the group. The advantage of groups with growing size (e.g. the logarithmic grouping) is that there is a lot of groups in the areas where the functions tend to compete in the performance (smaller inputs).

4.1.4 Limiting the historical data age

The ε in the function h from section 4.1.2 technically represents the influence of the execution environment. We suppose that it is a random variable from a normal distribution, whose parameters are determined among other factors by the environment state. If it gets changed, the distribution of the error changes as well. In such a case, we should not be using the observations from before and after the change together in the models that suppose the same distribution for all the observations (which is a common assumption).

Therefore, whenever we expect the environment to change often and the run times to fluctuate significantly (e.g. due to network traffic, computation nodes, etc.), we limit the history to the most recent observations in the selection process, which will reduce the probability of model failures. Not only can this lead to more significant results of the statistical selection strategies, but it also means that the framework will be able to quickly adapt to a new environment in case that the function performances get dramatically different.

Limiting the historical data age can also help with recovering from problems caused by occasional performance fluctuations. Such a fluctuation in one of the historical observations can influence the model and cause wrong decisions. If we, however, use only fresh data to construct it, the problematic performance measurement will not be used after a certain time. No long lasting errors can therefore be introduced in the system.

For these reasons, we need to have the historical run records marked with timestamps. The user of the framework will, optionally, provide a maximum age of a record that should be taken into account for a combined function. Upon invocation, the records that are older than this limit will be filtered from the history vector of each one of the functions. This should not lead to any significant overhead if the vectors preserve the ordering in which they are built, having the most recent data at the end. The time period itself is up to the user to configure and should depend on the expected frequency of the function runs and on the environment behavior.

4.1.5 Solving the decision failures

As mentioned before, we would like the decision process to take into account the certainty of the decisions, i.e., refuse to give a decision when the possibility of committing an error is too high. If this happens, it can either mean that the run times of the functions involved are very close to each other, or that the models are very inaccurate. The first situation does not mean any problem for us, as using any of the functions is equally good. In the second case, we would need to increase the precision of the model, ideally by adding more data. Therefore, the most useful action is to select a function with the least historical data.

4.1.6 Selecting from multiple functions

The goal of the decision process is obvious when selecting from two functions – compare the predictions and select the one that is better (taking into account the confidence of the predictions), or refuse to select when the decision cannot be done. Upon selection among three or more functions, the situation becomes less

clear. Ideally, we would like to be able to identify a function that is significantly better than each one of the remaining functions. The problem is that there might be a function that is significantly better than some of the remaining functions, but not better than all of them. This function cannot be selected, because we are not able to rule out the possibility that there is a faster option.

This could theoretically be solved by performing tests between subsets of functions – if we found out that all functions in certain subset are better than all the other functions with given certainty, we could limit the selection to the subset, or even choose from it randomly. The problem is that such a procedure would lead to an extremely high number of comparisons, which would increase both the chance of an error in some of them and the selection overhead.

Therefore, we will consider only the simplest option where one function has to be significantly better than all the others, and the remaining results will be treated as ambiguous with none being selected. This can, unfortunately, cause problems in a situation where we have two functions with equally good performance and one with worse performance. As we are not able to make a decision, we will keep cycling through all three functions.

4.2 The invocation chain

We have discussed the selection process using prediction techniques on a theoretical level. Now, we are going to have a look at the actual steps that have to be done upon invocation of a combined function. Suppose we have a combined function f with implementations $f_1, \ldots, f_n$. We are able to invoke f with an input *in*. It is expected to run one of the implementations $f_1, \ldots, f_n$ with *in* and to return result *out*.

The basic steps required to do so are the following:

1. Use the *group selector* to find the group identifier g and the *descriptor function* to find the *input descriptor* x for *in* (if either of the functions is not defined, the corresponding variable will have a special *undefined* value)
2. Locate the history vectors $d_1, \ldots, d_n$ of functions $f_1, \ldots, f_n$ for group g
3. Filter out records older than the maximum age (infinite by default) from each vector d_i into d'_i
4. Select the function f_k to be executed based on $d'_1, \ldots, d'_n$ and x
5. Invoke f_k, fetch the result *out* and evaluate its run time
6. Add the new evaluation result to d_k and update it for group g
7. Return *out* to the caller

In the following sections, we will discuss some of the steps of the process in more details.

4.2.1 Run time evaluation

The evaluation consist of gathering useful data from the function run. In this text, the data will be represented only by the function run time, but in the future, they might be replaced and might potentially contain more information that can help with classifying the function run and make better predictions in the selection process. An example of such data might be the execution counts of certain code paths of the function, like in [13]. The collection would, however, require a non-trivial instrumentation of the executed code.

Wall clock time and CPU time

When talking about function run time (or an execution time of a function), there are two types of values that we could be observing:

- ***Wall clock time*** – time elapsed between entering and leaving the function
- ***CPU time*** – time that the CPU actually spent executing our function

The *wall clock time* is always higher than the *CPU time*, because it includes not only the time when CPU is executing the function code, but also the time when the executing thread is waiting for its turn in time-sharing multitasking operating system or sleeping on a blocking I/O operation, synchronization primitive, or for any other reason. This means that the *wall clock time* also gets affected by concurrently running processes, network load, and other environment-based factors, and tends to vary much more between multiple invocations.

For the purposes of the adaptive framework, it might seem that the *CPU time* would be more appropriate, as it gives clearer results not affected by the state of the executing environment. The truth is, however, that many of the use cases of the framework require the thread sleeping time to be included in the measurement, as the functions run times are determined mainly by the duration of an I/O operation (e.g. database queries, network requests, etc.), so using *CPU time* would not give us the necessary results.

It is also quite difficult to determine the *CPU time* – it requires support from the operating system with tracking the time that every thread has spent in execution. For this reason and for the reason stated above, the selection process in this text is based on the *wall clock time*. It might be, however, interesting as a future extension to implement measuring of both of the times and allowing the user to decide for each combined function which time should be used.

The *wall clock time* is measured by fetching high-precision system time in nanoseconds right before calling the function apply method and right after returning from the call. The result is the run time in nanoseconds. This measurement is precise enough for all the use cases of the framework, because it is targeting mainly functions with non-negligible time complexities.

4.2.2 Storing and retrieving the evaluation data

After having invoked the function and evaluated its run, the evaluation data have to be stored before passing the return value back to the caller. The history of all the run evaluations for a given function on inputs from a given group is kept together as a vector of records. Each record consists of at least the following:

1. The *input descriptor* of the input for a given run

2. The evaluation data (in our case, the wall clock run time) of a given run

3. The timestamp of a given run

Whenever a new evaluation data are obtained, a record is created and added to the end of the history vector. Some additional metrics might be cached for the whole history record. In such a case, the metrics have to be updated as well.

For simplicity, we suppose that the run histories of the actual implementations will be shared between all the combined functions that use them. More on that along with different options will be discussed in chapter 5.

4.2.3 Selecting a function

The actual selection process, as described in 4.1, can be implemented in many ways using many different models. It is basically an isolated algorithm with the following input:

1. History data vectors $d_1, \ldots, d_n$ corresponding to functions $f_1, \ldots, f_n$, where $d_i = ((x_{i,1}, y_{i,1}), \ldots, (x_{i,l_i}, y_{i,l_i}))$, for $x_{i,j}$ being the *input descriptors* and $y_{i,j}$ being the corresponding run times of historical runs (the *input descriptors* can be *undefined*)

2. Current *input descriptor* x

It should return $k \in 1, \ldots, n$, which will determine the function f_k to run.

These algorithms are called *selection strategies*, and we suggested a few of them based on different approaches to predicting the run times and selecting between various predictions. They will be described in the following two sections, and for each that we included in the framework, a simplified pseudocode implementation will be presented at the end of the description.

4.3 Mean based selection strategies

In some cases, the function run time does not depend on the input, or the relation is not significant enough. For simplification, we suppose that the function complexity is constant in such a case. Using the same formalism as in 4.1.2, for run time t on input *in* with *input descriptor* x holds $t = h(x) + \varepsilon$ for a h constant, i.e. $\forall y\ g(y) = c$. Therefore, $t = c + \varepsilon$, where ε is the error which we suppose has a normal distribution with a zero mean. Under these assumptions, t is a random variable with normal distribution and mean equal to c, which is the expected constant run time of the function.

The *mean based* strategies will use the sample data to determine the means of the function time distributions with given certainty and use them as the expected run times in the decision process. The current *input descriptor* x, as well as the ones included in the history measurements, will be ignored in the process. They will not be therefore mentioned in the pseudocode algorithm input.

4.3.1 T-test for two functions

To compare means of two samples from certain distribution and make conclusion about their equality, statistical tests are commonly used.

Suppose we have two samples of run times for the two functions involved, $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$. Next, suppose that these samples come from normal distributions with means μ_1 and μ_2, respectively. These means are unknown for us, and neither are the variances, which we suppose to be different. The goal is to test the means of the two samples against each other.

Based on these requirements, we will use a *two-sample t-test*. Standard Student's t-test works only under the assumption that the sample variances are equal, so it cannot be used in this case. A generalized version can be applied, namely the Welch's (nonpooled) t-test [17], which is not as strong, but can work with samples with different variances. More detailed reasoning about the test selection can be found in [12].

The default hypotheses for the test are:

$$H_0 : \mu_1 = \mu_2$$
$$H_1 : \mu_1 \neq \mu_2$$

and the test statistics used:

$$T = \frac{\bar{X}_n - \bar{Y}_m}{S}$$

where $\bar{X}_n$ and $\bar{Y}_m$ are the sample means and S is the following:

$$S = \sqrt{\frac{S_X^2}{n} + \frac{S_Y^2}{m}}$$

with S_X^2 and S_Y^2 being the sample variances:

$$S_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$$

$$S_Y^2 = \frac{1}{m-1}\sum_{i=1}^{m}(Y_i - \bar{Y}_m)^2$$

This test statistics can be estimated according to [12] using Student's t-distribution with *df* degrees of freedom for:

$$df = \frac{(\frac{S_X^2}{n} + \frac{S_Y^2}{m})^2}{\frac{S_X^4}{n^2(n-1)} + \frac{S_Y^4}{m^2(m-1)}}$$

The hypothesis H_0 will be rejected in favor of hypothesis H_1 with the significance level α if

$$|T| > t_{df}(1 - \frac{\alpha}{2})$$

The $t_{df}(1 - \frac{\alpha}{2})$ is the $(1 - \frac{\alpha}{2})$-th quantile of *t-distribution* with *df* degrees of freedom.

Rejecting H_0 in favor of H_1 means that the sample distributions have significantly different expectations, i.e., one of the functions is expected to give better results than the other. We can then simply compare the sample means and determine, which of the functions has lower expected run time and should be selected to run with given certainty (the probability of $1-\alpha$). If we cannot reject the H_0, the strategy is not able to determine which function is better with given certainty and should not give a result.

Selection algorithm

The strategy is parametrized by the selection significance level α and by the secondary strategy to be used in case of decision failure.

Input: Historical data vectors d_1, d_2 where $d_i = [y_{i,1}, \ldots, y_{i,l_i}]$

```
1: equalityRejected ← twoSidedTTest(d_1, d_2, α)
2: if not equalityRejected then
3:     return useSecondaryStrategy(d_1, d_2)
4: if mean(d_1) < mean(d_2) then
5:     return 1
6: else
7:     return 2
```

4.3.2 T-test for multiple functions

The selection strategy that was described in 4.3.1 works only with two functions. If we want to select from three or more, we need to modify the strategy.

Suppose we have k samples $X_{1,1}, \ldots, X_{1,n_1}, \ldots, X_{k,1}, \ldots, X_{k,n_k}$ from normal distributions with unknown means $\mu_1, \ldots, \mu_k$ and unknown variances. We would like to compare the expectations and find out whether there is a sample whose mean is significantly different from the others.

Common techniques for testing hypotheses concerning the means of multiple samples involve *ANOVA*[3] or its modified version, the *Kruskal-Wallis test* (both described in [12]). The tests are based on variance comparison, assume that the samples have similar standard deviations, and can decide about the following hypotheses:

$$H_0 : \ \mu_1 = \mu_2 = \cdots = \mu_k$$

$$H_1 : \ \exists i, j, \ \mu_i \neq \mu_j$$

These multiple-sample tests are usually preferred over performing multiple t-tests, because it usually offers higher strength of the test for the same significance. In our case, however, this approach is not very suitable. We need to find out whether there is one sample deviating significantly from all the others – *ANOVA* guarantees only the existence of some deviation.

Based on these observations, it is necessary to come up with different hypotheses. We will consider a simpler task first.

[3] Analysis of variance

Testing one function against the others

Suppose we want to test μ_i against all the other expectations:

$$H_{i,0} : \exists j \neq i,\ \mu_i = \mu_j$$
$$H_{i,1} : \forall j \neq i,\ \mu_i < \mu_j$$

$H_{i,0}$ is basically saying that there is at least one other sample with the same mean, and the one-sided alternative $H_{i,1}$ puts μ_i as the lowest of all the means.

We can perform a simple t-test of $H_{i,j,0} : \mu_i = \mu_j$ against $H_{i,j,1} : \mu_i < \mu_j$ $\forall j \in \{1, \ldots, k\},\ j \neq i$. It is clear that the $H_{i,0}$ can be rejected in favor of $H_{i,1}$ only if all of the $H_{i,j,0}$ were rejected in favor of $H_{i,j,1}$. Note that the $H_{i,j,1}$ is an alternative hypothesis of a one-sided test – we need to take it into account when performing the test and modify the critical values.

The procedure of combining statistical tests usually requires artificially lowering the significance level of the individual tests with the goal of keeping the significance level of the whole set of tests reasonably low. These methods are based on the assumption that committing a type-I error[4] in any of the individual tests leads to a type-I error overall. Additionally, the probability of committing at least one gets higher with the number of tests.

In our case, however, a type-I error in one of the individual tests does not necessarily lead to a type-I error in the whole test. Suppose that $H_{i,0}$ is true, so for an $l \geq 1$ exist $m_1, \ldots, m_l$ such that $\mu_i = \mu_{m_1} \wedge \cdots \wedge \mu_i = \mu_{m_l}$. In order to commit a type-I error, we need to reject $H_{i,m_j,0}$ for all $j \in \{1, \ldots, l\}$ by committing the type-I error in the individual tests l-times. At the same time, we need to decide correctly about the remaining $k - l$ tests. So the probability of committing a type-I error overall is $\alpha^l(1-\alpha)^{k-l}$, and consequentially, the significance level of the entire test gets even lower than α.

As the α decreases with growing k, the test gets more pessimistic – it will require more data and more certainty in order to reject $H_{i,0}$. If we wanted to keep the significance level at exactly α, we would need to apply a reverse correction – artificially increase α. The exact method how to do so would, however, be quite difficult to derive, as there are many cases to consider with unknown probabilities. The assumption is that the number k of the functions involved will be low enough not to affect the significance too much.

Testing multiple functions for a presence of a deviating one

Now we are able to decide about a function whether it is significantly better than all the others. We can use it to decide about the following:

$$H_0 : \forall i,\ \exists j \neq i,\ \mu_i = \mu_j$$
$$H_1 : \exists i,\ \forall j \neq i,\ \mu_i < \mu_j$$

The one-against-others will be performed for all the functions one by one. If $\exists i$ so that $H_{i,0}$ can be rejected in favor of $H_{i,1}$, the H_0 can be rejected in favor of H_1 and we can use the i-th function as our selected candidate. If such an i does not exist, we cannot reject H_0 and no function will be selected.

[4]Incorrectly rejecting a true null hypothesis, more details can be found in [12].

In this case, the type-I error in any of the elementary tests leads to a critical error of the selection process. If we wanted to remain precise about the significance levels, we would need to apply a correction at this moment to lower the significance of the one-against-others test. As that test is, however, again composed from multiple individual tests, the situation gets complicated and deriving a precise correction is out of the scope of this work. In addition, such a correction leads to lowering the test strength, i.e., the H_0 being rejected less often. This could severely affect the strategy behavior, maybe even make it unusable for some higher numbers.

We will, therefore, use the unmodified significance level for simplicity, as the expected usages combine mostly three functions at maximum. The task of determining a suitable correction for the selection case, either theoretically using probabilities in the composed tests, or practically by observing behavior with different values, is left as a future extension.

Selection algorithm

The strategy is parametrized by the selection significance level α and by the secondary strategy to be used in case of decision failure.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [y_{i,1}, ..., y_{i,l_i}]$

1: **for** $i = 1$ to n **do**
2: $\quad better \leftarrow true$
3: $\quad$ **for** $j = 1$ to n **do**
4: $\quad\quad equalityRejected \leftarrow$ oneSidedTTest(d_i, d_j, α)
5: $\quad\quad$ **if** not $equalityRejected$ or $mean(d_i) > mean(d_j)$ **then**
6: $\quad\quad\quad better \leftarrow false$
7: $\quad$ **if** $better$ **then**
8: $\quad\quad$ **return** i
9: **return** useSecondaryStrategy$(d_1, ..., d_n)$

4.3.3 Mann–Whitney U-test

The t-tests used in strategies from sections 4.3.1 and 4.3.2 require the historical measurement data to come from a normal distribution. Even though it was one of our assumptions at the beginning, it does not necessarily hold in reality, as the fluctuations in the run time depend on many factors of the environment. We might try to use a test that is nonparametric, i.e., does not require any standardized and parametrized distribution of the data. One of these tests is the Mann–Whitney U-test (for details see [12]), which only requires the sample observations to be independent.

Supposing we have the samples $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$ from distributions with respective means μ_1 and μ_2, the hypotheses are going to be exactly the same as for the t-test:

$$H_0 : \ \mu_1 = \mu_2$$
$$H_1 : \ \mu_1 \neq \mu_2$$

The test itself is based on sorting all the observations from both samples together by their values and assigning them ranks corresponding to the position

in the sorted sequence (from 1 to $m + n$). Then, the ranks of the first and the second sample are added up, forming R_1 and R_2, respectively. The test statistic is the following:

$$U = min(U_1, U_2)$$

$$U_1 = R_1 - \frac{n(n+1)}{2}$$

$$U_2 = R_2 - \frac{m(m+1)}{2}$$

Now, U has, in case of valid H_0, a known distribution with critical values that can be either looked up for small fixed values, or approximated using a normal distribution for higher values. Therefore, we can reject the H_0 for $U < U(\alpha, n, m)$, where $U(\alpha, n, m)$ is the tabulated or approximated critical value for the significance level α and given sample sizes.

The strategy can be extended to three or more functions exactly in the same way as the t-test (see section 4.3.2), the individual tests involved will have to be changed to one-tailed (by using doubled α when looking up or approximating the critical value of U) and an overall α correction should be used.

Selection algorithm

The strategy is parametrized by the selection significance level α and by the secondary strategy to be used in case of decision failure.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [y_{i,1}, ..., y_{i,l_i}]$

```
1: for i = 1 to n do
2:     better ← true
3:     for j = 1 to n do
4:         equalityRejected ← oneSidedUTest(d_i, d_j, α)
5:         if not equalityRejected or mean(d_i) > mean(d_j) then
6:             better ← false
7:     if better then
8:         return i
9: return useSecondaryStrategy(d_1, ..., d_n)
```

4.4 Input based selection strategies

Input based strategies use the historical measurements along with corresponding *input descriptor* to construct a model that approximates the non-constant function h (see 4.1.2). A prediction for a new *input descriptor* can be derived from the model and used to decide which of the functions will have better performance on given input.

These strategies rely on the existence of *input descriptors* and cannot be used when the *descriptor function* is not specified.

4.4.1 Simple linear regression

The first strategy supposes that the function h can be approximated precisely enough using a linear function h' in the following form:

$$h'(x) = a'x + b'$$

This approximation is a *simple linear regression model*. The slope a' and intercept b' of the linear function can be determined using the least-squares criterion, which minimizes the squares of the distances of the actual observations from values predicted by the model (for detailed description see [12]).

The major advantage of the model is its simplicity and thus minimal overhead when generating the regression during the selection process. In addition, the model can be easily cached for fixed set of samples and updated when adding new data without having to construct it all over again.

The main problem is that applying the linear regression model to a case where the relation is not linear leads to non-trivial errors that increase with the range that we are trying to cover with the model. Figure 4.1 shows samples of run times of the basic selection sort algorithm, which has complexity of $O(N^2)$, on sample arrays with 0 to 30000 integers. Collected data obviously match the quadratic function graph, denoted by the dotted curve. The straight line shows the linear regression model. As we can see, the predictions based on this model would be quite inaccurate, especially if we tried to predict run time on an input significantly larger than the samples.

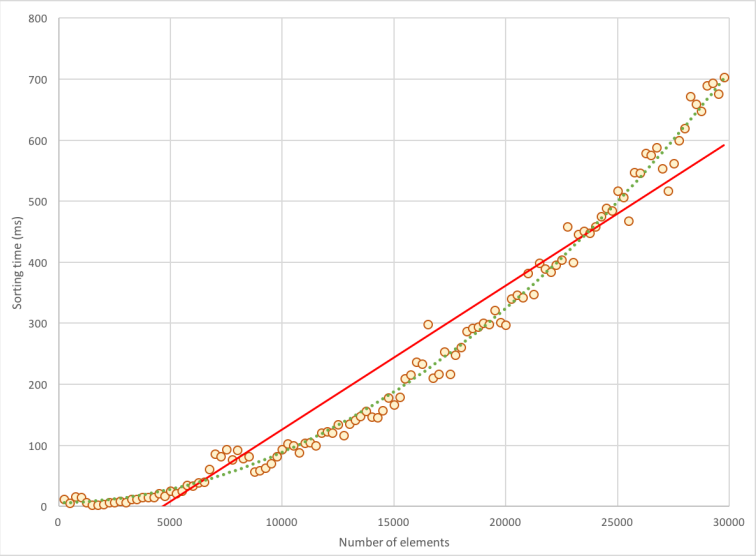

Figure 4.1: Linear regression model of selection sort runtime samples.

The model can then be used to formulate simple predictions, $a'x+b'$ being the prediction for *input descriptor* x. Next, we need to find a way how to determine the certainty of the prediction to be able to select the best function with given certainty using a set of these models.

Confidence interval based selection from two functions

Suppose we have the linear regression model $h'(x) = a'x + b'$ that approximates the function $h(x) = ax + b$ constructed using n data samples x_i, y_i (in our case,

x_i are the *input descriptors* and y_i are the run times). If we assume that the errors $\varepsilon_i = y_i - h'(x_i)$ of our sample from the regression model values are random variables with normal distribution, we can perform some observation about the model.

Firstly, the standard errors $se(b')$ and $se(a')$ of the intercept b' and the slope a' can be derived from the model. In addition, the statistics $t_a = \frac{a'-a}{se(a')}$ and $t_b = \frac{b'-b}{se(b')}$ have a t-distribution with $n-2$ degrees of freedom. This means that we are able to construct confidence intervals and perform t-tests for a and b. The default tests hypotheses have the following form:

$$H_0 : \ a = a_0$$
$$H_1 : \ a \neq a_0$$

We basically test equality of a (or b) and a fixed value a_0 (or b_0) – this test can be used to verify if the data have a specific linear relation. If we put $a_0 = 0$, we can find out whether the observations y_i are constant.

None of these tests and statistics, however, allows us to test the certainty of the prediction – we need to consider both the intercept and the slope estimation certainty, and, in addition, the distance of the x we want the prediction for from the mean $\bar{x}$. The reason is that if we commit an error in the slope estimation a', the corresponding error in predictions increases with the distance from $\bar{x}$.

With these facts, a confidence interval on level $1-\alpha$ for the mean of the actual values (y) for x can be constructed. According to [12], the width of the interval is the following:

$$w_{x,\alpha} = t_{n-2}(1 - \frac{\alpha}{2}) s_e \sqrt{\frac{1}{n} + \frac{(x-\bar{x})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}}$$

where $\bar{x}$ is the mean of x_i, $t_{n-2}(1-\frac{\alpha}{2})$ is the $(1-\frac{\alpha}{2})$-th quantile of *t-distribution* with $n-2$ degrees of freedom, and s_e[5] can be computed as:

$$s_e = \sqrt{\frac{SSE}{n-2}}$$

and

$$SSE = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (y_i - y_i')^2$$

for $y_i' = h'(x_i)$.

It now holds for any x that the mean of $y = h(x)$ lies in $[a'x + b' \pm w_{x,\alpha}]$ with probability $(1-\alpha)$.

Now in the situation where we have two regression models and we want to find the best prediction of x, we can construct these confidence intervals for both models and:

- If the intervals do not overlap, the predictions have at least a $(1-\alpha)^2$ probability of correctly setting the order, i.e. the worse prediction actually being the worse function

[5] Standard error of the estimate

- If the intervals overlap, there is a non-trivial chance of the predictions being mixed up by error

In the first case, we can decide that one function is significantly better and select it. In the second case, we cannot decide with given certainty.

Confidence interval based selection from multiple functions

The described method allows us to decide between two functions with given certainty. Now, when choosing among n functions with regressions $h'_1, \ldots, h'_n$ for given x, we are looking for the k for which the following holds:

- The confidence interval on level $(1-\alpha)$ for prediction $h'_k(x)$ does not overlap with any other of the confidence interval for prediction.
- $\forall j \neq k : h'_k(x) < h'_j(x)$

It can be done by constructing all the regression models and then $\forall i$ performing the comparison described in 4.4.1 of h_i with $h_j, \forall j \neq i$. If h_i is better than all of the h_j, we can choose the corresponding function.

The strength of this decision is influenced by the fact that we perform multiple confidence interval constructions – the errors an error in any of them can influence the result of the selection, se the significance gets lower in general. If we wanted the results to be precise, we would have to perform the correction of the significance level α. Just like in 4.3.2, we will leave the significance correction analysis as a potential future work.

Selection algorithm

The strategy is parametrized by the selection significance level α and by the secondary strategy to be used in case of decision failure.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [(x_{i,1}, y_{i,1}), ..., (x_{i,l_i}, y_{i,l_i})]$, *input descriptor* x

```
1: for i = 1 to n do
2:     r_i ← createRegression(d_i)                ▷ Uses the least squares method
3: for i = 1 to n do
4:     better ← true
5:     for j = 1 to n, j ≠ i do
6:         (k_1, l_1) ← predictionInterval(r_i, x, α)
7:         (k_2, l_2) ← predictionInterval(r_j, x, α)
8:         if (k_1, l_1) overlaps (k_2, l_2) or k_1 > k_2 then
9:             better ← false
10:    if better then
11:        return i
12: return useSecondaryStrategy(d_1, ..., d_n, x)
```

4.4.2 Window-bound linear regression

To be able to apply the strategy described in section 4.4.1 on a wider variety of functions, a simple approach to lower the approximation errors can be taken.

Instead of creating a regression model for the entire set of historical measurements at once, we can split it into more subsets and create different regressions for them. Figure 4.2 shows ranges 5000 to 10000, 10000 to 15000 and 15000 to 20000 of the input sizes of the data from 4.1. As we can observe from the second and third range, the data tend to be much closer to the linear regression line. The first range shows that a less significant fluctuation in the entire set has a lot bigger impact on the model in a smaller subset.

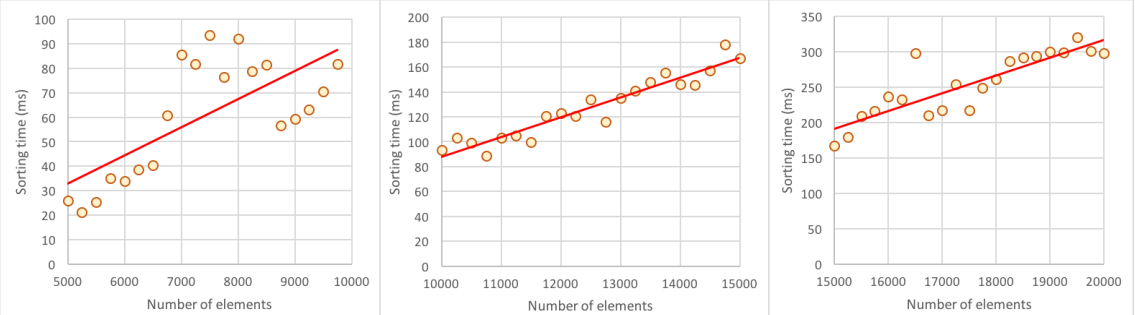

Figure 4.2: Examples of subsets of the data from figure 4.1 with corresponding linear regressions.

When selecting a function for a specific *input descriptor* x, a window around its value of specific width w can be created and the regression model built using the data from the said window. We do it by selecting only the historical samples (x_k, y_k) where $|x_k - x| < w$ when constructing the regression.

There are multiple ways how to decide on the window width w. Using a fixed number across all the functions and all the data samples would not be recommendable, as the functions can vary in their *input descriptors* by orders of magnitude. The width could be derived as a fraction of the width of the whole data set: $w = \frac{max(x_i) - min(x_i)}{p}$, where p would represent in how many parts we want it to be split.

More flexible approach is to dynamically determine the p. Suppose we want the average window to contain r records. We find out what is the average distance between two neighboring records as $d = \frac{max(x_i) - min(x_i)}{n-1}$ and then set the $w = r * d$. This way, the window size will always be adapted to the density of the dataset.

The window limitation might seem similar to the grouping described in 4.1.3, but there are two major differences. First, the windows are based on the *input descriptor*, whereas the *group selector* might use different features of the input. And second, the groups are deterministic and do not move with the *input descriptor* and change their size based on historical data. Both these concept can be used at the same time.

Selection algorithm

The strategy is parametrized by the selection significance level α and by the average record count per window avg.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [(x_{i,1}, y_{i,1}), ..., (x_{i,l_i}, y_{i,l_i})]$, *input descriptor* x

1: **for** $i = 1$ to n **do**

2: $\quad dist \leftarrow (max(x_{i,j}) - min(x_{i,j}))/(l_i - 1)$

3: $w \leftarrow dist * avg$
4: $d'_i \leftarrow [(x_{i,j}, y_{i,j}); \forall j : |x_{i,j} - x| \leq w/2]$
5: **return** useLinearRegressionStrategy$(d'_1, ..., d'_n, x)$

4.4.3 Local regression

The approach explained in 4.4.2 can be extended further – the locally constructed linear regression models might be used to build up a model for the entire data set. This leads to a model that is non-linear, i.e., the function that approximates the original data relation is not a linear function.

One possible regression method that behaves this way is *LOESS* (Local Regression, [18, 19, 20]). It uses techniques similar to the least-square regression on local subsets of the data set, and then smooths up the resulting curve. The output is a *Loess Curve*, an artificially constructed function that does not necessarily have to be expressed by a formula. This is one of the advantages of the model – the original relation between *input descriptors* and the run times did not have to be expressed by a simple function in order to be captured correctly.

The *LOESS* model can be used to predict the function run times. It is, however, a lot more complicated to derive the prediction confidence intervals. Therefore, only the values will be used in this selection strategy, which could lead to some wrong decisions.

There are other disadvantages of this strategy as well. First of all, the entire local regression model has to be recomputed whenever there is a new record in the dataset, the model cannot be simply updated like the *simple linear regression*. Secondly, it is much more computationally complex and the model construction takes non-trivial time, so it has a negative impact on the framework overhead, especially when used with short and simple functions. The last downside of *local regression* is its inability to predict further behavior past the minimum and maximum data sample.

Selection algorithm

The strategy is parametrized by the secondary strategy to be used in case of decision failure.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [(x_{i,1}, y_{i,1}), ..., (x_{i,l_i}, y_{i,l_i})]$, *input descriptor* x
1: **for** $i = 1$ to n **do**
2: $m_i \leftarrow$ interpolateLoess(d_i)
3: $p_i \leftarrow$ predictValue(m_i, x)
4: **if** $p_i = NaN$ **then** ▷ The prediction can fail
5: **return** useSecondaryStrategy$(d_1, \ldots, d_n, x)$
6: **return** $argmin(p_i)$

4.4.4 Window-bound t-test

The mean based t-test selection strategy described in sections 4.3.1 and 4.3.2 works under the assumption that the run times of a specific function have a normal distribution, and the prediction of run time is simply the mean of all

the times measured. This could be true only for the functions with constant complexity – whenever the function run time depends on the input, the samples measured have their distribution influenced by the distribution of the inputs.

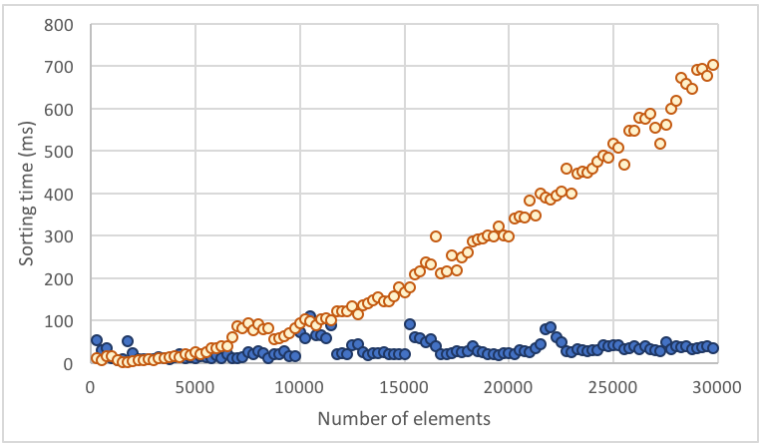

Figure 4.3: Run times of quick sort algorithm (dark dots) and selection sort algorithm (light dots) by input size.

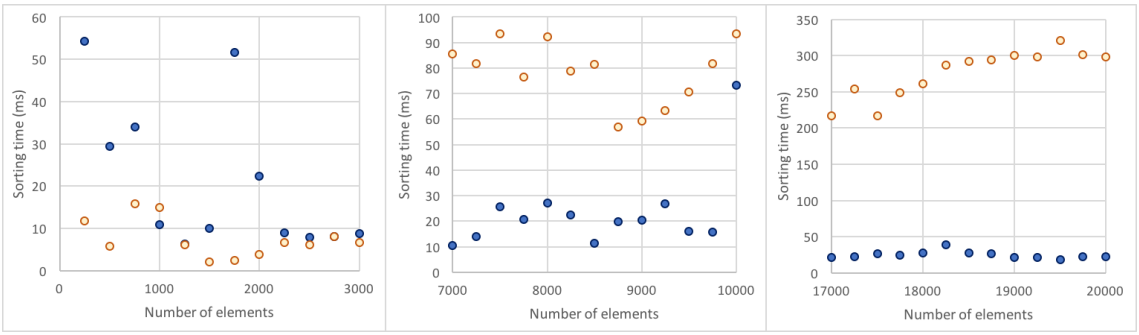

Figure 4.4: Examples of subsets of the data from figure 4.3

If we wanted to use the t-test strategy on the functions with non-constant complexity, the same approach as with the *simple linear regression* can be taken. Instead of assuming that the relation in the small window is linear, we want it to be constant and normally distributed. We can then predict the run time using the mean of the measurements from within a very small range of *input descriptor* values. Consequently, it is extremely important to keep the window size as small as possible, so that these assumptions held at least approximately. The t-test significance levels and strengths will be approximate as well, but our goal is to get reasonable selection behavior, we do not need to be precise.

Figure 4.3 shows the run times measured for the quick sort and selection sort algorithms on random arrays of sizes between 0 and 30000. Obviously, there is a relation to the input size, so the simple t-test should not be applied in this case. In figure 4.4, there are the same data in three different windows – 0 to 3000, 7000 to 10000 and 17000 to 20000. As we can see, in a window of this size, the run times are almost constant and the random fluctuations are more significant than the actual relation to the input size. We can apply the t-test as described in sections 4.3.1 and 4.3.2 to these windows and consider the result to be relevant enough with respect to the input.

The window sizes can be determined in a similar way as in section 4.4.2. If we reduce it to a constant 0, we will get a technique in which only the results for inputs with the same descriptor are tested. When there are none, we refuse to select and wait until data are gathered. This might be useful in cases where we know that the function will be called repeatedly on a limited set of input variants, and basically rules out the prediction factor, as we need to have actual observation for each input before making any conclusions. Similar approach is taken in [16]. The same goal, however, can be achieved more efficiently using the grouping mechanism (see 4.1.3).

Selection algorithm

The strategy is parametrized by the selection significance level α and by the average record count per window avg.

Input: Historical data vectors $d_1, ..., d_n$ where $d_i = [(x_{i,1}, y_{i,1}), ..., (x_{i,l_i}, y_{i,l_i})]$, *input descriptor* x

1: **for** $i = 1$ to n **do**
2: $dist \leftarrow (max(x_{i,j}) - min(x_{i,j}))/(l_i - 1)$
3: $w \leftarrow dist * avg$
4: $d'_i \leftarrow [(x_{i,j}, y_{i,j}); \forall j : |x_{i,j} - x| \leq w/2]$
5: **return** useTTestStrategy$(d'_1, ..., d'_n)$

4.4.5 Whitebox model construction

The methods that were described so far are examples of the blackbox techniques – they do not analyze the function itself, the models are based only on the relation of the run times and the *input descriptors*.

The techniques used in the research projects focused on formulating accurate predictions about program run times (like [14, 13, 21]) often build the model using more detailed information about every run. These extra data can be obtained by examining and instrumenting the function code. Key structures (loops, branches, method invocation, ...) can be identified and execution counts can be tracked. Then, a model that captures the relation between the input features, the described structure execution counts and the run time can be constructed. This added dimension might lead to more accurate model and therefore better predictions.

In [13], a Mantis framework for high accuracy program performance prediction was implemented. The basic principles are similar to the approach described above. It was tested on a series of data gathered from runs of the ImageJ application, and compared with a blackbox prediction model based on interpolating a polynomial of degree three. The prediction error of the whitebox approach was 5.5%, the blackbox approach error rate reached more than 35%.

This approach has, however, a few problems concerning the intended use cases of our framework. The model will not improve the predictions if the majority of the execution time is spent waiting on an I/O operation. Specifically, it will not help with any of the cases where we are selecting a database query, remote server to connect to, etc. In addition, it would require the method code to be instrumented at runtime, which would lead, together with the model construction, to a significant overhead.

For these reasons and because of the complexity, the whitebox model was not implemented in the framework, but it represents a potential future work that could be done on the topic.

4.5 Strategy comparison

In general, the input based strategies do not handle well more historical results for the same *input descriptor*, and are impossible to use when there is none specified. The following rules should be used to decide between input based and mean based strategies:

- **Dependency on the input**
  If the function does not have any input, or the input does not affect the expected run time (the complexity is constant), mean based strategies should be used.

- **Expected input values**
  If the function is expected to be invoked with a discrete and very limited set of inputs, mean based strategy is preferred. On the other hand, if the function will be called repeatedly with randomly distributed inputs, the input based strategies are a better choice.

Considering these facts, the framework will contain both an input based and a mean based strategy, and will automatically use the input based one if the user specified an *input descriptor*, and the mean based one otherwise. The user will be able to override this choice if he wants.

The actual strategies used should be easily replaceable, but some have to be chosen as default for the common uses. Therefore, a practical comparison of the selection success rate and overheads of the strategies that were described and implemented follows.

4.5.1 Input based strategies success rates

To compare the efficiency of the selection strategies, we will perform two tests. The first one will be based on an artificial function with linear complexity defined by the following lambda expression:

```
1 i => {
2   val j = (i * k).toInt
3   var acc = 0
4   Seq.range(0, j).foreach(l => {
5     acc = acc + (l * Random.nextInt(1000))
6   })
7   acc
8 }
```

Where `i` is the input of the function and `k` is a constant factor which can be used to slow the function down.

Analogically, we will define a quadratic function (using a similar lambda, only with second line replaced by `val j = (i * i * k).toInt`).

Now, the test will be based on combining two instances of the same function[6], one with $k = 1$ and the other with some $k > 1$, perform a series of runs and track how many times will the selection strategy choose the worse one (i.e. the $k > 1$ one).

For each test, we will generate 100 sequences, each one containing 200 integers between 100000 and 500000 in case of the linear function, or 100 and 500 for the quadratic function. One sequence will represent one test run – the measurement history will be flushed and the function will be invoked on the members of the sequence one by one. The strategies are set in the way that the first 60 runs will use a round-robin selection to gather 30 data samples for each function before starting the actual selection. It means that the strategy should perform 140 selections for each test run. We are going to count how many times the worse function is selected. The baseline solution in this case would be to keep using the round-robin technique, which would lead to the worse function being selected 50% of the time, 70 times out of 140.

Now, we will use the following values for k: 4, 2, 1.5, 1.2, 1.1, 1.01, and the following selectors:

- Window-bound t-test (WBTT) with α set to 0.05 and 0.25
- Linear regression (LR) with α set to 0.05 and 0.25
- Window-bound linear regression (WBLR) with α set to 0.05 and 0.25
- Local regression (LOESS)

For each combination, all 100 test runs were performed and an average percentage of the worse function selection was computed, the results for the linear function can be seen in table 4.1 and for the quadratic function in table 4.2. A surprising fact might be the overall better results for the non-linear case, which should be more difficult to model. The reason for this fact is that there is a dramatically lower range of inputs (100 to 500 instead of 100000 to 500000) in order to keep the test run time realistic. In addition, the run time differences between individual inputs grows much faster, so the error factor involved does less damage to the model. In general, it seems that functions with faster-growing complexities get more precise predictions.

As to the individual strategies, it might be unexpected that the linear regression has quite poor results, even in the case where the original function complexities were linear. Almost 16% of selections of a function that is four times slower in the $\alpha = 0.05$ case. This is caused by the fact that eventual measurements with extreme deviations influence the whole model, not just a local part of it, which might lead to an eventual series of wrong decisions. Therefore, linear regression strategy is not recommended to be used in general.

On the other hand, the absolute winner is the local regression, which is able to maintain extremely low error rates (6.2% for the linear functions, 7.4% for the quadratic functions) for the second-minimal case $k = 1.1$, and push it even lower with growing k. Neither one of the other strategies got below 15% not only for $k = 1.1$, but even for $k = 1.2$.

[6]Note that the actual implementation requires either using two different lambdas, or using custom identifiers, more on that topic in section 5.3.2

	WBTT (0.05)	LR (0.05)	WBLR (0.05)	WBTT (0.25)	LR (0.25)	WBLR (0.25)	LOESS
1.01x	52.4%	49.6%	51.5%	47.1%	49.6%	50.1%	43.6%
1.1x	49.4%	48.5%	37.2%	33.7%	44.6%	33.5%	6.2%
1.2x	40.1%	46.0%	29.0%	26.3%	41.6%	27.8%	3.5%
1.5x	30.2%	38.9%	13.9%	13.7%	25.6%	9.9%	2.9%
2x	17.6%	25.4%	7.8%	4.3%	14.4%	4.7%	2.4%
4x	2.9%	15.7%	5.0%	0.1%	9.7%	2.7%	2.5%

Table 4.1: Percentage of times the strategy selected the worse one out of two linear functions (average from 100 test runs, each containing 140 selections).

	WBTT (0.05)	LR (0.05)	WBLR (0.05)	WBTT (0.25)	LR (0.25)	WBLR (0.25)	LOESS
1.01x	52.3%	49.4%	44.6%	50.5%	50.3%	49.5%	44.6%
1.1x	44.9%	43.1%	28.9%	33.3%	37.8%	26.2%	7.4%
1.2x	38.2%	38.1%	16.3%	17.8%	31.2%	15.7%	4.0%
1.5x	16.6%	28.7%	8.4%	7.9%	18.5%	6.4%	2.5%
2x	8.8%	20.0%	4.1%	2.7%	13.4%	2.4%	2.3%
4x	3.5%	19.6%	3.0%	0.1%	14.6%	1.5%	2.3%

Table 4.2: Percentage of times the strategy selected the worse one out of two quadratic functions (average from 100 test runs, each containing 140 selections).

The window-bound strategies are somewhere in the middle – they perform very well for higher k $(2, 4)$, which is the main concern, and reasonably for the other sizes. The main difference between them being that the window-bound linear regression improves steadily with growing k, whereas the window-bound t-test reaches better results for the largest $k = 4$, but improves dramatically at the end and has worse results for lower k.

We can see that in all the cases, lowering the selection significance (by increasing the α) can lead to slightly better results overall, especially for larger values of k. The success of local regression, which does not employ certainty in the decision making process, supports this observation. It seems that the assumption of selection errors having a bad influence on the model and leading to chains of bad decisions was not confirmed by this test. It might be, however, different for real-life scenarios with more variable run times, environment changes, etc., and we should keep that in mind and be careful when lowering the α for better

selection results.

4.5.2 Mean based strategies success rates

Similarly to the input based strategies, we are going to perform a selection success rate test for the mean based strategies. As we need only a function with fixed non-trivial execution time with a possibility to slow it down a little, we are going to reuse the linear function from section 4.5.1, fixing its input to 200000. Again, we are going to combine an instance of the non-slowed function with the instances of function slowed by a factor k.

The values 2, 1.5, 1.2, 1.1, 1.01 will be used for k, along with the following selectors:

- T-test (TT) with α set to 0.05 and 0.25
- U-test (UT) with α set to 0.05 and 0.25

For each combination of k and a selector, 100 test runs will be performed, each one consisting of invoking the described combined function 200 times and removing all the history records afterwards.

The results can be seen in table 4.3. It might seem obvious that the u-tests deliver better results, and again, lowering the α improves the result as well. In order to get more insight into the results, we should take into account another factor – the number of test runs, where the overall success rate was worse than the baseline success rate, i.e. the 50% of the round-robin selection. This is shown in the table 4.4.

	TT (0.05)	**UT (0.05)**	**TT (0.25)**	**UT (0.25)**
1.01x	49.8%	49.5%	49.8%	47.1%
1.1x	46.6%	27.0%	39.1%	25.5%
1.2x	42.2%	18.5%	25.6%	14.3%
1.5x	11.6%	0.0%	2.6%	0.2%
2x	1.9%	0.0%	0.0%	0.0%

Table 4.3: Percentage of times the strategy selected the worse one out of two constant functions (average from 100 test runs, each containing 140 selections).

It looks like the t-test is more conservative, it refuses to make the decision more often, which leads to switching the functions in the round-robin fashion. On the other hand, the u-test risks more, which leads to better results overall, but there are some runs that are very bad – for $k = 1.2$, 15 runs out of 100 were worse than 50%, and 10 out of these 15 runs had a 0% success rate, which means that the model selected the worse function every time.

	TT (0.05)	UT (0.05)	TT (0.25)	UT (0.25)
1.01x	0	1	19	7
1.1x	0	15	5	20
1.2x	0	15	4	13
1.5x	0	0	0	0
2x	0	0	0	0

Table 4.4: The number of test runs (out of 100) where the strategy had a success rate worse than 50%.

The conclusion from the observations is that employing u-test (or increasing the α) increases the expected success rate in a common case, but brings a non-trivial risk of constructing a wrong model which leads to a sequence of errors that might last very long.

4.5.3 Strategy performance

When deciding for one of the strategies, there is another concern that we should keep in mind. The strategies tend to be quite complex and their execution time might become non-trivial. As they need to process the entire run history of each of the functions, their selection times increase over time, as more historical measurements are collected.

For some of the strategies, it is possible to compute and cache the statistical data in advance, at the moment of storing newly measured data, allowing them to avoid processing all the records again during the selection. From our strategy list, this is possible to be done for the linear regression and the t-tests. Unfortunately, neither the window-bound strategies, nor the local regression support this approach.

To compare the practical overheads, we ran the selection using each of the strategies 20000 times in a row (while evaluating the runs and adding new data in the process) for the task of deciding between quick sort and selection sort algorithm on sequences of 0 to 2000 numbers. The window-bound strategies had the window size calculated to contain 25 records in average. The resulting averages of selection strategy run times are in table 4.5.

The results are showing that local regression has extremely high run times that can grow above 50 milliseconds for situations with larger number of historical observations. An overhead of this size on every selection would be too high for most common usages, so the local regression strategy, even though it has the best success rate, should be used with extreme caution and only in cases where it is clear that overhead this high cannot damage the system performance.

As for the remaining strategies, it seems that caching speeds up the selection a little more than 10 times for this number of runs. The window-bound strategies cannot be improved by the caching (as the relevant statistics are different for every window), but they have the lowest execution time of all the strategies without

	Not cached	Cached
Linear regression	1.68	0.15
Window-bound LR	0.91	-
Window-bound TT	0.72	-
LOESS	57.47	-
T-test	1.32	0.10
U-test	5.91	-

Table 4.5: Average strategy selection time (in ms) in a sequence of 20000 consecutive selections.

cache, and therefore are very usable for the input based case. Regarding the mean based strategies, the t-test is almost 5 times faster than the u-test in the non-cached version, and more than 50 times faster when using the cache, so it is clearly a better choice if performance is the main concern.

4.6 Invocation policies

The selection process and all the strategies described in sections 4.3 and 4.4 are quite complex and have non-trivial overhead, especially after collecting a large amount of historical data, as discussed in section 4.5.3. In a common scenario where we expect the system to come to a decision about the best function (either overall, or separately for every group), it would be suitable if we could stop the selection process at a certain point and keep using only the most favored function with no extra overhead. Or, to use it most of the time, with only occasional attempts at the selection in order to detect possible changes in the system (response times, I/O operation duration, etc.).

Similarly, it might be desirable to control the overhead time spent on the selection, or, in general, on invoking the functions using selection (which might lead to a bad decision), either per a specific time period, or with respect to a specific action of the system (user request, processing unit, etc.). All of these decisions should be taken at a higher level than the run time history analysis performed by the selection strategies.

Invocation policies are a concept that tries to address these cases, with the main purpose of lowering its invocation overhead. Every combined function has an *invocation policy* associated with it. In the function selection chain, the policy gets evaluated right after invoking the combined function and is supposed to quickly decide how to proceed with the invocation. There might be scenarios where a faster ways could be taken without even using any of the strategies and analyzing the historical data.

The possible results of the policy evaluation are the following:

- **SelectNew** – Select function using the selection strategy, executing the chain described in section 4.2

- **GatherData** – Gather more data for the least-executed function (function selection is skipped, but the run time is measured and the data are stored to the history)
- **UseLast** – Use the function selected last time (without measuring run time and storing it to the history)
- **UseMost** – Use the most selected function (without measuring run time and storing it to the history)

We can notice that these results require only the basic statistical data about previous selection processes. This is the key idea of the policies – the decision and the execution should depend only on such a simple statistics.

The policy evaluation process also replaces the current policy with a new one. This technically creates a state machine – the state of the function is represented by the active policy, and whenever the function is invoked, a result is produced and new policy becomes active. This corresponds to producing output and moving to a new state in a state machine.

The advantage of the policy system over a regular state machine is its extensibility and reusability – there are no rules directly in the state machine (i.e. here in the combined function). All the logic of deciding on the result and the next policy is in the policy node itself. So the entire behavior of the state machine is defined by the initial policy, which should be specified by the user upon creating the combined function. As a result, reusable policies that are parametrized can be used in various chains of policies, smaller chains can be put together, etc.

4.6.1 Statistical data

The only input that the policy receives when making decision is a statistical summary of previous selection results of the combined function. The process itself is supposed to be fast and reflect just the trends in the selection, not to replace the whole selection strategy decision making as described in sections 4.3 and 4.4, so the statistical data do not contain the entire run history of all the functions involved.

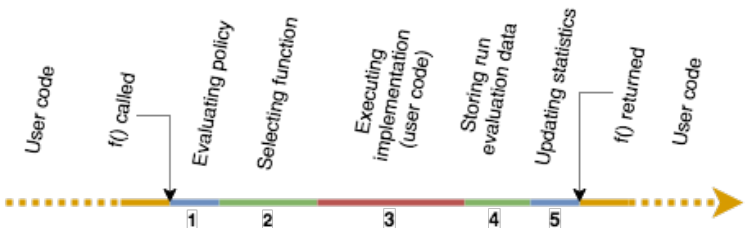

Figure 4.5: Process of combined function invocation.

The policies are a suitable tool to limit the potential damage of the selection process to the system. For this, it is necessary to observe and to store various times connected with the selection and execution. Figure 4.5 shows the whole process with policy being evaluated with the **SelectNew** result, split into individual parts. Upon every invocation, the following is tracked:

- Selection and run history storage time (2 and 4)
- Function execution time (3)
- Total time spent on the **SelectNew** result (2, 3 and 4)
- Total time spent on the **GatherData** result (3 and 4, 2 is skipped in such a case)

Sums of these times are kept in every combined function. The policy can work with these numbers and analyze the changes that happen to them over time. In addition, the number of times each function was selected, the total number of times we received the **SelectNew** and **GatherData** result and the total number of times the combined function was invoked have to be tracked.

In general the policy has the following data available to decide:

- Total run count of the combined function
- Total number of times each function was selected
- Total number of times of gathering new data (**GatherData** result)
- Total time spent on function execution (after **SelectNew** result)
- Total time spent on selection and storage overhead (after **SelectNew** result)
- Total time spent on processing the **SelectNew** result
- Total time spent on processing the **GatherData** result
- Number of times the last selected function was selected in a row (the *function streak*)

4.6.2 Implemented policies

As mentioned earlier, policies are designed to be reusable as building-blocks. The immutable policy itself is the only holder of the state and change of state can be performed only by transitioning to a different policy. Its functionality and very simple decision-making process can be visualized using a transition diagram, which shows the policy itself as a dark (red) square with solid border, all the other policies as a lighter squares with either solid or dashed border (determines if the policy is an argument or not), and arrows as state transitions. The transitions might be conditional, in such a case, the condition splits the arrow into two. The transition can produce a result which is shown next to the transition arrow. If a result is produced, the policy makes a decision and next policy will be evaluated during the following invocation. If a result isn't produced, the policy just delegates the decision to the next policy which is evaluated immediately. The chain of evaluation stops when first result is produced.

The transition conditions use policy parameters, which are passed to a policy in the constructor, and the current statistical data explained in section 4.6.1. It can also use static fields and methods that it has access to. The parameters of the

top-level (starting) policy are passed in by the user creating it, the parameters of the other policies in the chain are fixed when the policies are created, usually during the decision of the top-level policy, which typically builds the entire policy chain (that can be cyclic and lead back to it).

The transition diagrams and short descriptions of some policies will follow.

Building blocks

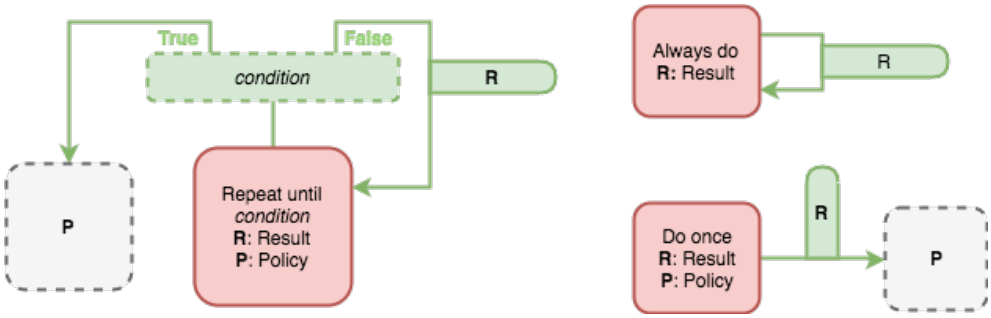

Figure 4.6: Transition diagram of the basic building-block policies.

Figure 4.6 shows the main policies that serve as a building-blocks. The most important is the **Repeat until condition** policy, parametrized by a *condition*, result R, which it keeps producing until the *condition* becomes true, and the policy P, which it makes transition to at that moment. The **Always do** policy represents an infinite loop producing the same result R every time, and the **Do once** policy produces the result R once and immediately makes transition to the policy P.

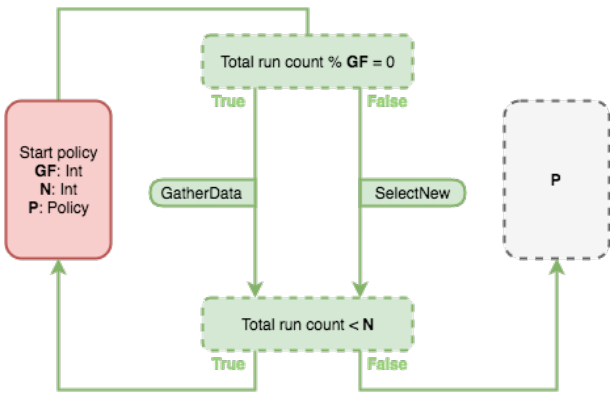

Figure 4.7: Transition diagram of the **Start policy**.

Figure 4.7 represents the **Start policy**, which accepts the gather frequency GF, number of invocation times N before moving to a next policy P. It is a little more complicated and specialized version of the **Repeat until** policy – it keeps producing the **SelectNew** result, and every GF-th invocation, it produces the **GatherData** result. It might be useful at the beginning, when it is necessary to gather data for all of the functions and perform some selections so that the following policies had statistical data to base their decision on.

Stop selecting when decided

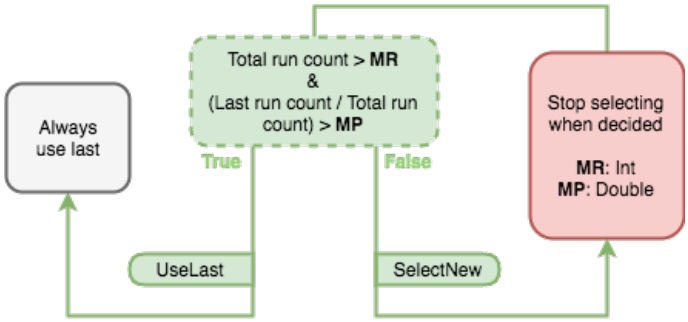

Figure 4.8: Transition diagram of the **Stop selecting when decided** policy.

In order to limit the overhead, it might be useful to completely stop selecting when we are sure that one of the functions gets selected a lot more often. The **Stop selecting when decided** policy shown in figure 4.8 keeps selecting new function until both the total number of invocations exceeds minimum run count *MR* and the ratio of the number of times that the last selected function was selected and the total number of invocations exceeds the minimum percentage *MP*. After that, it will keep using the last function forever.

The advantage of falling back to the infinite **Always use last** policy is its basically zero overhead – no condition is evaluated in the decision process. So this policy is useful for the fast functions where keeping low overhead for the long-term is critical. It cannot, however, undo a wrong decision, reflect a change in conditions, or anything else.

Pause selection after streak

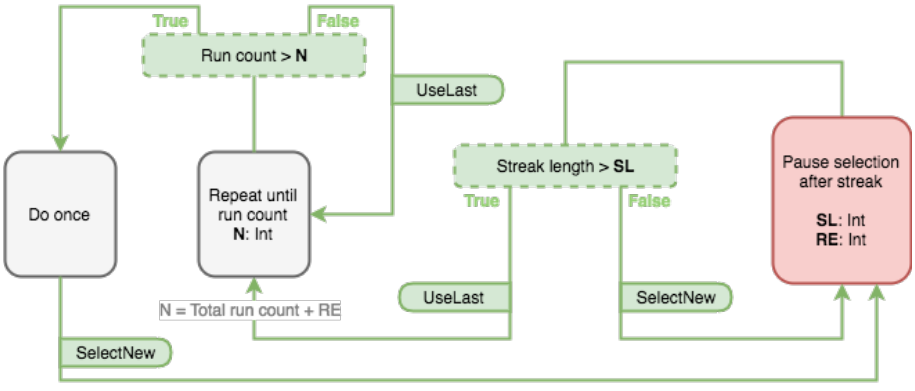

Figure 4.9: Transition diagram of the **Pause selection after streak** policy.

To lower the number of selections, an approach based on the streak lengths can be used as well. It is implemented by the **Pause selection after streak** policy in figure 4.9. Its functionality is defined by the *SL* (streak length) and *RE* (retry every) arguments. Whenever one function is selected *SL* times in a

row, the selection process stops and the **UseLast** result is being produced. Once every RE times, a new selection attempt is done, looping back to the original policy. If the same function is selected once again, the streak gets longer by one and the selection process stops again. If, however, a different function is selected this time, the streak becomes zero and we start selecting again.

The described behavior is the preferred way of limiting the selection overhead. By setting the streak length to a reasonable number (e.g. between 10 and 100, based on the data, situation, etc.), we can assure that if one function is significantly better than the others, it will be used without the need to select it. The retry that happens every once in a while makes sure that eventual wrong decision will be detected and will reset the system back to the beginning.

Limited overhead, gather time and selection time

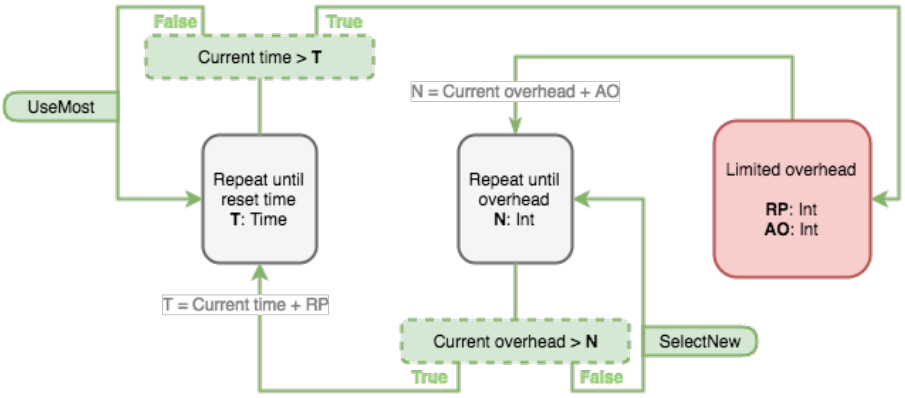

Figure 4.10: Transition diagram of the **Limited overhead** policy.

Policies can be used to limit the framework involvement per a real-time period, so that the selection overhead or the time spent on gathering data does not damage the system more than a specified limit. The **Limited overhead** policy in figure 4.10 limits only the selection overhead time to AO (allowed overhead) every RP (reset period). After depleting the allowed time, it cannot select anymore and keeps using the most selected function until the end of the period.

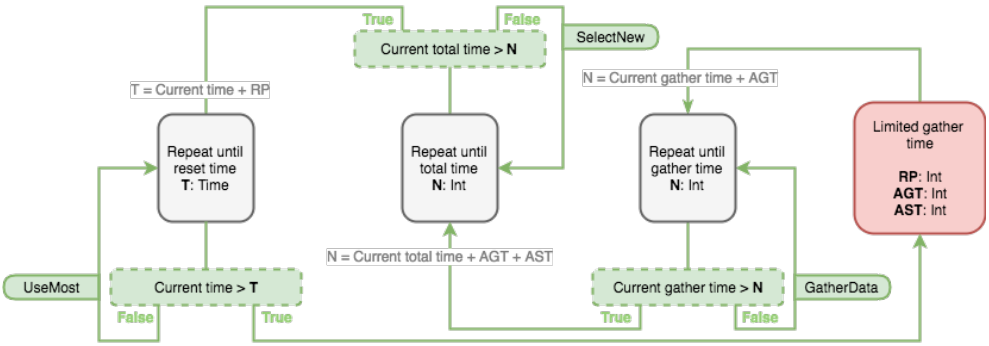

Figure 4.11: Transition diagram of the **Limited gather time** policy.

In exactly the same way, the time spent on executing the function for the result **GatherData** and the whole process of selection and execution for the

SelectNew result can be limited. The figure 4.11 demonstrates the **Limited gather time** policy, which keeps producing **GatherData** until total gather time reaches AGT (allowed gather time), after that, it continues with the **SelectNew** result until total selection time reaches AST (allowed selection time). A loop producing **UseMost** follows, for the rest of the real-time period specified by RP (reset period).

This policy gives us control over the time spent on selecting results in real-time. If we get a lot of requests at the same time, receive a large batch of data to process or for any reason need to momentarily increase the throughput of the system, the time or overhead limits get depleted very quickly and the **UseMost** fallback result means reasonably good performance with a minimum overhead.

4.6.3 Policy builder

As we could see from the policy examples, most of them were built only using the **Repeat until** or **Do once** blocks in a loop. This process of combining these generic blocks into sequences can be automatized and hidden behind a simple DSL (see 1.2) called *policy builder* that would allow the user to express his customized way to decide about the function run.

Figure 4.12 shows the syntax diagram of *policy builder*.

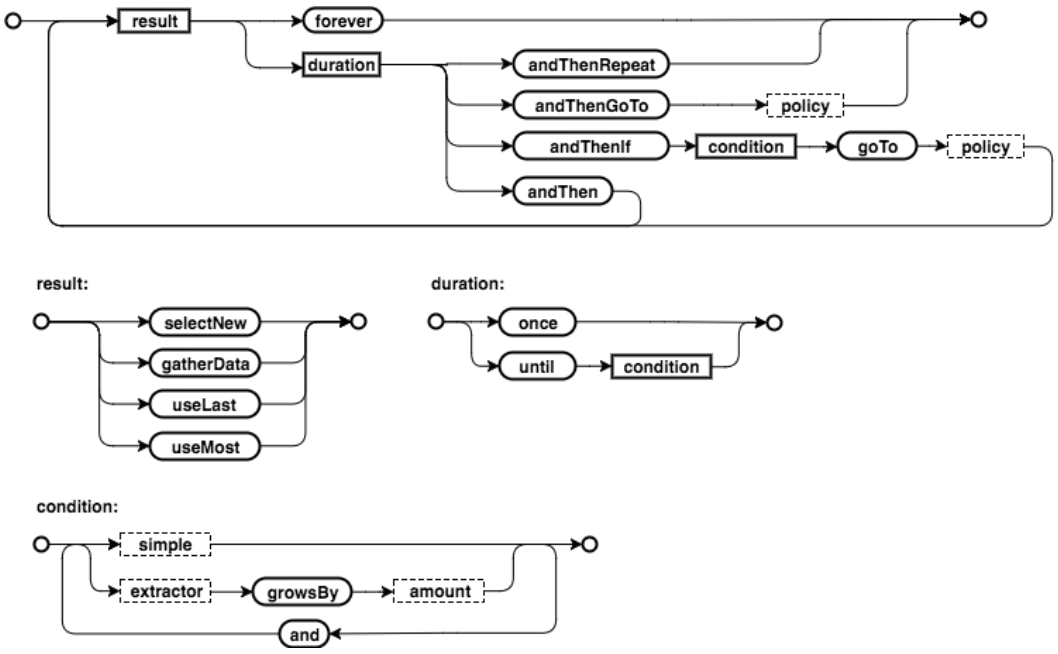

Figure 4.12: Syntax diagram of the *policy builder* DSL.

The policy built with *policy builder* has always one main loop. Its definition starts with the first result that it emits: *selectNew*, *gatherData*, *useMost* or *useLast*. A result is always followed by a duration specifier, and another result. When we specify the sequence of results and their durations, we can end it by either closing the main loop (using *andThenRepeat*), by transitioning to an external policy (using *andThenGoTo*, leads to a policy out of the loop), or by using *forever* as a duration for the last result, which limits the loop just this one result. In addition, it is possible to put a conditional transition to a policy out of the loop anywhere in between two results (using *andThenIf*).

The conditions can be either simple functions accepting statistical data and returning a boolean value, or a *growsBy* condition with a value extractor[7] and an amount. A simple condition is evaluated only once against current statistical data. In case of the *growsBy* condition, the extractor is executed once at the beginning of the policy loop and its result is stored, and the condition evaluation consist of executing it and comparing current result to the stored one.

An example of a simple policy described using the *policy builder* follows. It produces **GatherData** 50 times, then **SelectNew** 50 times, and then either loops forever on **UseLast**, if the streak is larger than 20, or repeats the whole process otherwise.

```
1 val conditional: Policy = (
2  gatherData until (totalRunCount growsBy 50)
3  andThen selectNew until (totalRunCount growsBy 100)
4  andThenIf ((stats: StatisticDataProvider) =>
      stats.getStreakLength >= 20) goTo (useLast forever)
5  andThenRepeat)
```

4.6.4 Policies and groups

One of the techniques for improving the selection process is grouping the history records using some input features, as described in 4.1.3. We will extend the grouping to function statistics and to the current policy. It means that the combined function needs to hold the policy and the statistical data for every group, and perform transitions and statistics updates separately.

The main advantage is the possibility to take faster decisions in some groups, using for example the *Pause selection after streak* policy. When a function is significantly faster than the others for some category of inputs, the streak will build up fast within the group and the invocation for future inputs from such a group will require no selection and will be very quick, yielding correct results.

4.6.5 Possible improvements

The policies were designed as a concept that is completely independent from the actual selection strategies and history analysis. One of the improvements that would require closer tying would be to allow the policies to choose the selection strategy, or to influence it somehow, by limiting it to a subset of data, etc.

Another improvement that might seem useful is to let the policies analyze the whole history data. This would, however, mean a lot of additional overhead time when evaluating the policy, which is not desirable. With a different approach, we could achieve the same thing by giving selection strategies a state that could involve their decision. The state would be managed by the combined function and it could even be the same state that is used by the policies.

[7] A function that extracts a value of some type from the statistical data. Default implementations are provided for the basic fields.

5. Framework implementation

The API, as described in chapter 3, and the decision making logic from chapter 4 were implemented into the ScalaAdaptive framework, a simple library that can be added to any Scala project and allows the programmer to use the adaptive functions in his application. It is included in attachment A, along with a brief user guide.

In this chapter we will go through the implementation of the framework, its basic architecture and the possibilities to extend it. We will also mention some decisions that had to be made during the implementation.

5.1 Goals of implementation

The main concerns upon implementing the framework were the following:

- To separate the API from the selection logic
- To keep the framework strongly typed and use static type checking even in the internal parts of the code
- To make the framework as extensible as possible
- To keep the framework overhead as low as possible
- To keep the code well-organized, to minimize code duplication

Development approach

The main development approach was based on using the Scala language features to make the code simpler, easier to maintain and less error-prone. We tried to benefit from the functional features whenever possible, while still keeping the high-level design object oriented. It lead to having the data and functionality separated in different classes. Inheritance is used only for implementing traits (except for some special cases, like the API function objects). All of the functionality-providing classes are accessed via corresponding traits, and thus communicating only through a simple and limited interface, which allows replacing them very easily. These trait-based building blocks are put together using composition and delegation patterns.

Error handling

The framework is designed so that the number of exceptions handled in the code was minimized. The ScalaAdaptive itself does not raise almost any exceptions in case of errors[1], and catches most of the exceptions from the libraries within to replace them with a *None* return value.

This approach is known from the functional programming and takes advantage of monadic operations over the *Option monad*[2]. The return values can be mapped

[1] The only exception being custom identifier validation in the function combination.

[2] In Haskell and other languages known as *Maybe monad*.

over using the *bind* operator, which allows smooth function chaining and the error propagation through the chain. More details can be found in the talk [22].

5.2 Architecture overview

This section will provide brief overview of the entire framework architecture and related decisions. More detailed description at the level of individual classes can be found in the generated documentation (included in attachment A).

5.2.1 API architecture

As briefly described in section 3.1.4, the public API of the framework is represented by a set of traits `AdaptiveFunction0`, ..., `AdaptiveFunction5`, that expose all the user accessible methods. Because only some of the methods depend on the number of input arguments and are therefore specific for each of these types, the rest of the interface was extracted into a common trait supertype `AdaptiveFunctionCommon`, which is parametrized by the adaptive function type that extends it.

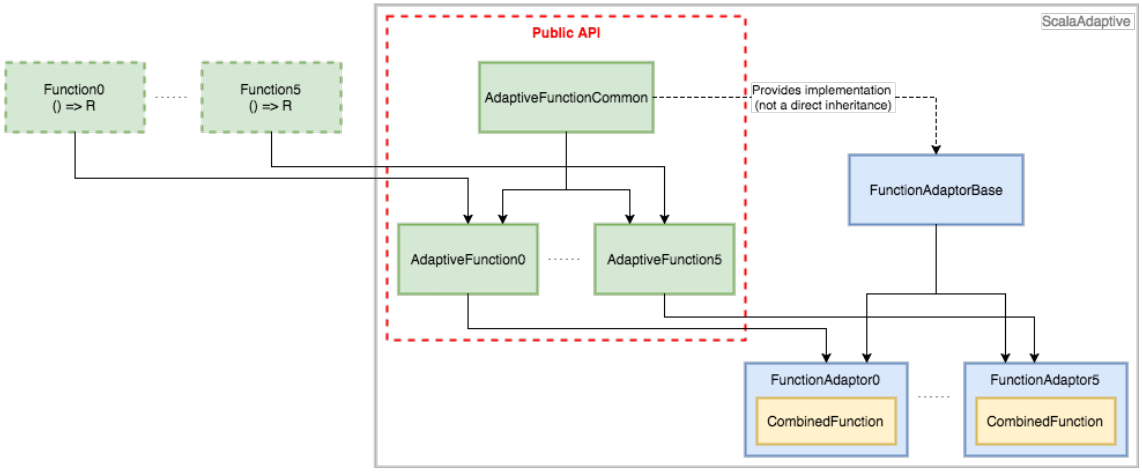

Figure 5.1: Diagram showing the inheritance chain of AdaptiveFunctionN and related classes.

The `AdaptiveFunctionN` traits are implemented by the `FunctionAdaptorN` classes that are directly inaccessible to the user. This allows us to modify or extend their internal API without changing the public API, thus maintaining backwards compatibility. The methods that can be implemented independently on the number of input arguments are extracted into an abstract class `FunctionAdaptorBase` that the other adaptors inherit from.

It would be impractical to manipulate with N different classes that represent the combined functions in the internal parts of the framework. Any Scala function with N arguments can be transformed into a function with one argument by simply using a *tuple argument*:

```
1 val function = (arg1: T1, ..., argN: TN) => ???
2 val tupleFunction = (tupleArg: (T1, ..., TN)) =>
3       function(tupleArg._1, ..., tupleArg._N)
```

We will take advantage of this and, in order to minimize the code duplication, hold all functions in this form in a `CombinedFunction` class parametrized only by the tuple argument type and the return type. The `FunctionAdaptorN` classes work as simple adaptors between the internal, tuple-based function interfaces, and the external interfaces of `AdaptiveFunctionN` and `FunctionN`. The whole inner part of our framework works only with `CombinedFunction` types. The figure 5.1 shows described inheritance and composition scheme.

The user of the framework will generate instances of mentioned function types using implicit conversion methods defined on the `Implicits` singleton, a part of the API that is described in 3.1.4. The actual conversions where the `FunctionAdaptorN` and the `CombinedFunction` instances get created will be wrapped inside another singleton object, `Conversions`, and the delegating calls will be done using a macro due to reasons explained later.

5.2.2 Internal architecture

The simplified internal architecture can be seen in the figure 5.2. It shows the key components of the core chain that is executed when a combined function is invoked.

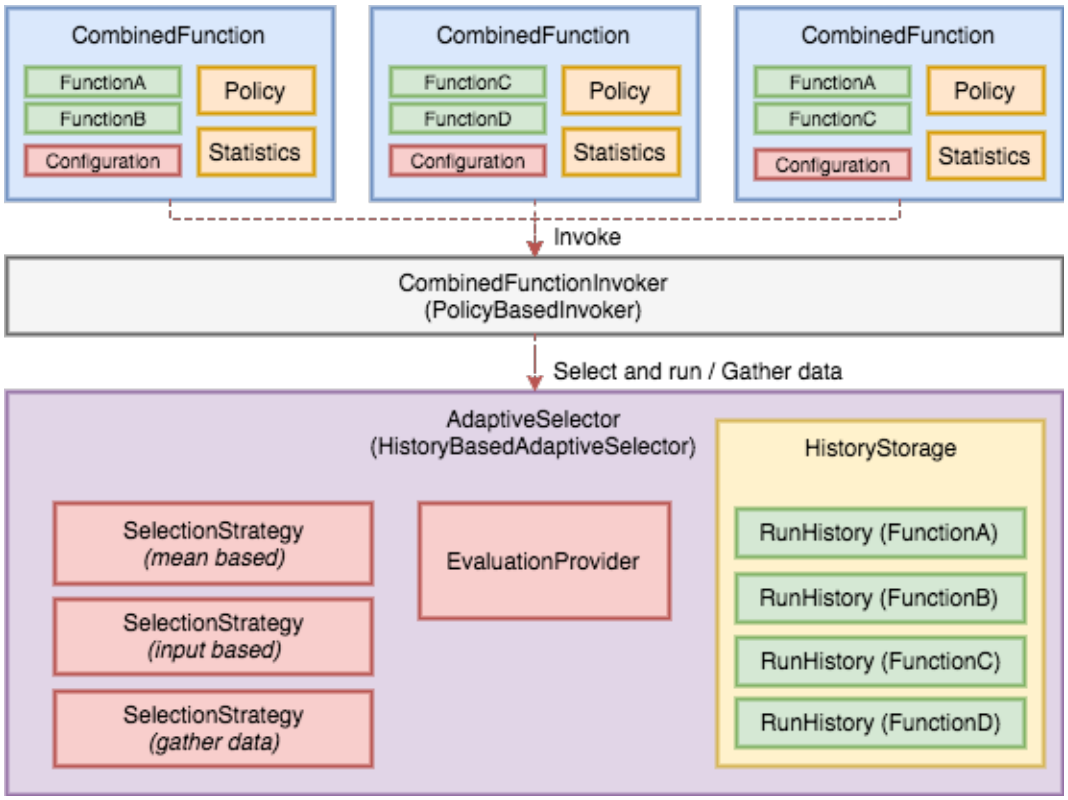

Figure 5.2: Diagram showing the internal architecture of the ScalaAdaptive framework.

As we can see, there are two principal logic units participating in the process:

1. **`CombinedFunctionInvoker`**
   Supposed to make a quick decision based on the policy of the combined function. It should either directly invoke one of the functions, or delegate

the call to the `AdaptiveSelector`, and update the statistics of the combined function afterwards. It works only with the data directly stored in the `CombinedFunction` instance, it does not have any state or shared storage.

2. **`AdaptiveSelector`**
   The key component of the framework, it receives a set of functions and an input, and it should select one of them. That function should be executed, its run evaluated and the the return value passed back. The default implementation works with an abstract history storage (`HistoryStorage`), three selection strategies (`SelectionStrategy`) and an evaluation mechanism (`EvaluationProvider`).

5.3 Implementation options

A few questions regarding the implementation arise from the basic architecture showed in section 5.2. We will provide a brief analysis before explaining the implementation details any further.

5.3.1 History storage location

`HistoryStorage` is supposed to gather and store historical run data for all the functions that are invoked through a combined function. The question is, where to save the data and how accessible to make it. We propose three variants with different use cases that were all included in the framework.

Local

First option is to make the storage part of the `CombinedFunction` itself. In this case, the history data would be unique for every instance of the class, i.e. for every `AdaptiveFunctionN` created by the user. This has two consequences:

- A function will have separate histories for every use in a combined function.
- Every new instance of a combined function will have an entirely new history for all the functions.

The second consequence might lead to some unexpected behavior – for example, if we create a combined function inside of a method call as described in section 3.2.1, it will have a clear history every time and will be useless:

```
1 def processData(data: List[Int]): Int =
2   (impl1 _ or impl2 withStorage Storage.Local)(data)
```

However, some use cases for this configuration exist, e.g. if we have a class that holds immutable data and we keep executing some computations on the data, then a combined function defined as a field on this class with local storage will adapt to that specific instance and will have the best possible results.

Global

Another option is to store all the measured data globally, in a static area of the memory accessible from all contexts. This will allow to collect run data for functions faster and share them across all the combined functions, which usually leads to more informed and therefore precise decisions. This is usually the preferred approach for most use cases. A unique identifier has to be used to match the function history record with the actual function in the `CombinedFunction` class. The selection of the identifier will be discussed later.

Persistent

The biggest problem of discussed storage locations is that the run history data are available only during a single run of the application. Whenever the application restarts, it will have to collect the data again. This might not be a problem for some long running services or daemons, but it is not very convenient for some tools or client applications with shorter lifecycle.

A solution to the problem is adding a persistent storage (e.g. the file system) to the global storage and persisting the data at certain points. That can be done in three different ways:

1. Immediately persist each run history record
2. Persist all of the records when the application terminates
3. Buffer the history records and persist them in batches

The first option seems like the best one, but it would lead to a series of I/O operations upon every invocation of a combined function. The added overhead of this solution could be dramatic. The second option, on the other hand, has no runtime overhead at all, but it requires a possibility to detect the application termination from our framework. JVM does not guarantee finalizers being called and relying on notifications from the user would complicate the API and be problematic in general.

There is a mechanism in JVM called *shutdown hooks* that allow the user to perform a custom action on shutdown (for more details see [23]). It will, however, not be called if the application crashes because of an unhandled exception. Additionally, it might be unexpected from a library to perform actions on shutdown and the delay caused by persisting a lot of run data on a slow HDD[3] (or even a network drive) could lead to the application being terminated immediately in some automatized environments.

Therefore, the third option was selected, as it leads to the runs being regularly saved and the I/O overhead is lower.

The persistent storage, just like the global one, relies on the unique function identifier as well. There are additional problems connected to persisting the run history, namely:

- Running multiple instances of the application at once (collisions on the persisted data file)

[3]Hard Disk Drive

- Changes in the application code (outdated run data, changed identifiers, etc.)

- Having to deal with larger amounts of history data

5.3.2 Function identifiers

As described in section 5.3.1, we need some sort of identifier of the function to be able to store the run history data in a global or persistent storage. It does not make sense to use the references to the function objects, as new closure instances are created every time a lambda expression (or an eta-expanded method) is assigned, which would lead to having new history for each new instance, a behavior equivalent to local storage discussed in 5.3.1.

This leaves us with three basic identifier options.

Type name

As explained in 1.1.1, functions in Scala are instances that extend the `FunctionN` trait. The default implementations are anonymous closure classes that are compiled from lambda expressions. Two different functions originate in two different lambda expressions and thus have two different type names, the ones that were generated and assigned by the compiler. The fully qualified type name can be used as an identifier of a function

Using the type names as unique identifiers is safe and straightforward. A small disadvantage is that they are not very readable, the compiler uses the name of the type that contained the lambda expression followed by a sequential number.

There is also a subtle danger connected – if we try to combine two different functions that originated from the same lambda expression (the difference might be determined by the closure arguments), they will have the same identifier, thus sharing the same history, which is usually not desirable.

The next problem is that in case of persisting the run history, the closure classes might get renamed automatically upon recompiling. The compiler usually assigns the closure names sequentially, so this could happen by just inserting another lambda expression into the enclosing class code before the current one. Run history data might even get mixed up as the newly added lambda expression could have the former name of the original closure (by taking its position in the sequence).

Last but not least, if the `FunctionN` trait has a custom implementation, the type name identification might lead to an undefined behavior.

Method name

The expected most common usage pattern of the framework is the one where the functions used with the `or` method are *eta-expanded* methods (see 3.2.1). There is a problem connected with methods when using the type name identifiers – every time a method gets eta-expanded, a new lambda expression with new type name is generated. In the following case, the `method1` would not have the same identifier in the two combined functions:

```
1 val fun1 = method1 _ or method2
2 val fun2 = method1 _ or method3
```

It would be handy to use the method name as an identifier in such a case, which would lead to better readability and allow the same method used in different combined functions to have just one run history.

The problem is that at runtime, the implicit type conversion method or the `or` method will always get the already eta-expanded function object with a compiled `apply` method holding the actual method call inside. The names therefore have to be extracted at compile time, using the def macros (see 1.5). At the moment of the implicit conversion from `FunctionN` to `AdaptiveFunctionN` (as described in 3.1.4), the AST of the `FunctionN` expression can be examined – if it's a lambda expression generated from eta-expansion, the method name can be extracted. The exact process how to do so will be presented in a separate section.

Custom identifier

In order to handle specific cases where type name and method name identifiers do not distinguish correctly between different functions, there is also a possibility of choosing a custom, arbitrary identifier. This has to be triggered specifically by the user in the API and should be used in the cases where:

1. User knows that the automatically assigned identifiers will not be sufficient
2. User wants to replace default type identifier with custom identifier because of readability

5.4 Extracting method name from eta-expansion AST

In section 3.1.4, we introduced an implicit method that will perform the conversion from `FunctionN` to `AdaptiveFunctionN`. In order to extract the method name from potential uses of eta-expanded methods in this conversion, we will have to implement it in the following way:

1. Replace the implicit conversion method by a def macro (see 1.5) and extract the conversion logic into a different, non-implicit method that allows explicitly setting the function identifier (in our case, the `Conversions` singleton object methods `toAdaptor`)
2. In the implicit macro, analyze the AST and try to locate the eta-expansion method call
3. If the method call is successfully found:
   (a) Generate the identifier expression as a `getTypeName` call on the target of the method call, concatenated with the method name
   (b) Generate the conversion code with explicitly specified identifier expression

4. Otherwise generate the conversion code with implicit identifier

The conversion is done using the `toAdaptor()` method with two overloads:

- Accepting only the function – implicit identifier is used (type name of the closure)
- Accepting the function and an identifier – the identifier provided is used

5.4.1 Eta-expansion AST format

First step in the macro implementation has to be parsing the input AST and detecting patterns that are generated from eta-expansions by the compiler. In this section, we will present what was discovered about the eta-expansion presence in the ASP using the `printAst()` macro mentioned in 1.5.1. Pieces of the AST structure will be shown, [10] can be used as a reference.

The list of facts follows:

- The result of eta-expansion is a lambda expression (referred to as function literal in the AST).

  ```
  Function(...)
  ```

- The lambda expression is always wrapped in a block, being its return value.

  ```
  Block(
    List(...),
    Function(...))
  ```

- If the target of the invocation is either a constant or *this*, it is captured in the lambda expression closure (i.e., the constant or *this* is referenced directly from the function body).
- If the target of the invocation is a variable or a result of a more complicated expression, it is extracted to the enclosing block, its result is stored in a variable local to the block and then captured in the lambda expression closure. The following example shows how the target, originally an expression `this.getInstance()` on a `Class` class, was extracted into the block body.

  ```
  Block(
    List(
      ValDef(
        Modifiers(SYNTHETIC),
        TermName("eta$0$1"),
        TypeTree(),
        Apply(
          Select(
            This(
              TypeName("Class")),
            TermName("getInstance")),
          List()))),
    Function(...))
  ```

- The function node contains argument definition and the expression itself, which is a single application (method call).

```
Function(
 List(
  ValDef(
   Modifiers(PARAM | SYNTHETIC),
   TermName("arg"),
   TypeTree(),
   EmptyTree)),
 Apply(...))
```

This has a few consequences for our case. First, we will look for functions with a body consisting from one method call. Secondly, we need to generate our conversion code (along with the method retrieval) into the block return value, because we need to be able to access the actual invocation targets that might exist only in the block (as the original expression could have been extracted and replaced by a local variable).

5.4.2 Retrieving the target from the method call

Now we need to go through the method call subtree and locate the invocation target and the method name. If the method does not accept any type arguments, the structure is quite simple:

```
Apply(
 Select(
  ...invocation target expression...,
  TermName("methodName")),
 List(...function arguments...))
```

Where the *invocation target expression* can have multiple forms based on the original expression, and can depend on the enclosing block variables. It is not, however, important for us, as we can work with the expression as whole and duplicate it. The function arguments are not needed either.

If the method is generic, the type arguments need to be applied in order to convert it to a function (which cannot be generic). In this case, the tree gets a little more complicated:

```
Apply(
 TypeApply(
  Select(
   ...invocation target expression...,
   TermName("genericMethod")),
  List(...type arguments...))),
 List(...function arguments...))
```

The method call is wrapped in a TypeApply node before being invoked using the Apply node. The TypeApply node can be ignored in our case.

And the most complicated situation we are going to analyze is when the method has some implicit arguments as well:

```
Apply(
  Apply(
    TypeApply(
      Select(
        ...invocation target expression...,
        TermName("genericMethodImplicit")),
      List(...type arguments...)),
    List(...function arguments...)),
  List(...implicit arguments...))
```

One more Apply node is added to the topmost level – the implicit arguments are applied after applying the actual function arguments. Their definition contains another nested block and lambda expression, but again, it is not important for our case. We just need to extract the invocation target and the method name, which can be both found in the Select node.

Note that there are situations where the eta-expansion might lead to even more complicated trees, for example when used on methods with multiple argument lists. These are, however, cases which we are not going to support with the extraction.

5.4.3 Generating the conversion

Supposing we have the invocation target expression and the method name, we need to create the identifier string expression that will be used in the manual `toAdaptor` invocation.

In order to extract the fully qualified name of the method call target, we need to generate the following expression:

```
1 invocationTarget.getClass.getTypeName + ".methodName"
```

The AST representing this expression has to be wrapped in a construction application of `MethodNameIdentifier` class (it is a case class with automatically generated `apply()` method) to get the resulting identifier.

Now, the entire function literal from the original AST along with this identifier expression have to be wrapped in a `toAdaptor` call and then be set as the return value of the original AST block.

The original tree in a simplified form can be seen in figure 5.3. The tree after the conversion is shown in figure 5.4.

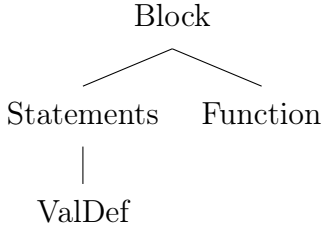

Figure 5.3: The original simplified eta-expansion AST.

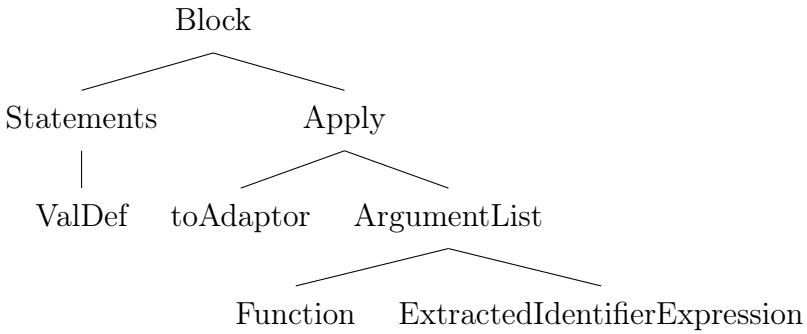

Figure 5.4: The simplfied eta-expansion AST after manually adding toAdaptor call.

5.4.4 Extracting method overloads

The approach that was described so far has one small issue – it does not recognize function overloads, so all the overloads share the same identifier.

It would not be difficult to extract the actual number of arguments that the method is being invoked with and include it in the identifier. As for extracting their types, the situation gets a little complicated – the function literal is generated without argument type specification, the TypeTree is empty:

```
ValDef(
  Modifiers(PARAM | SYNTHETIC),
  TermName("i"),
  TypeTree(),
  EmptyTree)
```

This is a valid AST in Scala and can be generated in a lot of situations, for example:

```
1 val function: (Int) => Int = { i => math.abs(i) + 1 }
```

In this case, the lambda expression does not have the type of its arguments specified either, because the compiler will infer it from the context in which the expression is used, in this case, from the type specifier of the variable it is being assigned to.

The compiler is able to infer the data type from the usage of the argument as well[4]:

```
1 def method(i: Int): Int = ???
2 val function = { i => method(i) }
```

Note that the eta-expansion is guided by the same rules, so whenever expanding a method without overloads, the compiler infers the types by itself:

```
1 def method(i: Int): Int = ???
2 val function = method
```

Upon expansion of a method with overloads, the resulting function type has to be provided either explicitly, or as an expected type in an expression (e.g. inside

[4]The code snippet is correctly compiled by the Scala compiler, although some IDEs (namely IntelliJ IDEA 2016.3.4) are not able to infer the type and mark the code as incorrect.

a method argument). The inference flow goes in the other direction in such a case:

```
1 def method(i: Int): Int = ???
2 def method(s: String): Int = ???
3 val function: (String) => Int = method
4 def useFunction(fun: (Int) => Int): Unit = ???
5 useFunction(method)
```

As a consequence, at the time of syntax analysis, the types are not inferred yet and there is no way for us to find out which method overload is being called. The method overloads therefore have to share the same identifier. If it leads to a problem, a type name or a custom identifier can be used in such a case.

5.4.5 The conversion demonstration

As a practical demonstration of the process described in this section, we are going to show the compile-time conversions that are done on a method identifier that is being eta-expanded and converted into a `AdaptiveFunctionN`. Let's consider the following code:

```
1 val combined = target.method _ or function
```

We will follow the conversions that the compiler and our macro perform on the `target.method _` expression[5]:

1. The explicit eta-expansion is performed:

```
1 { () => target.method() }
```

2. The implicit conversion is applied:

```
1 Implicits.toAdaptiveFunction0({ () => target.method() })
```

3. The macro `toAdaptiveFunction0` is executed and expanded:

```
1 Conversions.toAdaptor({ () => target.method() },
    MethodNameIdentifier(target.getClass.getName +
    ".method"))
```

The final state shows the code whose equivalent will be present in our compiled program and will be executed.

5.4.6 Macros in the combining method

In the section 3.2.1, we explained why the `or` method has to accept an argument of `FunctionN` type instead of `AdaptiveFunctionN`. This does not seem to cause any trouble as we can manually convert `FunctionN` into the `FunctionAdaptorN` inside the method body. With the macro expansion, however, we run into problems. Image we do it the following way:

[5]The actual changes are performed on the AST level, we will show them in corresponding Scala code for simplicity.

```
1 def or(fun: (T1) => R) =
    orAdaptiveFunction(Implicits.toAdaptiveFunction1(fun))
```

The `toAdaptiveFunction1` method is a macro – it gets executed at compile time, and it receives the argument AST as its input. In this case, an AST that consists of the `fun` identifier. We have no way to find out, where this function will be called from and with which arguments, so in order to extract the method name, we need to convert the entire `or` function to a macro which will generate the conversion code in the same manner as described in section 5.4.3, and then wrap it in the internal `orAdaptiveFunction` code.

This does not require any extra AST manipulations, only one method call generation. It does, however, mean a complication for us, as the macros cannot be called virtually. The macro definition therefore has to be a part of the `AdaptiveFunctionN` API trait, breaking somehow the encapsulation of the implementation. Fortunately, it does not perform any actions by itself, only replaces the `or` call with a conversion and an `orAdaptiveFunction` call, which is, again, a virtual method defined on the trait without implementation and therefore encapsulated.

5.4.7 Extraction evaluation

In order to evaluate the method name extracting solution presented throughout this section, we converted some expressions with function values to corresponding `AdaptiveFunctionN` types using the macro approach. Table 5.1 shows the results. Note that the package prefixes were omitted from the method names for simplicity, the actual identifiers contain fully qualified names. The conversion was performed in the scope of a `NameTest` singleton object.

Expression	**Assigned method name**
`method _`	NameTest$.method
`genericMethod[Int] _`	NameTest$.genericMethod
`Singleton.quickSort _`	NameTest$Singleton$.quickSort
`variableSorter.quickSort _`	Sorter.quickSort
`getSorter.quickSort _`	Sorter.quickSort
`new Sorter().quickSort _`	Sorter.quickSort
`new GenericClass[Int]().foo _`	GenericClass.foo
`(i: Int) => method(i)`	N/A (Closure name used)
`(i: Int) => method(i + 1)`	N/A (Closure name used)
`function`	N/A (Closure name used)

Table 5.1: The method names extracted from expressions during their conversion to combined functions.

As we can see, the method name extraction succeeded in all the *eta-expansion* cases and did not get confused even by a direct method call within a lambda expression.

5.5 Module implementation

All the modules mentioned in 5.2.2 are abstract traits with replaceable implementations that are put together using composition. The API classes have to be able to access the composed and active implementations at the moment of invocation. For this reason, a singleton Scala object `AdaptiveInternal` was introduced and holds the following composed modules:

- `AdaptiveSelector` with global storage
- `AdaptiveSelector` with persistent storage
- `CombinedFunctionInvoker`

We will briefly introduce the implementations of these modules and the parts that they are composed from.

CombinedFunction

The `CombinedFunction` is the main data class representing a function with multiple implementations in ScalaAdaptive. It is basically a data holder class, the important functionality was separated to be easily replaceable. It gets created and modified when the user manipulates with the `AdaptiveFunctionN` API type. Part of its attributes is immutable, representing the configuration and default setup of the function, and part of its attributes is mutable, holding its state in the invocation process.

The immutable data are the following:

- Set of functions that it is combined from (wrapped into the tuple form, see 5.2.1)
  - Each function holds up to two identifiers – one based on the closure type name, the other being either a method name or a custom name, depending on the creation process (see 5.3.2)
- *Descriptor function* (wrapped into the tuple form), see 4.4
- *Group selector* function (wrapped into the tuple form), see 4.1.3
- Function configuration defined by the user, will be described later in more detail

And the following state representing data:

- Function statistics (separately for each group), see 4.6.1
- Current policy (separately for each group), see 4.6
- Analytics data – full list of all the selections and their results

In case of the local storage setup, the instance also has to contain its own `AdaptiveSelector` that holds the local `HistoryStorage`.

CombinedFunctionInvoker

A module for invoking the `CombinedFunction` instances. Its basic implementation does the following:

1. Evaluate current policy using the statistics of the combined function
2. According to the result either invoke directly the function which is accessible through the statistics (the last one and the most selected one) in case of `UseLast` or `UseMost` results, or pass the decision onto the `AdaptiveSelector` in case of `SelectNew` or `GatherData` results
3. Update the function statistics

The `AdaptiveSelector` is accessed either through the `CombinedFunction` itself for the local storage setup, or using the singleton object `AdaptiveInternal` in case of global or persistent setup.

AdaptiveSelector

A module which is supposed to select one of given options to run, and to evaluate the run. It is parametrized with `TMeasurement` type, which represents the data measured from the function run. By default, a `Long` type for run time measured is used. It supports two basic operations – *selectAndRun* and *gatherData*, which correspond to the policy results. Both of these operations have also an option with delayed measurement (see 3.2.6).

The implementation `HistoryBasedAdaptiveSelector` contains the general chain described in 4.2. It uses a `HistoryStorage` to store and retrieve the run data of individual functions. In addition, it holds three `SelectionStrategy` instances, one for gathering new data, one for the input based selection, and one for the mean based selection. One of these three instances is used to select the function to run, which is then executed using an `EvaluationProvider`. The measurement retrieved is added to the history and the result is returned, along with a performance benchmark including execution and overhead times (independent on the actual measurement, always based on wall-clock times) for the function statistics update. Figure 5.5 shows the entire process.

In case of the operations with delayed measurement, the function run is not evaluated, an `InvocationToken` is generated instead. The token holds a callback to the `AdaptiveSelector` and whenever used to invoke some function, it will use the `EvaluationProvider` to evaluate the run and store the data to the corresponding history record.

Note that the `SelectionStrategy` for gathering new data should select a function for which data are needed the most at the moment instead of optimizing the performance.

HistoryStorage

A type with a map-like interface based on the concepts from section 4.2.2 – it is supposed to hold the historical measurement vectors represented by `RunHistory` instances for each combination of function and input group. A key composed of a function identifier and a group identifier is used to access the histories.

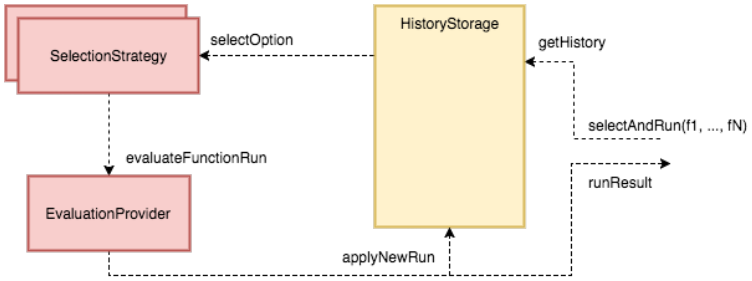

Figure 5.5: Diagram showing the HistoryBasedAdaptiveSelector execution path.

There are two implementations in the framework:

- `MapHistoryStorage` – stores the histories in memory
- `PersistentHistoryStorage` – a wrapping storage that delegates the calls to an internal `HistoryStorage`, and, in addition, it serializes every new run using a `HistorySerializer`; if asked for a history that is not present in the underlying storage, it tries to deserialize it using the same type

The `HistorySerializer` has two basic implementations, one for direct serialization and one for buffered serialization. The data are stored into a file, one per function (the format, root directory and file name pattern are fully customizable). There is no list of these history containing files anywhere – when trying to deserialize a function, a file with corresponding name is checked for existence. This means that the individual function histories are deserialized one by one, at the moment of the first invocation of a combined function that contains them, which might cause a delay in the first execution.

RunHistory

`RunHistory` represents a vector of historical measurement results and other details about a function run. The interface is designed to support immutable solution – the appending methods always return an instance of the same type, which might or might not be the same object depending on the implementation. The interface provides some basic operations on the history, allows iteration over the data and appending new data (always at the end).

Apart from that, it also has some methods to directly provide precomputed data for some of the selection strategies in order to support caching. These supporting methods can always be computed using the data available, and if the implementation of `RunHistory` does not support the caching or precomputation, there are *mixin* traits (see section 1.4) with default implementations available.

The basic implementations of `RunHistory` are the following:

- `FullRunHistory` – stores all runs in an `ArrayBuffer`, is mutable
- `ImmutableFullRunHistory` – stores all runs in an immutable `List`, is a little slower in general

These instances can be wrapped in the following decorating wrapper classes:

- `LimitedRunHistory` – limits the maximum number of stored items, whenever it reaches maximum, it throws the older half of all the records away
- `CachedStatisticsRunHistory` – stores the statistical data about the run items, speeds up the t-test strategy (see 4.3.2)
- `CachedGroupedRunHistory` – stores average evaluation data for each *run selector*, speeds up the local regression strategy (see 4.4.3)
- `CachedRegressionRunHistory` – stores the linear regression model built upon all the run items, speeds up the linear regression strategy (see 4.4.1)

These wrappers can be chained in order to reach the desired behavior. They usually only decorate one method and the rest is directly delegated to the inner instance.

The `LimitedRunHistory` wrapper should always be used, as it can prevent unexpected out-of-memory exceptions in the program run, as the base run history implementations have no size limits.

SelectionStrategy

Very simple trait that represents one of the selection strategies, as described in detail in chapter 4. It receives a set of `RunHistory` instances for various functions and groups along with an input descriptor, and should select and return one of the functions.

The implementations should be stateless and should work only with the set of `RunHistory` records that it receives (in order not to mix together decisions about two separate history storage locations). The strategy implementations often have a customizable fallback strategy that is used in cases where the current strategy is not able to make a decision, as described in section 4.1.5.

The strategies are parametrized by the `TMeasurement` type of the function run evaluation. All of the implemented strategies work with function run time represented either directly by `Long` or by a type viewable as `Numeric`.

The mean based strategies consist of the following classes:

- `TTestSelectionStrategy` – implements the t-test strategy described in section 4.3.2, uses [24] to compute the p-value.
- `UTestSelectionStrategy` – implements the u-test strategy described in section 4.3.3, uses [24] to compute the test statistics.

The implemented input based strategies are:

- `RegressionSelectionStrategy` – implements the linear regression strategy as described in section 4.4.1, uses [24] to compute the regression
- `LoessInterpolationSelectionStrategy` – implements the local regression strategy from section 4.4.3, uses [24] for the interpolation

In addition, some complementary supporting strategies were added:

- `WindowBoundSelectionStrategy` – a wrapper strategy that limits the history records to a flexible window before letting the internal strategy decide, as described in sections 4.4.2 and 4.4.4
- `LowRunAwareSelectionStrategy` – if any of the functions has less than a specified number of history records, uses one strategy, otherwise uses a different strategy
- `LeastDataSelectionStrategy` – uses the function with least historical runs

The common pattern is to select one of the input based or mean based strategies and to set the `LeastDataSelectionStrategy` as its fallback strategy. Optionally, the strategy can be put into the window-bound type. Then, it should be wrapped inside the `LowRunAwareSelectionStrategy` to postpone the decision process until some data are gathered.

The `LeastDataSelectionStrategy` is also recommended to be used as the strategy for the *gather data* operation of `AdaptiveSelector`.

EvaluationProvider

With the goal of potentially supporting larger variety of possible function evaluation data, the selected function is always executed using an implementation of `EvaluationProvider`, which is supposed to run the function and to evaluate it, and return both the result of the function and the evaluation data in the form of `TMeasurement`. The default implementation simply measures the wall-clock time of the function run.

5.6 User interaction and customization

The basic behavior of combined functions can be changed in two ways:

- Each combined function can have the most basic features configured individually
- The whole framework can be re-configured and customized

5.6.1 Combined function setup

As the simple API designed in chapter 3 does not allow directly setting the function selection behavior, an extension had to be added. In compliance with the DSL-like `or` chaining, the configuration was made possible via special methods on the `AdaptiveFunctionN` type that allow chaining as well[6] and are also meant to be used like operators. The combined function definition including the configuration can be very fluent and natural:

```
1 val sort = standardSort _ or selectionSort by (_.size)
    selectUsing Selection.InputBased storeUsing Storage.Global
```

We have a look at what can be set using these methods.

[6] In the *builder*-style pattern, by returning the same type.

Input descriptor and group selector

The most important part of the configuration API are the `by` and `groupBy` methods. The first allow the user to specify the *descriptor function*, i.e., function describing how to extract an *input descriptor* from the input (see 4.1.2). A function of type `(T1, ..., TN) => Long` is used for that, where `T1, ..., TN` are the type arguments of the `AdaptiveFunctionN`.

The second method expects a *group selector*, which assigns the input to a certain group (see 4.1.3). It should receive a function of type `(T1, ..., TN) => Group`.

Function configuration

The invocation and selection process can be configured even more using an instance of a `FunctionConfiguration` class that each combined function holds and that gets passed to the `CombinedFunctionInvoker`.

The choices are based mostly in selecting which one of the module implementations should be used with given function. In the `AdaptiveInternal` singleton, we hold two implementations of `AdaptiveSelector`, one with persistent history storage, the other without. Each of these two implementations then includes an input based and a mean based implementation of `SelectionStrategy`

The API methods that change the `FunctionConfiguration` and their possible values are the following:

- `storeUsing` – represents the history storage options as described in section 5.3.1
  - `Global` – uses the shared `AdaptiveSelector` without persistent storage
  - `Persistent` – uses the shared `AdaptiveSelector` with persistent storage
  - `Local` – creates a new instance of `AdaptiveSelector` locally in the function object
- `selectUsing`
  - `InputBased` – uses the input based strategy in `AdaptiveSelector`
  - `MeanBased` – uses the mean based strategy in `AdaptiveSelector`
  - If none is specified, uses the input based strategy if the *descriptor function* is set and the mean based strategy otherwise
- `limitedTo` – allows to specify the maximum age of the history records to be used in the selection process, as described in 4.1.4
- `asClosures` – represents the history storage identifier choice as described in section 5.3.2
  - *true* – always uses closure type names as the function identifiers
  - *false* – uses either method names or custom names as the function identifiers if available

- `withPolicy` – allows to specify the starting policy for the function, as described in 4.6

The user is expected to specify this configuration with most of the combined functions. The default values are determined by the framework configuration, except for the selection strategy choice, which is defaulted to input based if the *descriptor function* is provided for the combined function and to mean based otherwise.

5.6.2 Logging, analytics and control access

There are two basic mechanisms for the user of the framework to observe its functionality – logging and analytics.

Logging

Logging is performed by multiple modules that are involved in the selection process. An instance of `Logger` trait held in the `AdaptiveInternals` and passed around to other modules in their constructors is used for that purpose. The provided implementations are `ConsoleLogger`, `FileLogger` and `EmptyLogger`, where the last one does not actually save the logs anywhere, and is used by default to limit the overhead.

Replacing `EmptyLogger` with one of the other types will lead to detailed output describing each invocation of every combined function, which can be used to trace eventual problems with the framework behavior.

Analytics and control access

To provide a more systematic access to the framework behavior, an additional set of methods was made available on the `AdaptiveFunctionN` type by extending the `AdaptiveFunctionControl` and `AdaptiveFunctionAnalytics` traits:

- `train` – trains the combined function on a given set of inputs by executing all of the functions on each of the inputs and storing the results
- `flushHistory` – flushes the entire run history of all the implementations of given function[7] (in the corresponding history storage)
- `setPolicy` – sets the current policy of the function to a custom one
- `resetPolicy` – sets the current policy of the function to the starting one
- `getAnalyticsData` – retrieves analytics data regarding the function history

The **analytics data** represent a way of monitoring the decisions of the selection process for each combined function. They contain a complete set of records of previous runs, each one holding the identifier of the selected function, the *input*

[7]Note that this method can be called on the original implementation functions as well in order to flush history for only one of them. It will work thanks to the implicit conversion available.

descriptor, run time and the overhead time spent on the selection. The data can either be analyzed directly in the application by processing these records, or can be easily serialized into a CSV file.

Note that the analytics data collection is relies on the `AnalyticsCollector` trait. Again, we provide two implementations, `BasicAnalyticsCollector` that collects all the data mentioned, and `EmptyAnalyticsCollector`, that does not collect anything.

5.6.3 Framework configuration

In section 5.5, we mentioned a variety of building blocks that the main parts of the framework is composed of and the possibility to use multiple implementation to change the behavior of the entire system. This is the more advanced part of the configuration and it involves manipulation with the actual implementations.

As explained earlier, all the shared and commonly accessible functionality is held inside the singleton object `AdaptiveInternal`. This object can be initialized using an `initialize()` method on a publicly accessible singleton `Adaptive`. The static description of the module implementation composition that is followed in this process is provided by the `Configuration` trait. This can be thought of as a composition root of the whole system – it has to provide factory methods for all the building blocks used. In addition, it has an abstract type member `TMeasurement` that determines the type of the evaluation data used throughout the whole framework.

The implementation of the `Configuration` trait has to provide a specific `TMeasurement` type and factory methods that rely on the type. Note that the `AdaptiveSelector` trait and its methods do not depend on the `TMeasurement`, only its implementation does, so the type of the evaluation data does not leak outside of the composition root or to the API. As a result, the actual type can be changed at runtime by simply replacing the `AdaptiveSelector` instance.

The `Configuration` can be created by the user himself, using either a combination of provided implementations or his custom implementations in the factory methods. Alternatively, one of the predefined `Configuration` traits with some basic settings already in place can be utilized:

- `BaseConfiguration` – provides basic configuration of some of the types that do not depend on the `TMeasurement`
- `BaseLongConfiguration` – extends the `BaseConfiguration` and sets the `TMeasurement` type to `Long`

These traits have to be extended to define the unimplemented values.

Configuration blocks

To simplify the configuration process when working with existing implementations and to hide the initialization details from the user, a concept called *configuration blocks* was introduced.

Configuration blocks are *mixin* traits (see section 1.4) that provide implementation for just one (or a few) factory methods from the `Configuration` trait,

setting up part of the framework in given way. The user of the framework can use them to define the configuration by creating an anonymous class extending the base configuration along with desired *configuration blocks*. Thanks to the fluent and simple syntax of Scala, this concept is very expressive and easy to use. The compile-time check will automatically alert the user if the combination of block does not cover some required method.

In addition, some blocks can be parametrized. The parameters are always protected read-only attributes of the block trait that usually have default implementation (i.e. value), but can be overridden. It fits easily into the anonymous class usage pattern, as the new parameter values for all the blocks can simply be stated in the body of the class. Blocks can share a parameter by inheriting it from the same artificial base trait – in such a case, it is impossible to set different values for different blocks.

Some blocks call the parent implementation of the factory method to wrap its result into a different type. An example is the `CachedRegressionStorage` block, which creates a wrapper for caching the linear regression model atop of the `RunHistory` provided by its parent (for information about the decorating wrappers see 5.5). The parentage in the configuration inheritance chain is determined using *linearization* (see section 1.4). We have to be careful when mixing in a block with implementations and a wrapping block at the same time – the block with implementation has to be listed before the wrapping block in the extensions list.

The whole configuration can look the following way:

```
1  val config = new BaseLongConfiguration
2    with DefaultHistoryPath
3    with RunTimeMeasurement
4    with WindowBoundRegressionInputBasedStrategy
5    with UTestMeanBasedStrategy
6    with CachedRegressionStorage
7    with CachedStatisticsStorage
8    with FileLogging {
9    override val maximumNumberOfRecords = 20000
10   override val alpha = 0.25
11   override val logFilePath = "./adaptive/log.txt"
12 }
13
14 Adaptive.initialize(config)
```

List of all the configuration blocks available along with their parameters can be found in attachments.

Default configuration

A special class for the default configuration was created and is used to initialize the ScalaAdaptive framework at the beginning of the application run. It specifies the implementations that are used if the user does not provide his configuration.

Based on the results observed in section 4.5, we decided to include the t-test selection strategy with cached statistics as the default mean based strategy, and

the window-bound linear regression as the default input based strategy. The value of *alpha* for both of these strategies was set to 0.05.

The default configuration can be extended with any of the configuration blocks as well:

```
1 val config = new DefaultConfiguration with ConsoleLogging
```

5.6.4 Extending the framework

The framework can be extended without actually modifying it by creating custom implementations of some of the traits that are used by the invocation and selection process, and by supplying it to the `Adaptive` initialization using a custom `Configuration`, as described in 5.6.3.

An overview of the areas that are the simplest and most useful to extend will follow.

Selection strategies

A new selection strategy can be created by extending the `SelectionStrategy` trait. It can be used in the `Configuration` as either input based or mean based selection strategy[8]. The trait itself is technically a single function corresponding to the algorithm mentioned in 4.2.3, which is given a sequence of function run histories and an *input descriptor* and is supposed to return the key (function identifier and group identifier) of the function with best expected performance:

```
1 def selectOption(records: Seq[RunHistory[TMeasurement]],
2   inputDescriptor: Option[Long]): HistoryKey
```

The strategy will most likely have to work with either a specific `TMeasurement` data type, or at least put a constraint on it (by using the *viewability* mechanism, see section 1.3.1).

History storages

If we wanted to change the way in which the history data are stored (by filtering some of the results, performing some aggregations to save space, etc.), we would need to implement one or both of the following traits – the `RunHistory` trait, that represents the sequence of results for one group and function, and the `HistoryStorage` trait, that manages the `RunHistory` instances for different groups and functions.

Evaluation data

We can change the `TMeasurement` data type and make the framework perform the adaptations according to a different (or more complex) evaluation results. In such a case, we need to implement the `EvaluationProvider` trait, and then add our custom selection strategies that select based on the new evaluation type.

[8] It is also possible to configure new strategy for the *gatherData* operation, but that is not considered to be an interesting case

5.7 Framework distribution and usage

The framework is distributed in a form of JAR package containing all its compiled classes, which can be found in the attachment A. The simplest way to set up ScalaAdaptive in any Scala project is to use the SBT (see [25]). The `scalaadaptive_2.11-1.0.jar` file needs to be placed into the `lib` folder in the project root. SBT will automatically locate the content of the archive make it available in the compilation process.

Now, in any of the Scala source files in the corresponding project, the following line can be added:

```
1 import scalaadaptive.api.Implicits._
```

From now on, the original Scala function types have been enhanced with the ScalaAdaptive methods, including the most important `or` operator that allows creating functions with adaptive execution (see section 3.2).

For a quick introduction of the framework usage, a simple user guide based on examples is included in the attachment A, along with the complete generated source code documentation.

6. Framework evaluation

This chapter will present some applications and practical usages of the ScalaAdaptive framework, and evaluate the benefits and problems that it brings. Some problems and drawbacks of the adaptivity in a larger project in general will also be discussed.

Note that all the tests in this chapter, if not specified otherwise, were performed on a laptop with quad-core 1.3 GHz Intel Core i5 processor and 8GB of RAM. The tests and the resulting data are available in the attachment A.

6.1 Simple applications

We are going to suggest a couple of situations where the adaptive style of programming could be used. The problems will be described and an example usage of the ScalaAdaptive framework will be provided. A series of tests will be executed for each case, with results showing the impact that the adaptivity had on the overall performance.

6.1.1 Sorting algorithms

The first test of the ScalaAdaptive will be motivated by the example case of almost every lecture about algorithm complexity – sorting. We are going to try combining an implementation of selection sort with complexity $O(N^2)$ with an implementation of quick sort, which belongs to a category of algorithms with complexity $O(NlogN)$. From the theoretical point of view, the quick sort should be a better choice for every input. In practice, however, it is faster to use selection sort for smaller inputs, as the quick sort has an overhead connected to the recursive nature of the algorithm.

For our two implementations, we found out experimentally that the selection sort tends to be faster for inputs of less than 1000 items.

We are going to combine the two sorting algorithms using the following code:

```
1 val customSort = (
2   quickSort _ or selectionSort
3   by (_.length)
4   groupBy (d => GroupId(Math.log(d.length.toDouble).toInt))
5   selectUsing Selection.InputBased
6   withPolicy new PauseSelectionAfterStreakPolicy(20, 20)
7 )
```

The test consists of randomly generating 500 sequences of 0 to 5000 random integers. All the sequences are then sorted using the quick sort, selection sort and the combined function in configuration with the following selection strategies:

- Linear regression (LR) with $\alpha = 0.05$
- Window-bound linear regression (WBLR) with $\alpha = 0.05$
- Local regression (LOESS)

The sorting times of all the sequences were added up and the results are available in table 6.1. As we can see, using combined function was slightly worse than using quick sort in all cases in this scenario.

	Total time (ms)
Quick sort	1203.93
Selection sort	3493.43
Combined (LR)	1763.58
Combined (WBLR)	1605.37
Combined (LOESS)	1675.65

Table 6.1: Total times of sorting 500 random sequences of 0-5000 integers.

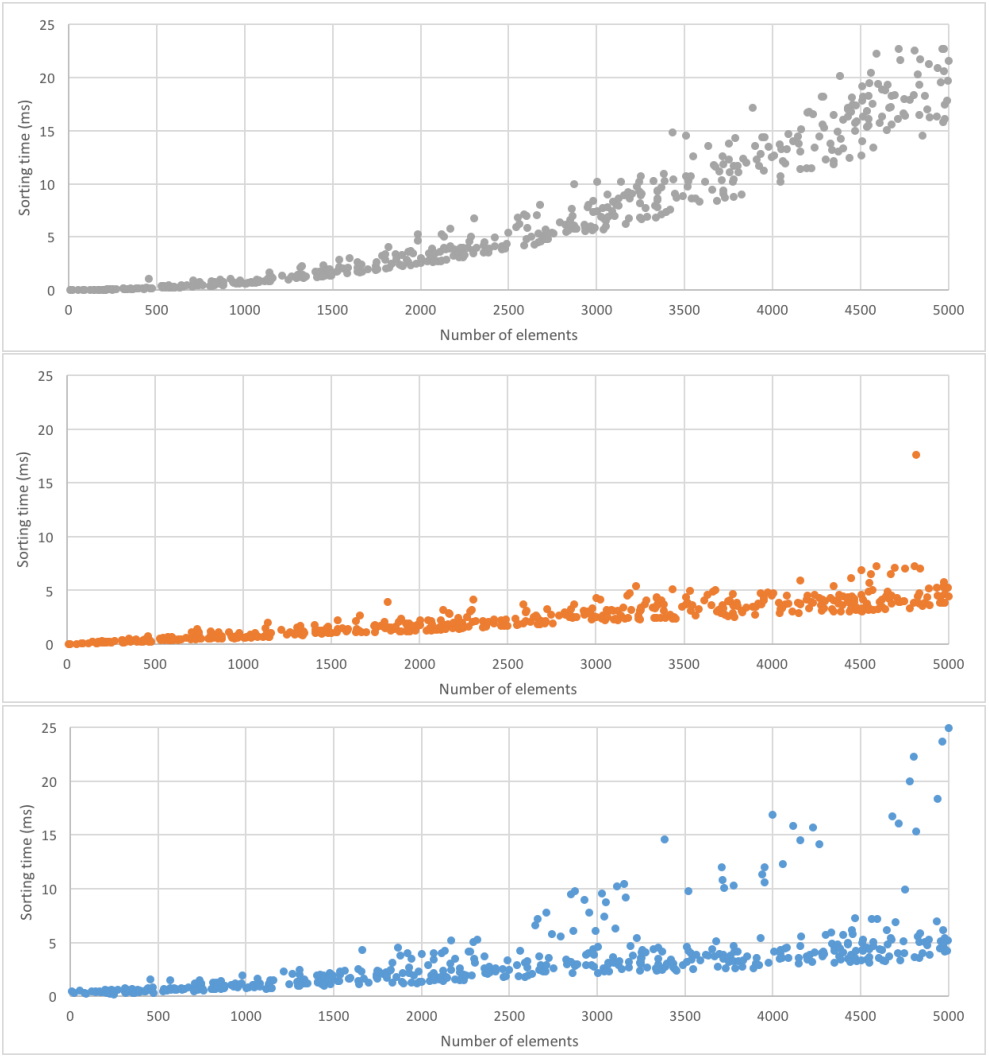

Figure 6.1: Sorting times of sequences of different sizes using selection sort, quick sort and a combined function (from top to bottom).

Figure 6.1 shows us the execution times of quick sort, selection sort and the combined function (using the window-bound linear regression strategy) for different input sizes from this test. As we can see, the combined function times are mostly similar to the quick sort times. The figure 6.2 shows only the interval between 0 and 1200, where selection sort tends to be better. The combined function has the worst performance, which is most likely caused by the selection overhead (see section 4.5.3).

Figure 6.2: Detailed view of the sorting times of sequences of different sizes using selection sort, quick sort and a combined function (from top to bottom).

In conclusion, ScalaAdaptive is not suitable for performing optimization for very small inputs of various algorithms. The overhead time leads to worse performance overall, even though the selection is working fine and the better options are correctly selected.

6.1.2 Matrix multiplication

Another example of an algorithmic problem where ScalaAdaptive can be used is matrix multiplication. The basic multiplication algorithm has complexity of

	Total time (s)
Basic algorithm	2230.21
Strassen algorithm	2168.29
Combined (LR)	1673.63

Table 6.2: Total times of multiplying 200 pairs of square matrices of sizes 50-900.

$O(N^3)$ and tends to get quite slow for larger matrices. A lot of more complex algorithms with slightly better complexities were discovered and are still being improved. The best achieved complexity so far is $O(N^{2.373})$ in [26].

The practical problem with most of these fast algorithms is that they have high constants and therefore are not suitable for smaller matrices. In this test, we will use the ScalaAdaptive with the Strassen algorithm (complexity $O(N^{2.807})$, [27]), and with the basic multiplication algorithm. We will use a simple implementation of both algorithms from [28]. The goal of this test is not to find the fastest way to multiply matrices, but to simply demonstrate that adaptation can be used with this class of algorithms to reach optimal results with almost no effort.

We tried generating 200 pairs of square matrices of random sizes between 50 and 900 filled with random integers. Then, we multiplied each pair using the basic algorithm, the Strassen algorithm, and the combined function. The window-bound linear regression strategy with $\alpha = 0.05$ was used. Because the run times are quite high in this case, the minimal number of observations before selection was lowered to 5 (using the `LowRunAwareSelectionStrategy`, see section 5.5).

The results can be found in table 6.2. As we can see, the overall results are in favor of the combined version. If we examine the relation between the run time and the size of the matrices in figure 6.3, we can notice the unusual character of the Strassen algorithm which is caused by it rounding up the matrix to the size corresponding to a closest power of two. This kind of an input dependency does not work well with linear regression, especially around the edges, and should optimally be used with logarithmic grouping and a mean based strategy. Nevertheless, the selection process worked fine enough and chose the optimal algorithm for almost all the inputs.

The matrix multiplication algorithms represent an ideal problem for the adaptive execution. Current approach to designing a fast library method for multiplying matrices would consist of measuring the matrix size limit where the more complex algorithm becomes faster than the basic one and then putting a branching on a fixed size test in the method, deciding which one to use. With ScalaAdaptive framework, this whole process can be skipped, and if the algorithms change, the system will adapt, finding the new limit by itself.

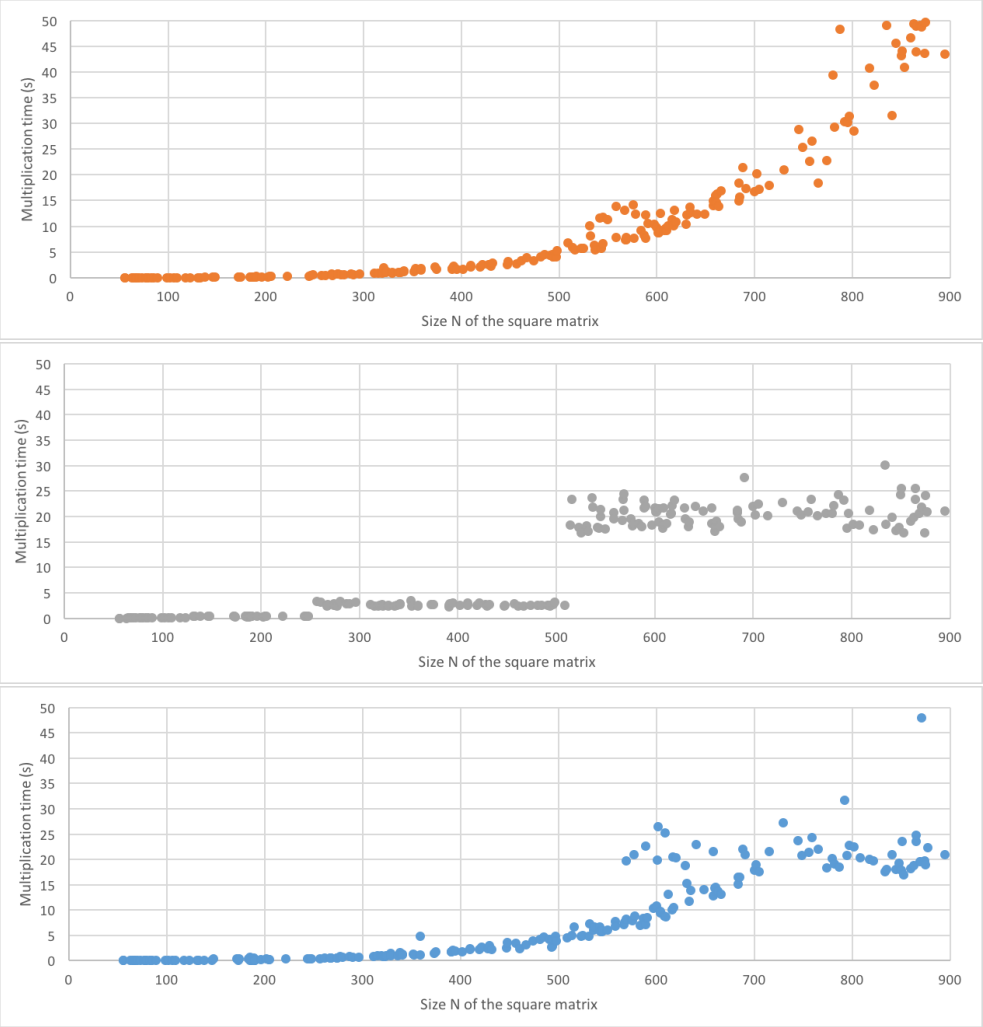

Figure 6.3: Multiplication times of matrices of different sizes using basic algorithm, Strassen algorithm and a combined function (from the top to the bottom).

6.1.3 JSON parsing

JSON[1] is a data-exchange format that can hold serialized object trees and collections. It is based on the notation for object (technically key-value dictionaries) literals in JavaScript. The format has recently become extremely popular due to its simplicity, data efficiency and readability, and as of today, is basically a standard for HTTP REST APIs. Most of the systems with distributed architecture that communicate over such an API (typically client-server applications) need to serialize and deserialize JSON upon sending a request or receiving a response, which might occur quite often.

The problem concerning JSON and other similar formats (XML, Protocol-Buffers, etc.) is that there is a variety of libraries available to perform the serialization and deserialization, and we have to decide for one when designing our application. The API tends to be similar and since it is used with such a frequency, optimizing its performance might be a suitable goal. There are per-

[1] JavaScript Object Notation

formance tests and comparisons of the libraries available, but as shown in [29], each one might be suitable for a different use case. In addition, [30] shows that the performance might significantly vary in different versions.

This library decision problem is a suitable use case for ScalaAdaptive – we can create a simple wrapper library with the same API as the original libraries that will expose the functions combined from the original serialization and deserialization methods. The user will be abstracted from the ScalaAdaptive, and the libraries can be added or removed at any time very easily.

Combining GSON and Jackson

In [29] we can find a benchmarks of the most popular JSON libraries from the Java world. For the test, we chose to use GSON[2] and Jackson[3]. We will measure only the deserialization part (parsing of string containing JSON), which is the more time-complex operation.

The combined function was created in the following way:

```
1 val parse = (
2  parseWithGson _ or parseWithJackson
3  groupBy ((json, c) => GroupId(Math.log(json.length).toInt))
4  selectUsing Selection.MeanBased
5  withPolicy new PauseSelectionAfterStreakPolicy(10, 200)
6 )
```

The test is based on parsing three different JSON strings:

- Small – 9KB, 10 records array
- Medium – 80KB, 100 records array
- Large – 8MB, 1000 records array

We know that we deal with fixed sizes of input files, so we can use the default t-test selection strategy without any problems, because every size will occupy one of the groups. If we knew that the sizes within a group might vary, we should use some other strategy. The value $\alpha = 0.05$ is used.

During the test, we ran each one of the three parsing functions 10000 times on the small JSON, 5000 times on the medium JSON and 200 times on the large JSON. We added up run times on these inputs for all the functions, and the results can be seen in the table 6.3.

As we can see, the combined function time is a little worse that the better alternative for each input size separately. But since the better alternative changes for different inputs, the total time of the combined function is the best.

We can compare the results with table 6.4 showing the same test, only without the **Pause selection after streak** policy. We see that by actually performing the selection every time, the overhead grows, which is noticeable especially on the small and medium inputs, where the results are significantly worse. Applying the overhead-limiting policies therefore makes sense and can cause significant performance differences.

[2]Version 2.8.1 of the GSON library from Maven repository was used in the test.
[3]Version 2.8.8 of the Jackson library from Maven repository was used in the test.

	GSON	Jackson	Combined
10000 * Small JSON	1830.36	4883.24	2566.54
Small JSON average	0.18	0.49	0.26
5000 * Medium JSON	4865.89	3142.05	3059.18
Medium JSON average	0.97	0.63	0.61
200 * Large JSON	15790.47	7298.75	7740.31
Large JSON average	78.95	36.49	38.70
Total time	22486.71	15324.05	13366.03

Table 6.3: Results of the JSON parsing tests (times in ms).

	GSON	Jackson	Combined
10000 * Small JSON	2180.20	5615.20	5667.64
Small JSON average	0.22	0.56	0.57
5000 * Medium JSON	4696.52	2944.04	3889.25
Medium JSON average	0.94	0.59	0.78
200 * Large JSON	15829.39	7433.50	7772.41
Large JSON average	79.15	37.17	38.86
Total time	22706.11	15992.73	17329.30

Table 6.4: Results of the JSON parsing tests without the policy applied (times in ms).

6.1.4 Load balancing

A different sort of problems that ScalaAdaptive might help with concerns adapting systems to environment changes. We can demonstrate this on a simple example – an application that relies on a remote web service running in multiple instances on different web servers. The service itself is stateless, so the requests can be performed on any of them.

The main goal of such a replication is usually to distribute the work between more nodes, so that the response times remain low. A technique called load balancing should take care of distributing the requests evenly between the nodes often just using a simple round-robin technique. This, however, has to be done on the side of the service, through a common gateway.

We will use ScalaAdaptive to implement a simple load balancing on the side of the client by evaluating the response times and selecting the target node using our selection strategies. We will take advantage of the possibility to limit the maximal age of the historical data (see section 4.1.4) – we want to make decisions

based only on the most recent measurements, because we suppose that they will change rapidly.

There are two goals of this load balancing example:

- To provide better performance than relying just on one node
- To limit the load on the node that is currently under pressure (and thus slow)

Suppose we have a simple web service running on multiple nodes and a system that performs synchronously running requests to these nodes. The request has the following form:

```
1 def request(url: String)(query: String): Option[String] = ???
```

We can take advantage of the possibility to curry the function and create the composed request that works with multiple nodes using the ScalaAdaptive API in the following way:

```
1 private def getRequest(s: Server) =
2   IdentifiedFunction(performRequest(s.url) _,
      s"request_${s.name}")
3 val balancedRequest =
4 (
5   servers.tail.foldLeft(getRequest(servers.head)) { (f, s) =>
6     f.or(getRequest(s)) }
7   selectUsing Selection.MeanBased
8   limitedTo Duration.ofSeconds(20)
9 )
```

Note that we need to use the custom identifier feature as mentioned in 5.3.2, because the function is combined from a single eta-expanded method with different arguments fixed in the currying process. The closure type is therefore the same for all the resulting functions. The selection will be done using the default t-test strategy with $\alpha = 0.05$.

For the actual test, we used two instances of a simple custom web service running on different ports of the same machine. Its implementation can be found in attachment A. We simulated the problems with load on the machines by artificially changing the response times of the servers every 50 seconds according to a set of predefined scenarios. From the client application, we sent one request every 0.5 seconds directly to each one of the servers and one using our balanced method.

The results for three different scenarios can be seen in figure 6.4. The lines represent response time evolution in time – direct requests are marked with double line and dotted line, the balanced request is solid. We can observe that most of the time, the solid line copies the lower of the two remaining lines. Sometimes, it repeatedly reaches the higher line, which means that the balanced request is sent to the slower server in order to gather fresh run data.

The average response times in these scenarios are shown in the table 6.5. The balanced response time is always lower than the worse of the response times. In case where the load alternates between the two servers (scenario 2), the balanced

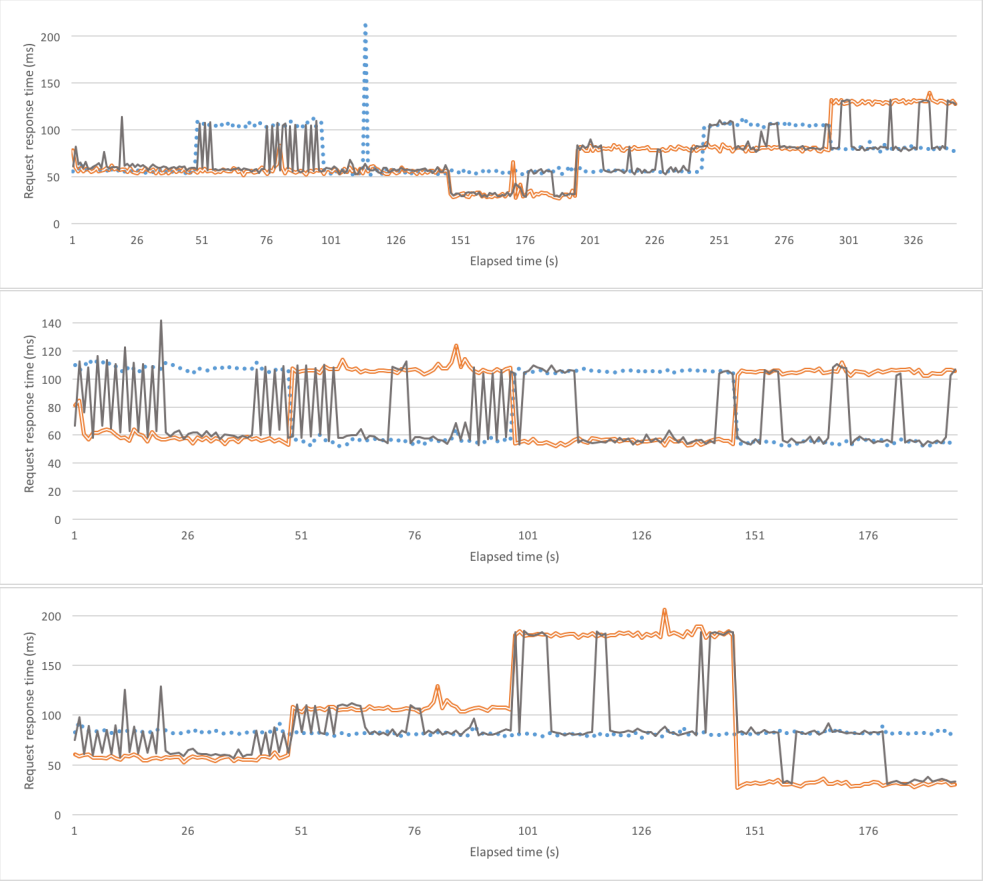

Figure 6.4: Time evolution of response times for request methods in scenarios 1, 2 and 3 (from top to bottom).

version has the best average response time. If, on the other hand, only one server fluctuates and the other one keeps a relatively low response time (scenario 3), the balanced version is slightly worse than the optimal one. Nevertheless, the second goal of moving the load between the servers is completed in all three scenarios.

	Server 1	**Server 2**	**Balanced**
Scenario 1	74.20	70.48	68.38
Scenario 2	80.92	81.77	72.52
Scenario 3	81.97	94.39	85.38

Table 6.5: Average response times for the request methods in scenarios 1, 2 and 3 (in ms).

A disadvantage of ScalaAdaptive that complicates the usage in this area is the fact that it does not support measuring run times of asynchronous functions, i.e. functions that perform callbacks or complete promises upon finishing.

6.2 Apache Spark

Apache Spark ([5]) is a framework for distributed computing that is focused on data processing in a similar way as the MapReduce distributed computation paradigm. It works with distributed data sets that can be mapped, filtered, grouped or reduced, but unlike the implementations in Hadoop or other systems, it is focused on in-memory data processing on the nodes. Currently, it is one of the most used systems in big data processing, designed to work with a large variety of distributed storage systems (e.g. HDFS, Cassandra, OpenStack Swift).

The Spark framework represents one of the most interesting possibilities to use the ScalaAdaptive framework. The data processing queries are long-running tasks, so any overhead imposed by the selection is negligible. On the other hand, the potentially saved time on a faster implementation increases. In addition, the distributed computation model and the necessity to create an execution plan which depends on the data locations, sizes, etc. leads to large differences in execution times based on:

- The query itself
- The cluster it is running on (number of machines, cores, network)
- The configuration of Apache Spark

The goal of this section is to show the basic ideas where the adaptivity might be used in such a framework.

6.2.1 Spark APIs

Spark currently supports three different APIs that can be used to construct Spark queries.

RDD

RDD (Resilient Distributed Dataset) was the first API introduced and is basically a distributed dataset of JVM objects. Spark is not aware of the structure of the internal JVM objects, it is treating the individuals like blackboxes. The RDD query is typed and constructed using lambda expressions that work with these objects:

```
1 rdd.filter(_.id > 5000).count
```

Upon execution, the Spark worker engine manipulates with the RDDs as a whole (partitions it, collects the results, etc.), but the operations on its members (filtering, transformation, grouping) are carried out by actually executing the lambda functions.

This approach is very simple and allows the user to write queries over custom data types without any limitation. The downside is that the Spark execution engine does not know what is going on within the lambdas, cannot optimize the execution plan based on the query, and, when moving the parts of RDDs between the nodes, the objects have to be serialized and deserialized again.

DataFrames and Spark SQL

DataFrames API was introduced later into the Spark framework with a goal of solving the problems of RDDs. The data within DataFrames are described by a schema, somehow similar to classic SQL schema – Spark knows exactly how are the data records structured. The queries are constructed using Spark SQL, a query language (DSL in Scala) that uses textual names of the record attributes (i.e. schema columns). As we can see in the following example, the actions are described by strings and parsed by Spark upon execution:

```
1 df.filter("id > 5000").count
```

Spark therefore knows how exactly the mapping, filtering, reducing or grouping is going to happen, and can use these information to optimize the execution and data distribution, to make estimations, etc., when building the query plan. In addition, the data described by a schema can be easily persisted or transfered between nodes without the need to perform a JVM serialization. A major disadvantage is the necessity to provide the schema and the fact that the actual query is parsed at runtime and is untyped – errors in attribute names or conditions in general will not be detected upon compilation.

Datasets

The newest Spark API, Datasets, was designed with an objective of combining the best from RDD and DataFrame approaches. The query programming API is typed and works with JVM objects, just like RDD, but the internal representation is schema-based, like in DataFrames. In order for this to work, Spark uses a mechanism of encoders that can transform the JVM representation to the internal one and vice-versa. It also analyzes the content of the typed lambdas and uses the same optimization as with the DataFrames API.

6.2.2 ScalaAdaptive tests

To perform the test of ScalaAdaptive with Apache Spark, two environments were used:

- Spark local mode with 1 worker[4] on a laptop with quad-core 1.3 GHz Intel Core i5 processor and 8GB of RAM
- Spark local mode with 8 workers[5] on the same machine
- Cluster of 12 Spark workers deployed in Docker containers on 3 machines, each with two 8-core Intel Xeon CPU E5-2660 0 @ 2.20GHz and 48GB of RAM

Randomly generated data were used for the test, no distributed storage was involved. The tests were running with the `StorageLevel.MEMORY_ONLY` data persistence configuration. The version 2.1.0 of the Apache Spark library was used.

[4]Using `master("local[1]")` setup.
[5]Using `master("local[8]")` setup.

Query selection

The most simple adaptation case that can be applied to any data processing or querying framework is the query selection. A data operation can usually be expressed using a variety of queries that differ in the order or character of the elementary steps. These queries can have different performance, based on the optimizations that the executing engine can perform, especially in the distributed environment where the main concern is keeping the amount of data transfered to the minimum.

An example of such a query performance variation can be the *group* versus *reduce* problem. Suppose have a set of key-value pairs and we want to group them by key and then reduce the groups (by aggregating their content somehow). The straightforward approach is to group the data and to map the aggregation function over the grouped values:

```
1 data.groupByKey()
2     .map(i => (i._1, i._2.flatten.toArray))
```

Because the execution engine does not understand the mapping action (RDDs do not analyze the queries), this requires the grouping to be done across all the workers to physically create the groups, even though we are just going to immediately reduce them.

To address this issue, a special operation for reducing is present in the API:

```
1 data.reduceByKey((arr1, arr2) => arr1 ++ arr2)
```

In this case, the reduced value can be first computed for each node out of the records with given key present on the node. Then, the partial reduction results can be passed around and further aggregated.

A test with ScalaAdaptive was performed on a 5000 key-value pair sequence in Local[8] mode. The two RDD queries presented in this section were combined together and the resulting function was executed 1000 times. The framework decided correctly for the reduce variant using the default t-test with $\alpha = 0.05$. The analytics data show the following average run times of the queries:

- **GroupByKey** – 3677.61ms
- **ReduceByKey** – 684.52ms

RDDs or Datasets

As explained in section 6.2.1, Spark has various APIs that allow the user to perform the query. Theoretically, the Dataset API should offer the same expressiveness and better performance. A lot of current production Spark code is, however, written in RDDs, as it was the first (and the only for some time) presented API. This poses a question whether the code should be kept and maintained in RDDs, or whether it would be better to replace it with Datasets and what the eventual performance gain would be.

The ScalaAdaptive can be used in this case to limit the potential negative impact of this change. The Dataset-based queries can be implemented into the system and wrapped along with the old RDD queries into a single combined query. If there was a place where the Dataset query would, for some reason, have worse

performance, the ScalaAdaptive framework should discover it and keep using the old RDD query. In the longer term, the results of ScalaAdaptive selections can be evaluated and used to discover problematic queries and to decide about next steps.

In this test, we are going to try combining two queries from the two different APIs and try them out in different environments and with different data sizes. We are going to track the execution times of both of them in the phase where the data is gathered in a round-robin manner, and then analyze the decision made by the ScalaAdaptive framework. We would preferably like to find out whether the decision will be the same for all the data and environments, or whether there are going to be significant differences.

For the test, we implemented the following two queries in both RDDs and Datasets:

- **Query 1** – generates n random key-value pairs using parallelized RDD generation, groups by the key and counts the groups
- **Query 2** – generates n records with multiple attributes from in-memory sequence, filters them by an attribute, groups them by a different attribute, reduces the groups into new records, filters the records again and groups them for the second time

The RDD and Dataset option of each of the queries were combined together and the result was executed 100 times in various environments and with multiple values of n. The average execution times of the queries, as retrieved from the analytics data (see section 5.6.2), can be seen for all the cases in table 6.6. The last column shows the ScalaAdaptive decision after gathering enough data – default t-test selection strategy with $\alpha = 0.05$ was used.

We can see that it is definitely not possible to assume that Datasets are generally faster than RDDs. The only situation where Dataset got better performance was the Query 1 test with the largest amount of data. We can observe that at least for Query 1, the Datasets perform better for larger data sizes, both locally and in the cluster. For Query 2, it seems that Dataset performance is constantly around 7 times worse than the RDD performance.

These observations might be caused by Datasets having the operations optimized to different cases, or even by our wrong selection of Dataset alternatives of the RDD queries. But such a case might be a real scenario where the ScalaAdaptive framework can be successfully employed and help to discover a potential problem.

Spark SQL adaptive execution

Spark queries are divided into stages, where each stage can be performed on a single node without the need to exchange any data. Typically, multiple maps or filters are chained within one stage, because every record can be mapped without the necessity of any other data. On the other hand, join, group and reduce operations have to be done at the beginning of a new stage, because they can potentially require data from other nodes. Before every stage, a shuffle is performed – the data is exchanged between the nodes.

	RDD	Dataset	ScalaAdaptive decision
Query 1: Cluster 500000	920.97	1633.84	RDD
Query 1: Cluster 2000000	7105.58	7068.34	Cannot decide
Query 1: Cluster 5000000	37140.28	3587.97	Dataset
Query 1: Local[1] 200000	2426.95	12554.66	RDD
Query 1: Local[8] 200000	1499.14	7446.45	RDD
Query 1: Local[8] 1000000	12070.14	12861.32	Cannot decide
Query 2: Cluster 100000	1152.19	6956.03	RDD
Query 2: Cluster 1000000	2162.72	14311.13	RDD

Table 6.6: Run times (in ms) of Spark queries on RDD and Datasets in various environments.

The data is divided into partitions and each node processes a specific subset during the stage execution. Upon shuffle, some partitions get exchanged. Originally, the partitioning of the data following each shuffle was fixed and based on the original partitioning – it did not reflect the actual results of previous stages, which could potentially lead to some nodes having a lot less work than the others if there were partitions with majority of data filtered out in previous stage. To solve this issue, an experimental feature called *adaptive execution*[6] was introduced into Spark SQL. It causes that the result sizes from previous stages are collected and a new partitioning is created before performing the shuffle.

The *Spark adaptive execution* feature is turned off by default. We can use ScalaAdaptive to experiment with the performance impact when enabling it. In addition, there are two configurable attributes of the feature:

- `targetPostShuffleInputSize` – the optimal size of a partition (will be targeted during shuffle)
- `minNumPostShufflePartitions` – the minimal number of partitions (will be strictly kept)

We can try a custom value for these attributes in one of the ScalaAdaptive combinations.

The usage of ScalaAdaptive in this case will, however, be a little more complicated. Some attributes of the Spark SQL configuration can be changed at any point in the execution, but doing so is not reliable, especially in large cluster – changes do not seem to be reflected immediately. A safer approach is to create the SparkSession already with the target configuration. If we create new session for each query, we can incorporate the session creation into the query function. If we, on the other hand, use one session for multiple queries, we need to split the selection process (session creation) and the measurement process (query execution). The only solution is the delayed measurement model described in section 3.2.6.

[6]Should not be mistaken with the adaptive execution of *ScalaAdaptive* framework.

The test was performed on a Dataset of n records which was filtered, grouped by an attribute, the groups reduced to a single element and the result filtered again and counted. We tried to combine various values of n with the following setups:

1. *Spark adaptive execution* disabled (currently the default configuration)
2. *Spark adaptive execution* enabled with default attributes
3. *Spark adaptive execution* enabled
   `targetPostShuffleInputSize` = 512MB
   `minNumPostShufflePartitions` = -1 (unlimited)
4. *Spark adaptive execution* enabled
   `targetPostShuffleInputSize` = 2MB
   `minNumPostShufflePartitions` = 200

The SparkSession was created 12 times for every test case and the query was executed 10 times using each session (120 executions in total). As the `LowRunAwareSelectionStrategy` (see section 5.5) was set to a minimum of 20 records, each of the setups had to be used to create at least two sessions. The t-test selection strategy with $\alpha = 0.05$ was used.

	1.	**2.**	**3.**	**4.**	**ScalaAdaptive decision**
Local[8] 1000	4322.24	1472.51	1916.87	3528.62	Cannot decide
Local[8] 10000	6911.03	4499.11	4933.53	6320.43	Cannot decide
Cluster 20000	1234.36	725.79	3892.56	1172.69	2.
Cluster 200000	2879.25	2538.94	5639.02	2853.10	2.

Table 6.7: Average run times (in ms) of Spark queries with different configurations in various environments, and the configurations that ScalaAdaptive selected.

The table 6.7 shows the average execution times for the setups based on the initial 20 runs. It additionally marks if the framework reached a decision after having evaluated these runs. As we can see, the performances of the options vary a lot depending on the input size and on the execution environment. The default configuration (with *Spark adaptive execution* disabled) in the local execution on small data is by far the worst, in large cluster is among the better ones. The configuration number 2, i.e., the manually activated *adaptive execution* with default setup, is the best in all the cases, but we can see that the results tend to change a lot. Using ScalaAdaptive can therefore help discover eventual better configurations.

The disadvantage of ScalaAdaptive in this case is that selection from quite a lot of alternatives is required, which suffers from multiple problems discussed in chapter 4. In case of the results discussed, the test was not able to decide in the two local setups, which lead to looping through all the options including the worst ones.

6.3 Problems with practical use of the framework

Using adaptive framework in the development process is a new concept with many consequences. While designing, implementing and evaluating the framework, we discovered a few potential problems that can complicate its practical application.

6.3.1 Invocation overhead

The framework adds a non-trivial overhead to the function invocation process due to the selection and the storing of the evaluation data. In order to limit this, the policy-based invocation system was added to the framework with a goal of skipping the selection in some situations. Therefore, two different types of overhead time per invocation can be measured:

1. **Policy evaluation overhead**
   Part of every invocation. It is expected to be small and it should be constant throughout the whole application lifetime. It cannot be measured from within the framework, but it can be computed by measuring the combined function run time and then by subtracting the run time and overhead time from the `AnalyticsData` collected by the function.

2. **Selection overhead**
   Exists only when the SelectNew or GatherData result is given by the policy. It is determined by the actual selection strategy used and is in general expected to be longer and to be related to the amount of historical data available for the functions involved. It is tracked by the framework and included in the `AnalyticsData` and in addition, it is one of the statistics that the policies are based on.

The figure 6.5 shows these times tracked for a sequence of 20000 invocations (sampled every 50th run) of a sample combined function using the window-bound linear regression strategy (see section 4.4.2) and a policy that emits the SelectNew result every time. We can see that the selection overhead is significantly larger and really grows with the number of records. The policy evaluation overhead remains basically constant.

The average selection overhead from the sample discussed is **1.29ms** per invocation, the average policy evaluation overhead from the same sample is **0.06ms** per invocation. We can see that the policy evaluation overhead is reasonably low, so the key to limiting the overhead is using the correct policies and avoiding selection whenever possible. Analysis in section 4.5.3 also showed that there are important differences in overheads of the strategies, so the correct strategy selection is important as well.

The overhead always exists and has to be counted with, but its significance depends on the actual problem that we are solving.

6.3.2 Maintainability

One of the main problems that an adaptive framework faces in larger systems and projects is an engineering issue. The code of current systems has to be kept

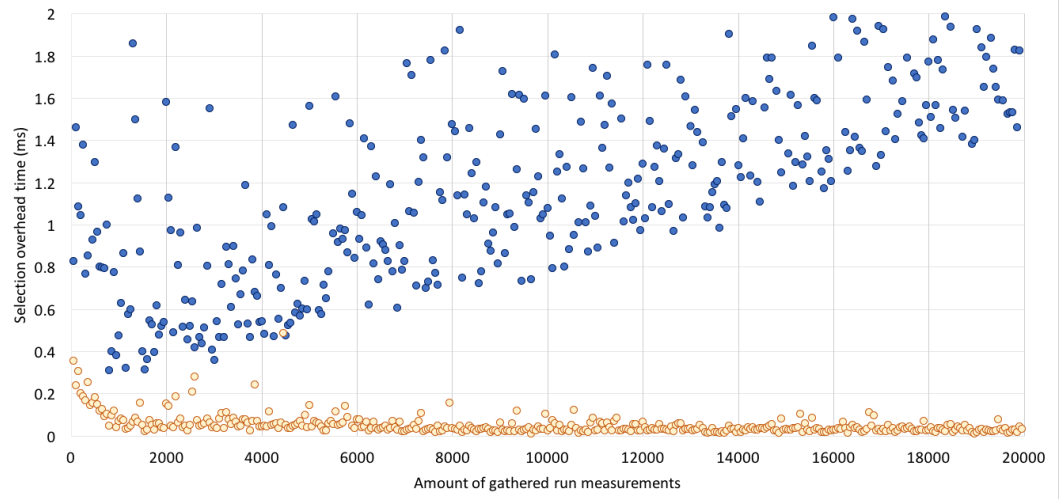

Figure 6.5: Selection overhead times (dark) and policy evaluation overhead times (light), both in ns, for a sequence of 20000 invocations of the same combined function.

working and maintained for several years, sometimes decades. During this period, bugs appear in the system, business requirements change and consequently, it is necessary to perform modifications in function code.

Maintaining multiple implementations of the same functionality at once brings a lot of issues. The developers have to know exactly how all the implementations work and whenever it is necessary to modify the behavior, be able to perform changes in all of them to achieve the same result. Described process itself is very demanding and has a significant time impact on the development.

What is more, subtle differences in behavior of the implementations could be unintentionally introduced. These differences might not have visible effects immediately and may appear after several other modifications. At that moment, it will be extremely difficult to locate the problem, because the misbehavior caused will not be deterministic thanks to the nature of the selection algorithm.

In cases where third-party libraries are used within the combined implementations, we introduce yet another factor of risk into the system – the libraries can change their behavior in newer versions, can have undocumented differences in their solutions of input corner cases, etc.

Using non-deterministic decision tools in general lowers the maintainability of the system, and the ScalaAdaptive framework is such a tool.

6.3.3 Testing and verification

When maintaining multiple implementations used in our program, there will have to be inevitable changes made to every one of them. Before being released, the system has to go through the verification process. Common techniques of regression testing are insufficient in this case due to the non-deterministic nature of the adaptive selection in the system – some of the implementations might not be tested at all during the regression test, but might be selected in the production environment.

The only way how to perform a regression test on the system is to change the

ScalaAdaptive framework configuration in the test environment, specifically the *default policy* and the *selection strategy*. The recommended settings are following:

- default policy: `AlwaysSelectPolicy`
- selection strategy: `LeastDataSelectionStrategy`

Using this setup and ensuring that there are no historical data present, the system will go through all the options in round-robin style upon every run. The test process has to be performed enough times for all the options to reach their turn.

The described testing process is cumbersome and can lead to errors in verification. For this reason, more emphasis should be put on unit testing in a project containing the adaptively selected functions. If all the implementations are well covered by unit tests, all the hidden bugs and deviations of behavior should be immediately detected. In fact, there could be one common set of unit tests covering all the implementations, so that the test cases and acceptation conditions were the same.

A simple library for generating the unit tests for all the implementation would be useful and could be introduced as a part of continuation of the project.

6.3.4 Same API requirement

In order to create a combined function from two different algorithms implemented in two different libraries or frameworks, we need to adapt them to share the same API. This could be a very simple task for some basic algorithms (converting number formats, wrapping arguments, etc.), but it might get complicated when more complex structures get involved. Even for such a basic example like sorting algorithms, we get a variety of interface options:

- Does it sort list, array or an abstract sequence?
- Does the algorithm sort in-place, or produces a new sequence?

Creating wrappers for the API that solve these issues is not trivial – data structures have to be converted, copied, etc., which will cause more unnecessary overhead. This overhead might get more significant with the complexity of data structures that we need to convert between the two APIs that we need to adapt.

7. Related work

A different approach for the same goal consists in manually selecting an implementation and guaranteeing its performance with unit tests. A special language, SPL (Stochastic Performance Logic), which allows to express assumptions about the performance of multiple functions, was introduced in [1]. The innovative concept is that it treats the performance as a random variable, not as a fixed value, so the conditions are evaluated with certain confidence. It was later used to examine the performance of JDOM framework in [2], and became the base of performance unit testing frameworks for C# ([30]) and Java ([31]).

The development of adaptive systems that can tweak their performance based on the environment is a widely studied topic as well. Systems that are capable of some form of *auto-tunning*, i.e., self reconfiguration based on an analysis of the execution environment (OS, CPU, etc...), are often among the fastest in its category – an example might be [32], which is considered to be the fastest non-commercial FFT[1] algorithm in the world. The adaptation in this case is done by recompiling the application code ([33]).

An idea of a universal adaptive framework that would not require manual analysis of the environment and work only with observations from actual execution is presented in [16]. It describes a system that has generally the same goal as our work – increase performance awareness by designing the systems adaptively. It is focused on large component-based applications where the performance of certain components is tracked. Multiple use case scenarios are presented, e.g. verifying component contract, observing trends, or selecting between multiple interchangeable components based on performance. The SPL is used to make the decisions, i.e., the components are guarded by conditions expressed in the language.

The article does not solve the issue of predicting run time with a certain input based on the performance observed with other inputs. The presented concept is also based on the interconnection between the adaptive framework and the system components. ScalaAdaptive provides a partial implementation of a similar system, more narrowly focused, with higher emphasis on the simplicity and efficiency of performing the single task of selecting faster implementation. In addition, it provides an API that allows easily using the adaptivity in any existing system with no need of architectural changes.

The task of predicting function (or application in general) run time based on the input is a separate problem examined in many different ways. In [34], a static analysis of LISP programs is performed in order to derive actual complexity. A runtime analysis approach was taken in [14], where basic blocks in the program called *locations* are identified and their performance is measured for a certain input described by a set of *features*. The performance is represented by an execution count in order to keep the measurement deterministic and platform or environment independent. A powerlaw regression is then built to find relations between *feature* values and *locations*. The resulting model can be used to formulate predictions.

Relatively precise predictions of an actual execution can be achieved using

[1] Fast Fourier transform

the Mantis framework [13], which is based on instrumenting the code and automatically identifying its *features* (loops, branches, variable values...). The *features* are evaluated (branch counts, loop counts...) at runtime and the machine learning process is used to select the ones that are important for the overall performance. A regression model is then constructed using the data. One of the possible models is described in [21].

Authors of [15] obtain the predictions in two steps – first, they use greedy or genetic algorithm to find similar inputs (jobs, workloads) in the historical run measurement, and then, they generate the prediction using either simple mean or inear regression, both with corresponding confidence interval. The basic approach taken here is quite similar to the ScalaAdaptive, even though neither this, nor the other works referenced in this chapter deal with the problem of real-time prediction upon invocation, where one of the main concerns is minimizing the overhead.

Conclusion

In this thesis, we designed and implemented a framework that allows a completely new style of performance-aware development by composing functions from interchangeable implementations. An API that allows fluent integration with almost no effort from the developer was introduced. Various statistical methods to identify the most appropriate implementation for given input based on historical performance observation were examined, implemented and compared.

The testing that was carried out identified some potential use-cases for this style of development. Among the more suitable ones were in general longer running functions, where the run time fluctuations were not as significant and where the selection overhead was negligible. We achieved better overall run times of adaptive functions on sequences of various inputs, either for an algorithmic problem (matrix multiplication), or in case of selection between two utility libraries (JSON parsing). On the other hand, optimizing fast-running functions for small inputs does not appear to be a useful application, as the overhead surpasses the potential benefit from adaptation.

In addition, we demonstrated the potential of environment adaptation on the Apache Spark distributed data processing framework, where different configurations and query types achieved better results depending on the cluster they were running on. The Spark itself seems to be a suitable use scenario for our framework, as it is being actively developed and many features have experimental character and get iteratively optimized.

Among the main drawbacks of the solution is the fact that the user has to identify the key feature of the input, which has to be one-dimensional (simple integer) in order to enable input based selection. The prediction models are not perfect, especially in cases where the observed performances of all the functions are very similar. In addition, upon selecting from more than two functions, the system might get stuck in a situation where the selection strategy cannot decide due to a pair of equally good variants and rotates all of them in a round-robin manner.

The majority of current problems and issues can be addressed in the future by extending the framework and adding new functionality. This can be done without actually modifying the framework code due to the modularity and runtime composition.

We believe that the biggest potential of this framework lies in its simplicity and possibility to be used in all kinds of systems without any structural changes. As we do not expect a common developer to implement one functionality multiple times and then combine it using our framework, the main use case in some larger project would most likely consist of combining multiple libraries, configuration, queries or other simply obtainable entities.

To see the actual benefit that this development approach can bring to a larger system, we would need to practically test it in such an application, i.e. a long running service, and track the performance changes. This kind of test would also expose other problems connected with either the framework or the whole concept of adaptive development that could not have been identified in the isolated artificial tests that were part of this work.

Future work

Improving selection from multiple functions

As mentioned in 4.1.6, the task of selecting the most suitable function out of more than two is complicated by its nature and we have taken a simplified approach in this work. In addition, the significance-based selection strategies have to perform multiple tests (or confidence interval constructions) in the process and the probability of an error increases. We can either keep the significance high and lower the strength of the test, which leads to more situations without any function being selected, or let the significance drop while keeping the strength, causing more selection errors.

The potential future extension could either completely redefine the goal of selection from multiple functions and design a new process, or extend the current solution by determining suitable significance corrections.

Extending the policy logic and connecting it to selectors

In section 4.6.5 we stated that the policies were currently relatively limited, both in the data they have to base their decision on, and in the results they can produce. It would be interesting to analyze possible extensions while keeping the evaluation overhead as minimal as possible. The policies could give hints to the selection strategies, limit the data that they are working with, etc.

Custom selection rules based on the SPL

The [16] introduces an extension of the SPL language designed to describe conditions based on the trends of historical performance data of functions. A special selection strategy could be designed to accept the SPL conditions and make decisions according to them. This would allow the framework user to have more control over the selection process.

Thread safety and concurrency

The framework in general is currently not thread safe. It is designed, however, with regard to the possible concurrency, so the modifications should not be dramatic. For the possibility to safely use the framework from a multi-threaded environment, we would need to ensure three basic points:

1. Thread-safe initialization and static runtime composition
2. Thread-safe access to the run history
3. Thread-safe access to the combined function state (run statistics and current policy)

The run histories are already prepared to be made thread-safe by having an API to support immutability (mutating operations return new instances) and an immutable implementation (with all the caches) as well. Because of that, only adding a simple lock on the history updating method should be enough. The third point is currently the biggest issue, because it has been optimized for performance and the statistics are mutable in general.

Optimizing run time of the whole system

Instead of optimizing the run times of separate functions, we could aim for optimizing the whole system. We could, for example, have a set of different data structures that our algorithm will be working with. The data structures would have all the necessary operations implemented and presented using a trait or an abstract class. The framework would generate a factory method that would create one of the implementations, and then track the overall behavior of different methods on the implementations.

```
1 val data = createDataStructure(list)
2 data.getMin()
3 data.getMax()
4 data.add(i)
```

Selection strategies supporting multidimensional input descriptors

As mentioned in section 4.1.2, algorithm complexities are quite commonly functions of multiple features of the input. In order to create selection strategies that would be able to correctly decide in case of such algorithms, we would need to add support for *input descriptors* with more features. The input based strategies would then have to construct multidimensional regression models and analyze the dependency on multiple features.

Detecting the input features

The fact that the user has to know which features of the input might affect the function run time and manually specify the *descriptor function* and the grouping is quite limiting. Another way of improving the framework might be some kind of analysis that would examine the history measurements and look for correlation between the input feature changes and the run time changes. The main limitation is that the input has to be some structure known to the framework in order to extract the features.

Implementation in other languages

The framework is implemented in Scala and can be used to combine functions for any JVM-based languages (see section 3.2.7). Transferring the whole framework to a different language and platform might be desirable in the future. In such a case, the main problem would be the API of the framework, which, in its current design, relies heavily on the Scala DSL features of Scala (see section 1.2). It requires the following features from the language:

- Implicit type conversions
- Functions as first-class values
- Extensibility of the function types
- Eta-expansion of methods
- Infix operator syntax for methods

- Macros to parse and modify the AST upon compilation

With only a subset of these features in the language, a limited version of current API could be designed.

The best approach for a different platform implementation would probably be to analyze the features that the target language offers and the common approach to designing APIs in that language, and to create a new, better suited API.

Bibliography

[1] Lubomír Bulej, Tomáš Bureš, Jaroslav Keznikl, Alena Koubková, Andrej Podzimek, and Petr Tůma. Capturing Performance Assumptions Using Stochastic Performance Logic. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 311–322, New York, NY, USA, 2012. ACM.

[2] Vojtěch Horký, František Haas, Jaroslav Kotrč, Martin Lacina, and Petr Tůma. Performance Regression Unit Testing: A Case Study. In *Computer Performance Engineering*, Lecture Notes in Computer Science, pages 149–163. Springer, Berlin, Heidelberg, September 2013.

[3] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr Tůma. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 289–300, New York, NY, USA, 2015. ACM.

[4] The Scala Programming Language. `https://www.scala-lang.org/`. [Online; accessed 2017-07-05].

[5] Apache Spark^{TM} - Lightning-Fast Cluster Computing. `https://spark.apache.org/`. [Online; accessed 2017-07-05].

[6] Gradle Build Tool. `https://gradle.org/`. [Online; accessed 2017-07-06].

[7] Chef – Automate Your Infrastructure. `https://www.chef.io/chef/`. [Online; accessed 2017-07-06].

[8] scalatest: A testing tool for Scala and Java developers. `http://www.scalatest.org/`. [Online; accessed 2017-07-03].

[9] Trait Linearization in Scala. `https://www.trivento.io/trait-linearization/`, October 2016. [Online; accessed 2017-07-12].

[10] wolfe: Wolfe Language and Engine - Scala AST reference. `https://github.com/wolfe-pack/wolfe`. [Online; accessed 2017-07-05].

[11] The AspectJ Project. `https://eclipse.org/aspectj/`. [Online; accessed 2017-07-12].

[12] Neil A. Weiss. *Introductory Statistics*. Pearson, Boston, 9 edition, December 2010.

[13] Byung-Gon Chun, Ling Huang, Sangmin Lee, Petros Maniatis, and Mayur Naik. Mantis: Predicting System Performance through Program Analysis and Modeling. *arXiv:1010.0019 [cs]*, September 2010. arXiv: 1010.0019.

[14] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM*

SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pages 395–404, New York, NY, USA, 2007. ACM.

[15] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting Application Run Times Using Historical Information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, pages 122–142, London, UK, UK, 1998. Springer-Verlag.

[16] L. Bulej, T. Bures, V. Horky, J. Keznikl, and P. Tuma. Performance Awareness in Component Systems: Vision Paper. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 514–519, July 2012.

[17] B. L. Welch. The generalisation of student's problems when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[18] William S. Cleveland and Susan J. Devlin. Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. *Journal of the American Statistical Association*, September 1988.

[19] William S. Cleveland, Susan J. Devlin, and Eric Grosse. Regression by local fitting. *Journal of Econometrics*, 37(1):87–114, January 1988.

[20] William S. Cleveland and E. Grosse. Computational methods for local regression. *Statistics and Computing*, 1(1):47–62, September 1991.

[21] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, NIPS'10, pages 883–891, USA, 2010. Curran Associates Inc.

[22] Railway Oriented Programming | F# for fun and profit. `https://fsharpforfunandprofit.com/rop/`. [Online; accessed 2017-07-05].

[23] Know the JVM Series: Shutdown Hooks - DZone Java. `https://dzone.com/articles/know-jvm-series-2-shutdown`. [Online; accessed 2017-07-12].

[24] Math – Commons Math: The Apache Commons Mathematics Library. `http://commons.apache.org/proper/commons-math/`. [Online; accessed 2017-07-12].

[25] sbt - The interactive build tool. `http://www.scala-sbt.org/`. [Online; accessed 2017-07-12].

[26] Virginia Vassilevska Williams. Multiplying Matrices Faster Than Coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM.

[27] Volker Strassen. Gaussian Elimination is Not Optimal. *Numer. Math.*, 13(4):354–356, August 1969.

[28] JLinAlg - Open Source And Easy-to-Use Java-library For Linear Algebra. http://jlinalg.sourceforge.net/. [Online; accessed 2017-07-03].

[29] Josh Dreyfuss. The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP. http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/, May 2015. [Online; accessed 2017-07-12].

[30] Tomáš Trojánek. Capturing Performance Assumptions using Stochastic Performance Logic. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2013.

[31] Jaroslav Kotrč. Run-time performance testing in Java. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2015.

[32] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998*, volume 3, pages 1381–1384 vol.3, May 1998.

[33] Matteo Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[34] Ben Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, September 1975.

List of Figures

4.1 Linear regression model of selection sort runtime samples. 43
4.2 Examples of subsets of the data from figure 4.1 with corresponding linear regressions. 46
4.3 Run times of quick sort algorithm (dark dots) and selection sort algorithm (light dots) by input size. 48
4.4 Examples of subsets of the data from figure 4.3 48
4.5 Process of combined function invocation. 56
4.6 Transition diagram of the basic building-block policies. 58
4.7 Transition diagram of the **Start policy**. 58
4.8 Transition diagram of the **Stop selecting when decided** policy. 59
4.9 Transition diagram of the **Pause selection after streak** policy. . 59
4.10 Transition diagram of the **Limited overhead** policy. 60
4.11 Transition diagram of the **Limited gather time** policy. 60
4.12 Syntax diagram of the *policy builder* DSL. 61

5.1 Diagram showing the inheritance chain of AdaptiveFunctionN and related classes. 64
5.2 Diagram showing the internal architecture of the ScalaAdaptive framework. 65
5.3 The original simplified eta-expansion AST. 72
5.4 The simplfied eta-expansion AST after manually adding toAdaptor call. 73
5.5 Diagram showing the HistoryBasedAdaptiveSelector execution path. 78

6.1 Sorting times of sequences of different sizes using selection sort, quick sort and a combined function (from top to bottom). 88
6.2 Detailed view of the sorting times of sequences of different sizes using selection sort, quick sort and a combined function (from top to bottom). 89
6.3 Multiplication times of matrices of different sizes using basic algorithm, Strassen algorithm and a combined function (from the top to the bottom). 91
6.4 Time evolution of response times for request methods in scenarios 1, 2 and 3 (from top to bottom). 95
6.5 Selection overhead times (dark) and policy evaluation overhead times (light), both in ns, for a sequence of 20000 invocations of the same combined function. 103

List of Tables

4.1 Percentage of times the strategy selected the worse one out of two linear functions (average from 100 test runs, each containing 140 selections). . . . 52
4.2 Percentage of times the strategy selected the worse one out of two quadratic functions (average from 100 test runs, each containing 140 selections). . . . 52
4.3 Percentage of times the strategy selected the worse one out of two constant functions (average from 100 test runs, each containing 140 selections). . . . 53
4.4 The number of test runs (out of 100) where the strategy had a success rate worse than 50%. . . . 54
4.5 Average strategy selection time (in ms) in a sequence of 20000 consecutive selections. . . . 55

5.1 The method names extracted from expressions during their conversion to combined functions. . . . 75

6.1 Total times of sorting 500 random sequences of 0-5000 integers. . . . 88
6.2 Total times of multiplying 200 pairs of square matrices of sizes 50-900. . . . 90
6.3 Results of the JSON parsing tests (times in ms). . . . 93
6.4 Results of the JSON parsing tests without the policy applied (times in ms). . . . 93
6.5 Average response times for the request methods in scenarios 1, 2 and 3 (in ms). . . . 95
6.6 Run times (in ms) of Spark queries on RDD and Datasets in various environments. . . . 100
6.7 Average run times (in ms) of Spark queries with different configurations in various environments, and the configurations that ScalaAdaptive selected. . . . 101

List of Abbreviations

- **JVM** – Java Virtual Machine, the target platform of Java, Scala and other languages
- **HTML** – Hyper Text Markup Language
- **API** – Application Programming Interface
- **REST** – Representational State Transfer, a philosophy of web API
- **JSON** – JavaScript Object Notation, a format for data exchange
- **XML** – Extensible Markup Language, a format for data exchange
- **FFT** – Fast Fourier Transform
- **JAR** – Java Archive, a distribution format of JVM based applications and libraries
- **SPL** – Stochastic Performance Logic
- **OS** – Operating System
- **CPU** – Central Processing Unit
- **HDD** – Hard Disk Drive
- **I/O** – Input / Output
- **IoC** – Inversion of Control, an application development pattern
- **AST** – Abstract Syntax Tree, the result of syntax analysis of a code
- **DSL** – Domain Specific Language
- **SBT** – Scala Build Tools
- **RDD** – Remote Distributed Dataset, a Spark API

Attachments

A The attached CD

The disc attached to this thesis contains the following directories:

- `doc` — The documentation of the ScalaAdaptive framework source code automatically generated by the Scaladoc tool.
- `evaluation_data` — Raw output data from the test and evaluation runs that were used in the text.
- `introduction` — A brief user guide that introduces the ScalaAdaptive framework and its usage on a series of examples.
- `jar` — The compiled version of the ScalaAdaptive library in a JAR package.
- `source_framework` — The source code of the ScalaAdaptive framework, can be opened as IntelliJ IDEA project or built from the command-line using SBT. Note that it does not include any test code, as it includes macros and therefore cannot be used from the same compilation unit.
- `source_tests` — The source code of the ScalaAdaptive framework tests and evaluation code, can be opened as IntelliJ IDEA project or built from the command-line using SBT. The package **evaluation** contains all the tests that were used for evaluations in the text. Each test contains a brief description of its inputs, outputs and configuration. The package **tests** contains additional, mostly uncommented, test code. And lastly the package **tutorials** contains examples from the user guide mentioned before.
- `text` — An electronic version of this text.
- `tools` — A simple custom Node.js web server that was used to perform the load balance test.

B Complete list of configuration blocks

The following blocks can be used to configure ScalaAdaptive. If there is not a suitable block, a custom configuration implementation can be done.

Analytics

- `AnalyticsCollection` – turns on the analytics data collection
- `NoAnalyticsCollection` – turns off the analytics data collection

Logging

- `ConsoleLogging` – framework logs to the standard output
- `FileLogging` – framework logs to a specified file (*logFilePath* argument)
- `NoLogging` – framework does not log

Persistence

- `BufferedPersistence` – framework persists the run data in batches of *serializationBufferSize* into a directory (*rootHistoryPath* argument)
- `DirectPersistence` – framework persists the run data into a directory (*rootHistoryPath* argument)
- `NoPersistence` – framework does not persist the data, global storage is used for persistent setup

Selection strategies

Sets the corresponding strategy as the main strategy of given type. Note that the `LowRunAwareStrategy` is always used as a wrapper with configurable *lowRunLimit* argument. Some strategy blocks allow setting *alpha* and *averageWindowSize* arguments.

- `LinearRegressionInputBasedStrategy`
- `LoessInterpolationInputBasedStrategy`
- `TTestMeanBasedStrategy`
- `UTestMeanBasedStrategy`
- `WindowBoundRegressionInputBasedStrategy`
- `WindowBoundTTestInputBasedStrategy`

History

- `CachedGroupHistory` – caches average run times for input descriptors
- `CachedRegressionHistory` – caches linear regression model
- `CachedStatisticsHistory` – caches statistics (used for T-test)