

Sebastian Hungerecker • Malte Schmitz (Eds.)

# Concepts of Programming Languages – CoPL'15

CS 3702 Bachelor Seminar Informatics, CS 3703 Bachelor Seminar Medical Informatics,  
CS 5480 Seminar Software Systems Engineering, CS 5840 Seminar in English  
Lübeck, Germany, winter term 2015  
Proceedings

Institute for Software Engineering and Programming Language  
University of Lübeck

# Preface

In this seminar we explored modern concepts of programming language design and implementation. This encompasses semantic aspects such as paradigms or type systems as well as implementation aspects such as parsing, intermediate representations or optimization.

The aim of this seminar was to give an overview on a selection of topics within the field of programming languages. This seminar does not consist of tutorials for different programming languages, but tries to introduce at least some of the main concepts used in programming languages. The participants could choose a topic that appeared most interesting to them. As a result, there is no clear focus on one specific aspect, but the seminar covers a wide range of exciting topics.

Instead of pre-selected literature the students received at first only a few keywords defining their topic. For example the sub-

ject *evaluation strategies* was described with these four keywords:

- lazy, by need (Haskell, ...)
- strict (most known)
- by name (Scala, ...)
- usage and implementation

This open characterizations lead to much more research effort, but the final contributions give a wider overview on the selected topic and some of the results are quite impressive. The 13 contributions are altogether way above average students' homework.

February 2016

Sebastian Hungerecker  
Malte Schmitz

# Contents

Alexander Schramm		Gilian Henke	
<b>Metaprogramming</b> . . . . .	<b>4</b>	<b>Logic programming</b> . . . . .	<b>44</b>
Merlin Laue		Larissa von Witte	
<b>Basics of Garbage Collection</b> . . . . .	<b>11</b>	<b>Parser</b> . . . . .	<b>49</b>
Malte Skambath		Alexander Harms	
<b>Intermediate Representations</b> . . . . .	<b>16</b>	<b>Array programming</b> . . . . .	<b>54</b>
Toni Schumacher		Thiemo Bucciarelli	
<b>Static vs. Dynamic Typing</b> . . . . .	<b>22</b>	<b>Just in time compilation</b> . . . . .	<b>59</b>
Gunnar Bergmann		Jan Pascal Maas	
<b>Memory models</b> . . . . .	<b>27</b>	<b>Synchronous vs. asynchronous programming</b> . . . . .	<b>65</b>
Timo Luerweg		Jan Niklas Rösch	
<b>Stack based programming</b> . . . . .	<b>33</b>	<b>Aspects of object orientation</b> . . . . .	<b>72</b>
Moritz Flucht			
<b>Evaluation Strategies</b> . . . . .	<b>38</b>		

# Metaprogramming

Alexander Schramm

**Abstract**—The automatic generation of programs has been a research topic for many years in different Contexts. Metaprogramming is one way to write code that generates or manipulates code. This paper will provide an overview of different metaprogramming concepts and their implementation in different programming languages.

For this purpose this paper will show examples of Reflections in Java, Lisps S-Expression and Macros, Template Haskell, C++ Templates and Runtime Metaprogramming in Ruby.

## I. INTRODUCTION

Automatic generation of programs and code has been a long term research field, e.g. to generate test cases, generate software from specifications (UML to Java Classes) or compiler generation. There are different definitions of metaprogramming, based on the language it is implemented in and what should be examined, but for this paper we will use to the following one based on [1]: Metaprogramming is a term describing different ways to write code that generates new code or manipulates existing one. Metaprogramming can be used for language virtualization, type providers, materialization of type class instances, type-level programming, and embedding of external DSLs [2]. While metaprogramming is often seen as an academic approach to programming and is considered bad style in many languages and communities, there are also big projects leveraging its powers and showing how it can be put to great use. Examples we will glance at in this paper are the JUnit test framework from Java and the JBuilder project in Ruby In this Paper multiple implementations and concepts of metaprogramming in different languages will be shown.

*Outline:* In Chapter II there will be an overview of metaprogramming concepts in theory and an explanation of the concept itself. In Chapter III we will look at a first, small example of metaprogramming called Reflection in the Java Programming Language. Chapter IV shows a technique of compile time metaprogramming in the C++ language using Templates and Chapter V will show compile time metaprogramming in a purely functional and type safe manner. Afterwards we will look at runtime metaprogramming in Ruby in Chapter VI and its rich runtime object model. With this knowledge we will look at the richest compile-time metaprogramming model: Makros in Lisp in Chapter VII. In Chapter VIII we will give an overview of further interesting implementations of metaprogramming in other languages and talk about when metaprogramming should or shouldn't be used.

## II. WHAT IS METAPROGRAMMING

The term metaprogramming is used for different ways to generate and manipulate Code. Two big, different approaches

are compile-time and runtime metaprogramming, which are supported in different languages and allow different concepts of code generation. General use cases of all kinds of metaprogramming are to support growth of a programming language (add new syntax without the need to update the compiler/runtime), support of external Domain Specific Languages (DSLs) (e.g. SQL or JSON Fragments) and to write wrapper libraries around other systems with changing interfaces. Furthermore metaprogramming concepts can be used to write more concise code and adapt the style of the language to a specific problem domain [3], [4].

Another differentiation in metaprogramming is the domain- or metalanguage and the host- or object-language [1], which can be different languages or the same. The domain language is the programming language, in which the metaprogram is written, while the generated code is output in the host language. In this paper we will only look at metaprogramming systems, where the domain language is a part of the host language, therefore metaprograms can be embedded in normal programs of the language. An example, where the domain language differs from the host, is Yet Another Compiler Compiler (YACC)[5], which has a special kind of a Backus-Nauer-Form (BNF) as the domain language and which outputs a C++ program [1].

### A. Compile-Time metaprogramming

Metaprogramming, which is happening at compilation time, is supported in many languages like Scala ([2]), C++ ([6], [1], [7]), Haskell ([8]) or maybe most idiomatic in nearly all Lisp dialects in the form of Macros. Most known forms of it are macros and templates, which offer different kind of mechanisms to generate code based on defined patterns which are used by normal code. This patterns can be used to generate multiple, similar methods without writing them explicitly or to transform parts of the program.

Compile time metaprogramming happens before the program runs and therefore the generation of the code doesn't have an impact on runtime performance of the program. This can be used to solve computationally intensive problems during compilation – so only once: when the program is build – and not every time the program is run and therefore to boost the performance of a program.

Compile time metaprogramming gives the user of a programming language a mechanism to interact with the Compiler to allow the construction of arbitrary program fragments from user code. [4] This interface to the compiler can be used to modify the Abstract Syntax Tree (AST) of the compiled code, and rearrange, manipulate, delete or generate new nodes in the tree.

## B. Runtime Metaprogramming

Runtime metaprogramming is all about modifying a program while it is running. This can lead to a very dynamic nature of programs written using these techniques: undefined methods can be generated on the fly, so every user input could be translated into a new method or a program can adapt to changing interfaces of third party services without the need to recompile it. To enable such behavior, the host language has to provide a rich runtime object model. In many classic programming languages, most meaning and structure (read classes, inheritance, methods) of code gets lost, when it is compiled and only sequential instructions to execute are left. Languages which support Runtime metaprogramming like Ruby or Java have to keep some information of the code, e.g. the class hierarchy or where a method is defined, to present this information towards the programmer at runtime. Generating and using an object model produces overhead and slows performance at runtime, but is necessary to provide runtime metaprogramming Application Programming Interfaces (APIs) which can boost performance of the programmers. So runtime metaprogramming is a tradeoff between runtime performance and developer productivity.

### III. JAVA AND RUNTIME REFLECTION

The Java programming language is a compiled one, but it's not compiled into machine code but into a platform independent bytecode, which then can be run in a Java Virtual Machine (JVM) on different host platforms. While there is no compile time metaprogramming API, some parts of the object model are preserved in the compiled code and can be used for runtime metaprogramming, mainly with the reflection API. The entry point for Reflection is the `class` object, which can be retrieved and used as shown in Listing 1.

```
Class<Date> c1 = java.util.Date.class;
// class java.util.Date
Class<?> c2 = new
    ↪ java.util.Date().getClass();
// class java.util.Date
Class<?> c3 =
    ↪ Class.forName("java.util.Date");
// class java.util.Date
```

Listing 1: Using the Java Reflection Api to get the `class` object [9]

`class` objects can be used to retrieve information about the class they represent or objects of that class. They provide methods to get declared fields, methods, constructors and annotations. Annotations are a Java specific way to add more Metadata to language constructs. The popular Java testing framework JUnit<sup>1</sup> makes excessive use of annotations and reflections to run testcases and manage testing environments. Listing 2 shows, how a parser method could get all methods of a class annotated with the `@Test` annotation and call them.

<sup>1</sup><http://junit.org>

```
public void parse(Class<?> clazz) {
    Method[] methods = clazz.getMethods();
    for (Method m : methods) {
        if (m.isAnnotationPresent(Test.class)) {
            m.invoke(null);
        }
    }
}
```

Listing 2: A class parser to find and call methods in a class annotated with a `@Test` annotation

An important distinction is that the `class` object of a class is not the same thing as the declared class, like it is the case in Ruby, shown in Chapter VI, but merely a proxy objects which is used to collect metadata about that class. Therefore it can't be used to manipulate the underlying class by adding new variables or methods. This behavior doesn't satisfy our definition of metaprogramming, nonetheless it's a showcase of an enhanced runtime object model, which provides metaprogramming-like access to the running code.

### IV. TEMPLATE METAPROGRAMMING IN C++

C++ is a compiled, strongly typed programming language, which was developed as a superset of the C language. The Template mechanism is a turing complete [10] subset of the C++ Language. It can be used to generate all kinds of C++ Programs, hence it's turing complete: for an example of a complex implementation using this power see [6]

It's intended purpose when introduced, was to provide a mechanism to define functions (or classes) in a strictly type checked language, which can work with instances of multiple types. Listing 3 defines a template method, which can be called with two values of the same type, but the type can be anything, as long as it provides a comparison operator.

```
template <typename T>
T max(T x, T y)
{
    x < y ? y : x
}
```

Listing 3: Usage of C++ Template mechanism to define a type independent function

While the method mostly looks like a normal function, which could be normally compiled without any metaprogramming facilities, what actually happens is, that the compiler will generate a function for each type it is called with. E.g. there will be no compilation of the method, when it's never called, as soon as it is used with `max("a", "b")` a version of the function with the type `(string, string)→string` will be generated and compiled. When there is another usage of the function like `max(1, 2)` another version will be generated and compiled with the type `(int, int)→int`. All this happens before the program is run, by analyzing the source code for invocations of template functions or classes, followed by generation of the appropriately typed counterparts and finally the compilation of them like normal classes would be compiled.

This differs drastically from the concept of Generics in the Java language, where no code generation is happening at compile time. Java Generics look quite similar to templates, where a method or a class can have arguments of an unspecified type, but aren't nearly as powerful as them. Rather than generating the appropriately typed methods or classes during compilation, every generic class or method is represented by one class or method at runtime, where the generic parameter is given a type based on what is known about it during compilation. A generic method like `public T result() ...` would be compiled similar as the function `public Object result() ...`. In conclusion, Java Generics are syntactic sugar for automatic type casting and can't be used to generate code.

The compile time part of C++ is often referred to as a pure functional language [1] because of a property it shares with languages such as Haskell: (meta) data is immutable and (meta) functions can have no side effects [1], where (meta) data and functions mean the data and functions invoked at compile time to generate code as part of a template and therefore are a part of the metaprogram. While we will come back to a pure functional language, Haskell, we will look more closely at some of the concepts in C++. Abrahams shows an interesting effect of this functional paradigm in [1]: Since you can't write a (non-infinite) loop without examining some mutable state in its termination condition, iteration is simply beyond reach at compile time. Therefore, recursion is idiomatic for C++ metaprograms.

To write recursive templates, one has to use something called template specialization, which takes the role of the termination condition provided in loops. A classic example of this is the computation of fibonacci numbers, where each number is the sum from its two predecessors, so that  $fib(0)=1$ ,  $fib(1)=1$ ,  $fib(n>1)=fib(n-1)+fib(n-2)$ . The simple, incremental implementation, solving the problem at runtime, would have a complexity of  $O(n^2)$  to generate the  $n$ 'th fibonacci number. This could be improved with a template metaprogramming approach using partial template specialization: Because template functions only have to be generated at their first invocation by the compiler, their calculated value will be reused and doesn't have to be computed again. This could also be implemented without templates by using memoization, but with templates we get the benefits without needing to explicitly write the memoization. Listing 4 shows how template structs can be used, where the first part is a normal template and the second part is a specialized template.

```
template <unsigned n, bool done = (n < 2)>
struct fibonacci {
  static unsigned const value =
    I^Ifibonacci<n-1>::value +
    I^Ifibonacci<n-2>::value;
}
template <unsigned n>
struct fibonacci<n, true> {
  static unsigned const value =n;
}
```

Listing 4: C++ template used to calculate fibonacci numbers with template specialization from [1]

The second, specialized struct definition is used by the compiler, when the fibonacci algorithm has reached its termination point, namely when `fibonacci<1>` or `fibonacci<0>` is called, in all other cases the first struct definition is used. Obviously this approach is limited: Because the calls to `fibonacci<n>` have to be fully specified during compilation, also the value of  $n$  has to be known. Therefore one can't use this function to calculate the fibonacci number of a value computed during runtime, e.g. to write a calculator which allows calling the method with user input.

As shown, C++ Templates are a very powerful tool, but as stated in [11] they got this power merely by accident and were never intended to be used as a functional language on their own. This can be seen by looking at the solution of the  $n$ -queens problem in [6]: Previous work solving the same problem were compiler-dependent, used error messages to print out the solutions or the solutions weren't accessible by non-template parts of the program. The implementation in [6] fixes some of these issues, on the other hand it introduces new issues and is complicated to compile or use.

In conclusion, templates can and should be used to design clean code without unnecessary duplication or coupling, but most of the times not to implement something that can be solved at runtime by classic C++ code. This is a characteristic of many metaprogramming techniques, which we will talk about further in chapter VI.

## V. TEMPLATE METAPROGRAMMING IN HASKELL

Templates are an extension to the strongly-typed, purely functional programming language Haskell introduced in [8] and included in the standard Haskell compiler. Similar to C++ templates, templates in Haskell are a compile time metaprogramming mechanism, where code is generated based on given code snippets in a special syntax that is an integrated part of the Haskell language. Different from C++ templates, Haskell templates give the programmer direct access to the AST to modify given code and to generate new one.

The basic syntax of code manipulation is the so called splice operator `$`, which encapsulates a code fragment that is manipulated at compile time. A basic use case for this is the implementation of a `printf` method, which can be used similar to the C method: It takes a string with special placeholder symbols and a variable number of further arguments of a type specified by the placeholder symbols, to interpolate

them into the given string. This can't be done in basic Haskell because of the strict type safety: The type of every argument of a function has to be known at compile time. To solve this, [8] shows a template implementation of the `printf` method, which examines the given string for placeholder symbols and generates a `printf` function with the needed arguments which is inserted at the place of the splice. Listing 5 shows how a splice is used to generate a `printf` function which takes a string and an integer to generate a error message.

```
$(printf "Error: %s on line %d") msg line
```

Listing 5: Usage of a splice to generate a `printf` method with the appropriate types as shown in [8]

When compiled, first the splice is evaluated, which generates the code `(\ s0 -> \ n1 -> "Error: " ++ s0 ++ " on line " ++ integerToString n1)` This code is inserted at the place of the splice, afterwards it will be compiled with the two arguments `msg` and `line` passed to it. This mechanism allows strict type checking of the arguments, because the generated method uses the placeholders inside the string to generate a method which only works with arguments of a matching type for them. As an example one can't use the `%s` placeholder and pass an integer as the first argument, because the generated method would need an argument of type string.

To generate code, there has to be a mechanism to distinguish between code that is executed during splicing, e.g. the `parse` method for the `printf` macro in [8], and code that is inserted and compiled later, namely the code that is returned by the splicing. This distinguishing is achieved with the quasi-quote symbols `[|` and `|]`. Code inside the quasi-quote characters is not evaluated during splicing and can be inserted into the place of the splice. It's important to note, that inside quasi-quoted expressions further splicing can happen, therefore splicing has a recursive character: When a splice is encountered inside a quasi quote it is expanded the same way splices are evaluated outside of them, therefore a splice could generate some code which contains another splice, which would be expanded again.

The quasi-quoting mechanism shows an important characteristic of AST manipulation, that is further explored and used in chapter VII: Code as Data. Inside quasi-quotes, code isn't evaluated, basically it can be treated like any other data, like a string or an integer. Especially data can be generated and manipulated by a Haskell program without the need of a special syntax, because data-manipulation is one of the main use-cases of most programming languages, including Haskell. Similar as one could write a function that return the length of a string, it is also possibly to write a function that returns a new function or manipulates a given one.

As noted in [8] it is a disadvantage, that the programmer at the caller side has to use specific annotations (the splice operator) to use templates, because this separates the host from the domain language. Lisp macros in contrast require no

extra syntax on the caller side, a macro call looks exactly the same as a method call. This removes complexity for beginners, because they don't have to understand the difference between templates (or macros) and normal methods and don't have to learn about metaprogramming at all.

## VI. RUNTIME METAPROGRAMMING IN RUBY

Ruby is a dynamic, interpreted language with a very rich set of runtime metaprogramming APIs. Metaprogramming is such an essential part of the Ruby language, that it blends in with non-metaprogramming parts of it to such a degree, that often times it can't be distinguish whether it is used at all or not. Many parts of Ruby, which look like language level syntax constructs, are actually implemented using metaprogramming techniques.

To allow this mixture of metaprogramming with normal coding Ruby provides a very rich object model at runtime, which can be seen as the counterpart to the AST, which is manipulated by compile time metaprogramming. Basically every part of the Ruby syntax has a runtime counterpart in the object model. This rich object model is not just the API for a programmer to write metaprograms, it is actually used by the Ruby interpreter itself to evaluate code. For example whenever a `class` is defined, at runtime there will be an actual object that represents that specific class. Differently than the class construct of Java we have talked about in chapter III, the class object in Ruby is not just a wrapper around metadata belonging to that class. Quite on the contrary, it is the defined class itself. To understand this, let's look how Ruby implements method calls on objects of a class:

Lets say we have a class `Person` with an instance method called `name` (normally refered to as `Person#name`), and instantiated it with `instantiated_person = Person.new("martin")`. At this point there will be two objects in the object model interesting for us right now: The one which is the instantiated `Person` and the one that is the actual class `Person`. If somebody would call `instantiated_person.name` the Ruby interpreter would look at the object that represents the class of `instantiated_person` and call the method on that class object.

This object model is accessible to the programmer, so one could change it at runtime, which can be used for many metaprogramming techniques to generate or manipulate code. In the next few examples we will see some of the techniques made possible by this accessible, rich object model.

### A. Defining methods dynamically

Ruby offers at least three ways to add a method to an object, from which only one isn't actual a metaprogramming technique. All three methods can be seen in Listing 6.



```

class Person

  def name
    "Hans"
  end

  define_method("greet") {"hola"}

  def method_missing(method, *args, &block)
    if method.to_s.match /^go$/
      "goodbye"
    else
      super
    end
  end
end

```

Listing 6: Different ways to define Methods in Ruby

The method `name` is defined in a normal way, using the `def` syntax, `greet` is defined using `define_method`, which is actually a method on it's own which can be used to define methods dynamically, and then a method named `go` is defined by overwriting the `method_missing` method, which is called on a Ruby object whenever a method can't be found elsewhere on it's ancestor chain. These techniques are explored further in [12] under the names of *dynamic methods* and *ghost methods*. The `define_method` method can also be used to define class methods and has multiple related methods like `remove_method` and `define_singleton_method`.

### B. Altering existing methods

As methods are another part of the object model and therefore accessible by the programmer at runtime, they can be manipulated in different ways. A very common use case is to wrap methods inside other methods to add behavior, for example logging. One technique to achieve this is referred to as an *around alias* in [12]. Basically you rename a given method, redefine it and call the old implementation from within the new method. An important concept of Ruby which are necessary for this technique are *open classes*, meaning that one can expand classes with further methods and fields after they have been defined. Listing 7 shows how this can be implemented to wrap a function from the `String` class by opening it again.

```

class String
  alias_method :old_length, :length

  def length
    old_length > 5 ? "long" : "short"
  end
end

```

Listing 7: Wrapping a given method in an around alias [12]

### C. Defining Classes at runtime

Similar to methods, classes and modules can be generated at runtime. The syntax for this is quite familiar when you think

about how an object is instantiated: `obj = MyClass.new`. To dynamically create a class you would write `MyClass = Class.new do ... end`. This similarity is not by coincidence: As previous stated, classes are actually objects, and as an object they must have a class. This class is the class `Class`, and therefore you can create a class by instantiating an object of the type `Class`.

### D. Usage examples

After diving into some of Rubys metaprogramming techniques, lets look at an actual use cases of metaprogramming in a Ruby library.

We will look at a library called `Jbuilder`<sup>2</sup>, which can be used to generate JSON from Ruby code. In a more strict language, this task could lead to very verbose code that can't easily be recognized as JSON generation, like in a Java implementation<sup>3</sup> shown in Listing 8

```

generator
  .writeStartObject()
  .write("firstName", "John")
  .write("lastName", "Smith")
  .write("age", 25)
  .writeEnd()

```

Listing 8: Usage of the Java `JsonGenerator`

`JBuilder` on the other hand relies heavy on *ghost methods* to enable a very straightforward API that looks more similar to the generated JSON as shown in Listing 9

```

json.firstName "John"
json.lastName "Smith"
json.age 25

```

Listing 9: Usage of the `JBuilder` JSON generator

Even in this small example, the Ruby syntax is more readable and is closer bound to the problem domain. When dealing with more complex JSON data like arrays or objects this gets even more obvious.

As we have seen in this example, metaprogramming is an existential part of the Ruby language and can't be separated from it, often one can't even distinguish between code that uses metaprogramming and that doesn't. Or as it is stated in [3]: "There is no such thing as metaprogramming, It's just programming all the way down."

## VII. LISP MACROS

Lisp describes a family of programming languages with many different implementations, most known are Common Lisp, Scheme and Clojure. The name is an acronym for *List Processing*, which is also a description of it's purpose. Lisp was first specified in 1958 at the MIT in [13]. Lisp has a very simple syntax with a minimum of reserved symbols, where every statement is a so called *S-Expression*. An S-Expressions

<sup>2</sup><https://github.com/rails/jbuilder>

<sup>3</sup><http://docs.oracle.com/javase/7/api/javax/json/stream/JsonGenerator.html>



is formed recursively: Either it is an *atom* or in the form  $(a . b)$ , where  $a$  and  $b$  are S-Expressions themselves. The dot in  $(a . b)$  is left out in most modern S-Expressions notations so that only  $(a b)$  remains.

A Lisp program is now a series of S-Expressions, which can also be interpreted as linked lists. Based on the S-Expressions only a minimal set of operators are needed to provide a Lisp system. These are: one to access the first element of the list, one to access the rest of it, one to concatenate two lists, one to prevent evaluation of a list, one to compare two lists, one for conditional evaluation and a mechanism to define functions.

When encountering a list, Lisps treats the first element of it as a function name and the remaining elements as arguments for that function. Nested lists will be evaluated from the most inner list towards the outermost. Based on the simple structure of Lisps syntax, Lisp code also has a very simple structured AST. This AST can be manipulated by Lisps macro system.

Macros are a special form of functions, that get compiled in two steps: first their arguments get passed to them as unevaluated S-Expressions, then their return value, called their expansion, will get compiled like a normal function. When a macro is encountered, Lisp won't evaluate the, potentially nested, arguments passed towards it, like they would be with a function, but rather will pass them right towards the macro. Let's look at an example: `(print (+ 4 3))` would first evaluate the inner list and pass a 7 to the function `print`. If `print` were a macro, the inner list wouldn't be evaluated but passed to the function as a list which contains data, namely three atoms, and the macro could do with this list as it pleases, for example evaluate it. This is another example of Code as Data, which was already mentioned in Chapter VI: A List can be a normal list of data, which can be manipulated, and it can be legal code at the same time.

To define useful macros, two keywords have to be defined: one to evaluate a list and one to prevent evaluation of a list. For the rest of this Paper we will use `~` to evaluate and `'` to prevent evaluation. Listing 10 defines a macro which takes a list as a parameter, which adds two numbers, and replaces the addition by subtraction.

```
(defmacro toggle (x)
  '(- (hd ~ (tl x)) (hd ~ (tl (tl x))))
)
(toggle (+ 3 4)) # -> -1
(toggle (+ (* 2 3) (3))) # -> 3
```

Listing 10: Defining and using a Lisp macro

The forced evaluation of the arguments of the addition are needed, so that nested lists will be evaluated before their first element is extracted by `hd`, like shown in the second call. After the evaluation of the arguments, a quoted list is build with `-` as the first symbol and the evaluated two arguments as the second and third elements. This quoted list is returned from the macro and will be compiled as a normal S-Expression

Using this macro facility, it is simple to add new functionality to the language in the language itself. As an example

Listing 11 shows a macro implementation of an `unless` condition, which acts similar to an `if` statement.

```
(defmacro unless (condition x y)
  '(if (not ~condition) ~x ~y)
)
(unless (> 1 2) "everything's fine" "something
  → is wrong")
```

Listing 11: Defining an unless condition using a macro

One couldn't implement `unless` as a normal function, because the arguments that should be evaluated based on the condition, would be evaluated when the function is called and their return value would be passed to the `unless` function.

## VIII. CONCLUSION

We have seen many different techniques of metaprogramming and methods of using them. It's obvious that the term can't be used for only one thing but rather references a whole family of programming paradigms. To complicate the matter even more, there are many more ways of metaprogramming and even more implementations of them.

The Groovy programming language<sup>4</sup> offers runtime and compile-time metaprogramming at the same time, which makes it difficult to separate between those. For Scala<sup>5</sup> there has been a recent implementation of type-safe, Lisp like macros [2], which introduces different types of macros for different language constructs. The fairly new Elixir programming language<sup>6</sup> combines many concepts of Ruby with a Lisp like macro system and an interesting Code as Data approach, where every statement is represented by a tuple which can be modified like lists in Lisp.

The question remains when to use metaprogramming. Primarily it depends on the language, e.g. Java's runtime reflections come at the cost of performance<sup>7</sup> and the C++ template system is often a fragile and hard to understand mechanism to implement complex algorithms. On the other hand in languages like Ruby and Lisp, metaprogramming is deeply entrenched and is often the goto approach. Specifically compile-time metaprogramming can be used to get better runtime performance without the need to write repetitive code. So one should use offered metaprogramming techniques mostly in the way they were intended: C++ templates shouldn't be used to solve complex algorithms but to write concise, type independent code where possible. Java reflections can be used whenever performance is no primary concern. In Ruby and Lisp metaprogramming is an essential part of the language and should be used whenever it offers an advantage over other implementations.

Metaprogramming can be a dangerous tool and hard to understand, therefore other ways of implementation should be evaluated every time a metaprogramming approach is

<sup>4</sup><http://www.groovy-lang.org>

<sup>5</sup><http://www.scala-lang.org>

<sup>6</sup><http://elixir-lang.org>

<sup>7</sup><http://docs.oracle.com/javase/tutorial/reflect/>

considered. But as different implementations should always be considered, metaprogramming should just become another tool for a programmer to use.

#### ACRONYMS

API	Application Programming Interface.	2, 4, 5
AST	Abstract Syntax Tree.	1, 3, 4, 6
BNF	Backus-Nauer-Form.	1
DSL	Domain Specific Language.	1
JVM	Java Virtual Machine.	2
YACC	Yet Another Compiler Compiler.	1

#### REFERENCES

- [1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, 2004.
- [2] E. Burmako, “Scala Macros : Let Our Powers Combine ! On How Rich Syntax and Static Types Work with Metaprogramming,” 2013.
- [3] P. Perrotta, *Metaprogramming Ruby: Program Like the Ruby Pros*, 2010. [Online]. Available: <http://book.douban.com/subject/4086938/>
- [4] L. Tratt, “Compile-time meta-programming in a dynamically typed OO language.” 2005. [Online]. Available: <http://eprints.mdx.ac.uk/5919/>
- [5] S. C. Johnson, “Yacc : Yet Another Compiler-Compiler,” in *Computing Science Technical Report No. 32*, 1975, p. 33.
- [6] D. V. Dubrov, “N Queens Problem : a Metaprogramming Stress Test for the Compiler,” pp. 35–45.
- [7] a. Alexandrescu, *Modern C++ Design*, 2001.
- [8] T. Sheard and S. P. Jones, “Template Meta-programming for Haskell,” 2002.
- [9] C. Ullenboom, *Java ist auch eine Insel - 13 Einführung in Datenstrukturen und Algorithmen*, 2011.
- [10] T. L. Veldhuizen, “C++ Templates are Turing Complete,” pp. 1–3, 2003.
- [11] A. D. Robison, “Impact of economics on compiler optimization,” *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande - JGI '01*, pp. 1–10, 2001. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=376656.376751>
- [12] G. Sebastian and M. Fischer, *Metaprogramming in Ruby A Pattern Catalog*, 2010.
- [13] J. McCarthy, “Recursive functions symbolic expressions and their computation by machine, Part I,” *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

# Basics of Garbage Collection

Merlin Laue

**Abstract**—Garbage collection is a widely appreciated functionality which is delivered by a lot of modern programming languages and compilers. This paper will explore the mechanics behind garbage collection and evaluate the pros and cons associated with all of the named collectors. We not only evaluate the inherent time space trade-offs which can decrease the performance of a collector by 70%, but also reveal that the number of collection cycles is not the most important reason to look at when trying to figure out the performance. Further onwards the native problem of fragmentation and determinism is presented. We analyse all the details behind the mechanics of a garbage collection algorithm and observe how each collector deals with these issues. Sometimes simple solutions can result in massive increases in performance reaching up to 800% over brute force, just by changing to a Merlin-type tracing method. Even more specific issues like cycles in the memory and locality of the mutator, have and can mostly be dealt with in a quite simple fashion as well. The basic algorithms we present include Semi-space, Reference counting, Mark-and-Sweep, and Mark-and-Compact. Alongside, we will also present a more advanced generational collector which is equivalent but not exactly the same for each basic collector type. Lastly, vastly different collectors which oppose the normal exact type collector, the conservative collectors, will be introduced and analysed. After reading this compilation one has the basic knowledge to determine if the use of a garbage collector and which kind of collector is appropriate.

## I. INTRODUCTION

In modern times everything is made easier. Programming follows this trend, as most programmers choose to use an environment that supports automatic memory management (Garbage collection). The previous `free()` and `malloc()`-methods usually require quite a lot of attention from the programmer, lengthening the development process considerably. They can restrict the programmer from some modularity features and have a tendency to create memory leaks if one is not careful enough with allocation and freeing. With this in mind having the means to automatically collect dead objects is very convenient. Despite the convincing features, the question whether you should use a garbage collector is not a simple one to answer. Garbage collectors tend to require a lot of memory to do their duty. Ranging from a decrease of 17% at three times relative memory to a decrease of 70% at two times relative memory [8], an unoptimised algorithm can cause a lot of trouble in memory scarce environments. Performance is, against common belief, not as much dependent on the amount of collection but more on the locality of the mutator<sup>1</sup>. Having a very compact space of data to work with is very important. This leads to the next topic of fragmentation. Holes in the memory combined with a bad allocation method lead to easily recognizable performance

<sup>1</sup>A mutator is part of a running program which executes application code. Its name is based on the fact that from the collector's point of view it mutates the graph of objects.

drops. The most simple garbage collection algorithms are Semi-space, Reference counting, Mark-and-Sweep. Using these algorithms, said issues can be observed in a simple fashion. Semi-space without compacting naturally causes fragmentation because it only copies one space of memory to the other and then switches the spaces. Reference counting only counts references to each object with a free-list and then sweeps zero count objects. It does not compact the memory and thus leaves holes in the memory which are then filled with a first fit strategy. These are only some examples of causes for fragmentation. To fix a lot of these issues the generational collectors were invented. They partition the memory in different 'age'-spaces and follow the weak generational hypothesis [1]. Because the young generation contains fast decaying objects it is collected faster and surviving objects are then moved to the adult and later permanent spaces. The spaces use one of the basic strategies (MS,RC,Semi-space) to lazily collect dead objects.

Said algorithms are exact collectors requiring assistance by the compiler and runtime. In contrast, there also exist less elaborate conservative collectors. They examine the memory and most notably the execution stack to check for every word that looks like a reference to an object. In case such a word exists, they then make the *conservative* assumption that it is indeed a reference. The term for such a reference is ambiguous reference. Lastly, the algorithm proceeds to filter and pin them. Pinning can create memory overhead because of excess retention.

Onwards this paper will explain all details behind the algorithms and give a deeper insight into the complexity of garbage collection and its performance.

## II. PROBLEMS

Garbage collection frees programmers from memory management, prevents most memory leaks and adds improved modularity. However this convenience comes with a fairly heavy price to pay in most cases. This section will provide you with reasons why, in memory scarce or specialized environments a programmer is better off using manual memory management.

**Time Efficiency** Time efficiency relates to the amount of time the garbage collector needs to record and collect all of the unused data. Depending on the type of collection algorithm and allocated memory size this can lead to a drastic performance degradation. For example an appel-style generational collector [1] with a noncopying mature space is just as fast as explicit management when having five times its memory. At three times this degrades to 83% and when only given twice the memory this degrades to 30% on average. [8] This is because at low heap sizes the collection frequency usually dominates the locality benefit of contiguous allocation. As heap size increases the mutator locality advantage of contiguous allocation starts

to take effect resulting in faster execution time. [5] Contiguous allocation also provides fewer misses at all levels of the cache hierarchy.

**Memory Usage** As stated above garbage collection tends to have a close affinity to the amount of memory allocated towards it. When examining garbage collection algorithms you come across some common major obstacles which bring up the memory usage. If the algorithm uses a table or a list to note the current references and/or time stamps then this has to be stored in the memory. Furthermore some algorithms store data about the state, location and size of the object by adding additional overhead to each object. Having write barriers to limit the collection sizes adds overhead to objects as well. In some cases the overhead can be stored inside the actual object but that is a very rare case. Lastly if you happen to use a copying garbage collector then it needs space to copy the current set of objects to. Equally important, but not as easily observable, is the use of memory because of the timings the garbage collector has preset. Late finalization will cause more memory usage in comparison to an early finalization. This is accommodated by the three terms SyncEarly, SyncLate, and SyncMid. [6]

**Fragmentation** Alongside memory usage garbage collection algorithms can become prone to fragmentation. If there is no implemented compactor component, then just marking and collecting loose objects in the memory leads to a lot of small gaps inside the memory. Given the fact that these holes are too small to be filled in again the effective memory you have for the program to execute gets smaller by the time. Thus garbage collectors have to collect more often resulting in up to exponentially increasing execution times. More advanced algorithms counter this problem by allocating objects into generational areas which are then compacted as they are promoted to move into the area.

**Determinism** Determinism refers to the topic of finalization. To elaborate, finalization is a method which the garbage collector calls when it wants to free an object. In contrast to destructors, a finalization method has no implicit time stamp on which it actually frees the object. Thus creating an unknown factor in program execution making it possible for the program to not be deterministic. This becomes an issue if the object uses shared resources because it leads to unpredictable waiting times in between the objects who use the resource.

### III. ALGORITHMS

Before analysing the mechanics behind each algorithm we have to define certain phrases which become necessary. Firstly, there are 3 kinds of garbage collectors : Stop-the-world, Incremental, and parallel (Realtime versions are also available). Stop-the-world is the most naive type. To collect objects and to trace the heap the algorithm needs the execution to stop completely. This is to avoid mutations in the object graph. In contrast to stop-the-world, an incremental type collector collects garbage and analyses the heap bit by bit without having to stop the program for too long making it potentially faster. The probably most optimal type of collector works parallel with

the execution. The pros and cons of each type will become relevant when further explaining the algorithms.

#### A. Semi Space

The semi space collectors use two equal sized copy spaces. It contiguously allocates into one and reserves the other space for copying into since, in the worst case, all objects could survive. If one half happens to run out of memory it copies live objects into the complementary half and proceeds to swap both regions. Thus the collection time is proportional to the number of survivors inhabiting the full copy space. The performance of such an algorithm suffers due to the large amount of memory required to set up the two spaces and the fact that semi space collects the entire heap every time.

Copying tracing proceeds the transitive closure in the following way. Firstly it enqueues the location of all root references and then iterates along the roots. If the referent object is uncopied, it copies it, leaves a forwarding address in the old object, enqueues the copied object in another 'gray queue' and then adjusts the reference to point to the new object. [5] Semi space does not use a write barrier to separate the 2 copy spaces.

#### B. Reference Counting - RC

RC algorithms use a free list to note the amount of references pointing towards any given object. During mutation, the write barrier ignores all operations that store into the roots and only logs the mutated objects. Afterwards it consults the list and periodically generates reference increases and decreases for root referents. Objects with a zero count are being freed and all the objects it references are being recursively decreased. The collection time is proportional to the amount of dead objects. Due to the mutator having to log all changes to the free list, the burden on it is significantly higher than normally. Besides the significantly increased workload, RC also has big problem with reference cycles in the memory. The most simple example of a cycle is  $A \rightarrow B \rightarrow C \rightarrow A$ . In this scenario (and without proper cycle detection) A could never be collected if it loses all outside references because C always references A. Thus most standard RC algorithms feature trial deletion as a common method to detect cycles. [2]

RC only really records objects upon its first modification. Afterwards it buffers the in- or decrements using the logs in the list. When needed to collect dead objects it firstly generates increments for all roots and referents, then secondly introduces temporary increments [3] for roots and the objects referred to, and lastly deletes objects with a reference count of zero. If an object becomes a zero count during this process, its added to the free list by setting a bit and decrementing recursively as described above. These objects will be collected in the next collection cycle.

#### C. Mark and Sweep - MS

There are a lot of different MS variants with a wide variety of efficiencies. The following will only cover the basics. In contrast to RC, MS uses a free list and a tracing collector. Every time when the heap is full, it triggers a collection using

tracers to mark live objects with bitmaps. Allocation works by simply finding slots with a first fit strategy. Tracing is proportional to the number of live objects while reclamation is proportional and scaling with allocation. The most basic tracing works the same way as Semi Space. The only difference is that instead of copying the object it sets a bit in a live object map. For very efficient tracing, the Merlin algorithm is available. It can increase the performance of tracing by a factor of 800 times over brute-force tracing. [6] The downside of this collector is it being a whole heap collector, meaning it will collect the whole heap in every cycle. Another fairly big problem is the speed in which the memory fragments itself.

The free-list uses slotted-fits with different size classes. The basic classes are 1)4 Bytes apart 2)8 Bytes apart 3)16 Bytes apart 4)32 Bytes apart 5)256 Bytes apart and 6) 1024 Bytes apart (Classes may vary. These classes are for MMTk). A possible optimisation is the use of a best fit strategy instead of bump pointers to reduce fragmentation, but this will result in a 10% degradation of speed. [5]

#### D. Mark and Compact

Essentially this is a mark and sweep algorithm but instead of leaving the memory in a state of fragmentation, it compacts the loose sections of memory in one localised strip. The first step is always the marking of alive objects to add them to a free-list. Just like a normal mark and sweep this triggers in case of a full heap. In contrast to a normal sweeping phase the compacting and second phase is vastly different. Instead of just sweeping through the whole heap, each block of memory gets moved to one side. First it computes the forwarding addresses for each object, then it updates the pointers to point to the new region which the referents are moved into and finally relocates every object into its new space. [13] Mark and compact has greatly decreased allocation times due to the locality of all used objects, making it easy to just allocate in the free memory zones. Additionally it does not need twice the heap size like normal semi space algorithms and is freed of fragmentation. Although this seems to be very efficient, the disadvantages outweigh the advantages in most cases. The overhead needed to store the forwarding pointers is far too big. Adding the fact that it also needs 3 runs for each time the algorithm tries to compact the memory, mark and compact is uncompetitive against any optimised garbage collection algorithm. [9]

#### E. Generational

The most basic idea of a generational garbage collector is the partitioning of the memory in different 'age' sectors using write barriers. The young (or nursery) space contains objects that were just created and have not been processed so far. The nursery has different policies regarding its size. A flexible-sized nursery consumes nearly all the usable heap space and gets reduced by collecting dead objects. A fixed nursery on the other hand will never expand beyond its fixed sizes making the collection times very foreseeable. Lastly, a

bounded space uses an upper and a lower restriction. In case of a full memory space, the collector reduces the nursery size until the lower bound is reached. Only if the restriction is reached and the heap is full, it will collect the whole heap. Surviving objects are then promoted to an old (or mature) and lastly into a permanent space. Assuming, the young generation has to be collected quite often because young objects die very quickly while the older generation have a much higher rate of survival (Weak generational hypothesis [1]), one comes to the conclusion that generational collectors have a much higher efficiency than other collectors. Additionally this provides the possibility to only collect the entire heap when the mature region is full and enforces a collection. A basic generational collector is always incremental.

Considering the partitioning of the memory a generational collector is usually affiliated to one of the other algorithm types depending on the collection strategy/-ies they include.

**GenCopy** is using a semi-space type algorithm to collect objects in the different sectors. In contrast to normal semi-space, GenCopy attains much better performance due to the increased rate of collection in the young generation and decreased rate of collection in the old generation. This yields a lot of free memory for the allocation of new objects. Moreover it compacts the other generations to improve the locality of the mutator. Since the nursery is smaller than the entire heap the collection time for it is also vastly reduced.

**GenRC** is a hybrid of bump pointer allocation and copying collection in the nursery, while the mature generation uses a free-list allocation and reference counting collector. This works with all kinds of nursery size policies. The mechanics behind the nursery handling (copying, write barriers etc.) are the same as the GenCopy nursery mechanics. A collection is triggered every time the nursery runs out of memory. The first phase is root scanning, processing of modified objects, tracing the nursery and then integrating nursery survivors into the RC space. Afterwards the normal reference counting method is applied to all objects. [10] As to be expected, the GenRC is a lot more efficient compared to the normal RC. This is based on fact that it is able to ignore a lot of the frequent mutations of nursery objects. However GenRC still has the same fragmentation problems, although in the generational form it can negate this effect a little bit by lazily repetitive collections in the mature space.

**GenMS** is, in the same way GenRC is a hybrid between normal RC and a copying space, a hybrid between copying nursery combined with a Mark and Sweep algorithm for the mature generation. Policies regarding the nursery, write barriers and all other aspects are equivalent to GenCopy and normal Mark and Sweep. Theoretically, GenMS should have a better performance than the normal MS, but this is heavily dependent on the state of fragmentation of the memory caused by the algorithms.

To compare and summarise all algorithms we want to introduce figure 1. We can see that Semi Space starts off with the worst execution time when combined with a very small heap. This shows how the need for two big copy spaces

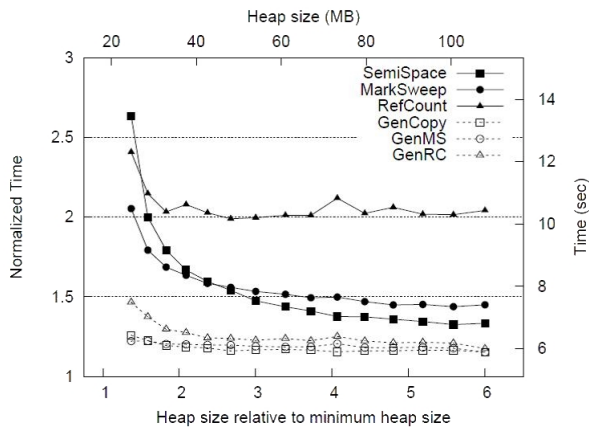


Figure 1. Comparison between all algorithms using a Pentium 4 system. Image from Myths and Realities: The Performance Impact of Garbage Collection [5]

influences the execution time. RC and MS are a bit faster because they do not require such a large copy space. The generational algorithms are always faster than the normal variants. Following the curves we can examine that normal RC has by far the worst performance, even when increasing the heap size. This trend is caused by the heavy burden on the mutator and the need to keep the free-list. The free list also causes fragmentation and thus has bad locality. Moving along this is also visible when examining GenRC. It inherently has the worst performance of all generational collectors, although the generational variant is very close to all other generational algorithms. In contrast to RC, Semi-space becomes far more efficient the more you increase the heap size. The larger the copy spaces the less it has to collect and copy, making it even more efficient than MS upon crossing the 2.5 times heap barrier. Since GenMS is vastly improved by the fact that this GenMS compacts its memory, its not only faster than normal MS but also equally as fast as GenCopy.

At this stage we also have to point out the importance of the type of processor that has been used. Garbage collection is also dependent on cache misses and level of cache associativity. If we were to observe this test on an equivalent athlon processor we would see an 20% increase in performance due to the larger L1 cache and the higher associativity at level 2. Although using an athlon processor increases the performance, this does only influence the way the curves look by a little bit. The graph for an athlon processor is basically equivalent of the one presented in figure 1. Only the curves have to be lowered to match the faster execution times. [5]

#### IV. CONSERVATIVE VS. EXACT

So far we heard a lot about algorithms. All these belong to the class of exact garbage collection. This method of collection is defined as one which requires cooperation from the compiler and language runtime. [11] The assistance from the compiler does not have to be inherent though, creating a second class of exact collection named uncooperative exact. Uncooperative exact collectors can be found in strongly typed systems which

are mostly implemented with C as the designated language. Using the stock C compiler means that the runtime has to create a shadow stack. This is a separate data structure which identifies the number of alive objects for each frame. Consequently making it not directly compiler assisted, but semi (or forcibly) assisted instead, hence its name uncooperative.

#### Conservative Collection

Conservative garbage collection is a less elaborate and early variant of garbage collection. This does not mean conservative collectors are very simple, but they wont require as much assistance as normal exact collector would. This type of collector works by 'guessing' the references in the memory (most notably the execution stack). It scans the whole storage for words and if it finds a word fitting the style of a reference it will make the *conservative* assumption that it is a valid address referencing an object meaning it should not be collected. The special term for these references is *ambiguous reference*. While working under the assumption that it has found a valid reference there are 3 major constraints the collector now has to work with.

- Because the ambiguous reference might be a normal value the collector cannot move or modify the referents, exclusively *pinning* them .
- To prevent corruption the collector needs some sort of mechanism to *filter* all references.
- In case the collector finds a valid reference it has to retain all referents meaning it has to incur *excess retention*.

All three of the constraints lead to some sort of drawback which burdens the collector and/or lead to extended memory usage.

**Pinning** will incur fragmentation of memory since the algorithm is not able to move referents. In some safe languages without real manual memory management, the algorithm only has to pin objects targeted by ambiguous roots. In unsafe languages on the other hand, all references are ambiguous regardless of whether in the runtime or heap and must be pinned. This means in unsafe languages the performance decrease will be quite significant.

**Filtering** means the algorithm will separate and delete all references not referring to actual objects. The reasons why such words exist in the stack are either just the sheer mishap that a program value is identical to a memory address, the existence of old temporary pointers the compiler once used, or references remain in the stack for far beyond their lifetime. Filtering mechanics depend on the type of conservative collector in use.

**Excess retention** is a space overhead caused by the collector. Since it is a possibility to mark some dead objects as alive some objects cannot be collected. This means more memory has to be used in order to store the false alive object. Even though this seems to be a fairly big issue, remembering how space effected exact garbage collectors, it only adds about 0.02% to 6% memory overhead in a Java. [11]

All conservative algorithms use one of the basic principles that were mentioned earlier.

Semi space has a counterpart called MCC, a mostly copying conservative collector. MCC<sup>2</sup> is very similar to normal semi space in that it uses two copy spaces and all the other basic mechanics III-A. The only differences exist in the continuousness of the two copy spaces and the use of pages. The two copy spaces are only logical spaces consisting of linked-lists of discontinuous pages. Assuming, the pages are alive and referenced by an ambiguous root, they are then logically promoted into the other free list by unlinking them from one list and linking them into the other.

Another big counterpart is the the Boehm-Demers-Weiser (BDW) [7], which is the equivalent of an exact mark and sweep algorithm. It uses a free-list allocator and mark-sweep method to collect dead objects. When it comes across an ambiguous reference it checks the free-list block the reference is pointing towards and creates a size class for the cells in that block. Then it checks if it is the *start* of an alive cell inside the block. Only references pointing towards the start are actually treated as valid. The rest of the algorithm then follows the standard mark and sweep procedure III-C.

To compare and evaluate both methods we will now have a look at the performance of conservative garbage collection. When only observing the text, you could come to the conclusion that conservative algorithms might be faster than other exact algorithms of the same type. MCC for example only swaps entries in free-lists after all. Sadly, the performance of such conservative garbage collectors is usually worse than the performance of exact garbage collectors. In case of a 2 times minimum heap size MCC will be 9.3% slower than a normal semi-space collector. This is mainly because of the pinning which induces heavy fragmentation resulting in far more collection cycles and bad mutator locality. MCC is genuinely the worst out of all conservative collectors. Comparisons between Mark-and-sweep and BDW result in a far better outcome for the conservative collectors. The BDW is only 1% slower and even features a smaller overhead. The problems are located in the object map and allocation time costs.

As a brief digression I want to mention the newest generation of garbage collectors using Immix [9]. The conservative Immix versions are usually only 0.02% slower than their exact counterparts. Recent studies have also resulted in a collector called RCImmix [12] which exceeds the previously fastest Immix version, GenImmix. Surprisingly, the conservative version of RCImmix, while slower than the exact version, also exceeds GenImmix even if only by 0.01%. This shows that conservative collectors gradually become faster and can perform nearly as good as exact collectors.

## V. CONCLUSION

Garbage collection offers a wide variety of solutions making the life of a programmer easier, but also delivers some severe problems. Complementary to the idea that number of collection cycles is dominant, this paper shows that locality of the mutator

is a bigger factor. This results in the issues of fragmentation, memory usage, and efficiency being very closely tied together. Additionally, more elaborate problems like cycles in the memory arise. All basic algorithms have some way to deal with them. Simple modifications, e.g. with utilising a Merlin-style algorithm, increase the performance drastically. Nevertheless, there are limits to all the improvements. Generational garbage collectors outperform their counterparts because of the mostly fixed fragmentation and cycle intensity. Equally important are the timings of garbage collectors. They may either stop the program completely (Stop-the-world), optimise in collecting partially (incremental), or run parallel to the execution. In contrast to the exact compiler supported algorithms, the conservative algorithms are presented. They run by guessing the references (ambiguous references) in memory and execution stack. In essentially the same way exact collection has problems, this type of collector has to deal with pinning, filtering, and excess retention. To conclude, the new generation Immix-type collectors are presented as a way to show that more advanced research creates algorithms which push the performance of conservative and exact collectors to nearly the same values.

## REFERENCES

- [1] A. W. Appel. Simple generational garbage collection and fast allocation, 1998.
- [2] D. F. Bacon, P. Cheng, and V. Rajan. A unified theory of garbage collection, 2004.
- [3] D. F. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems, 2001.
- [4] J. F. Bartlett. Compacting garbage collection with ambiguous roots, 1988.
- [5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection, 2004.
- [6] S. M. Blackburn, M. Hertz, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Generating object lifetime traces with merlin, 2006.
- [7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment, 1988.
- [8] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management, 2005.
- [9] K. S. McKinley and S. M. Blackburn. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance, 1998.
- [10] K. S. McKinley and S. M. Blackburn. Ulterior reference counting: Fast garbage collection without a long wait, 2003.
- [11] K. S. McKinley, S. M. Blackburn, and R. Shahriyar. Fast conservative garbage collection, 2014.
- [12] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting immix, 2013.
- [13] P. Thomas. Incremental parallel garbage collection, 2010.

<sup>2</sup>We will be using a Bartlett style MCC [4] since it is one of the most popular among the mostly copying collectors.



# Intermediate Representations

Malte Skambath

malte.skambath@student.uni-luebeck.de

## I. INTRODUCTION

Developing software in general has the goal to use and run it on real machines or share it with other people who want to use this software on their hardware. As real machines only understand one special machine code based on their hardware-architecture (for example Intel's IA64 CPU-Architecture) programs in a high-level language need to be translated into this machine language.

Obviously we need such kind of compilers for this translation between programming languages and the hardware. But whenever it is possible that our software has to run on different architectures, this requires several compilers.

However developing a whole compiler for each combination of a programming language and hardware-architecture would require much effort and is unpleasant work as the single steps are nearly always the same. Solutions for this problem have already been developed and can be found in practical use. Those solutions, like virtual machines, are abstracting parts of the entire compilation process. For this we need hardware or platform independent abstractions or data structures for programs. Such data structure are called *intermediate representations* (IR) and need to be powerful enough to describe the semantics of a program but also need to be small and simple to be easily translatable into machine code.

It is possible just to use simple programming languages like C as an intermediate representation and use compiler of this language. Nevertheless special languages have been designed to be simpler and allowing more optimizations during all compilation steps.

In this handout we give a short overview about the common types of intermediate representations. At the beginning we give a short and simplified overview about the classical process of compiling. After this we introduce into common machine models and types of intermediate representations used for abstraction in most applications. Finally we present two different implementations of such intermediate representations in section IV.

## II. CLASSICAL COMPILE PROCESS

Most classical compilers use about three phases (see Fig. 1) for generating the machine-code for certain hardware out of a given program written in high-level programming language. At the beginning, in the *frontend*-phase, a sequence of tokens, like numbers, strings, or keywords get extracted from the source code. Using this sequence a *parse tree* which represents the syntactic structure of the code will be generated. When a parse tree gets simplified or annotated with special informations it is called an *abstract syntax tree (AST)*. Figure 2 gives

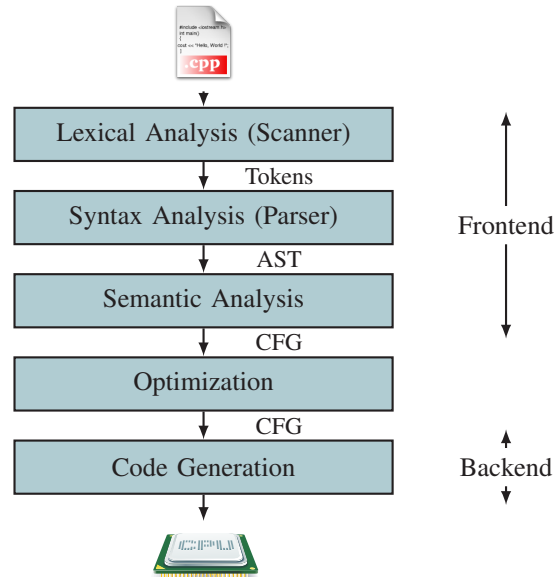


Figure 1: The Architecture of a classical Three-Phase Compiler. In the frontend the control flow of the program will be extracted without dependencies to the target architecture and in the backend the machine-code will be generated for the target architecture.

an example for such a tree. Those trees are often used in the semantical analysis of the compiling process in which syntactical informations like variable declarations are checked and the control flow graph can be generated. Sometimes they can be converted partially into directed acyclic graphs for detecting redundant use of variables or computations.

The *control flow graph* (Fig. 3) is a directed graph representing all possible paths a program can follow during its execution. Each node contains instructions or is a condition with two outgoing edges. It can be possible to optimize the CFG and finally it can be translated directly into a given assembly or machine language in the last phase of the compiling process. It should be mentioned that already this data structure is kind of an intermediate representation for internal use in a compiler[5], [14].

## III. MACHINE MODELS

As already mentioned most compilers and especially platform-independent implementations use intermediate representations to represent the control flow graph of a program. Out of this flow they generate machine dependent code in

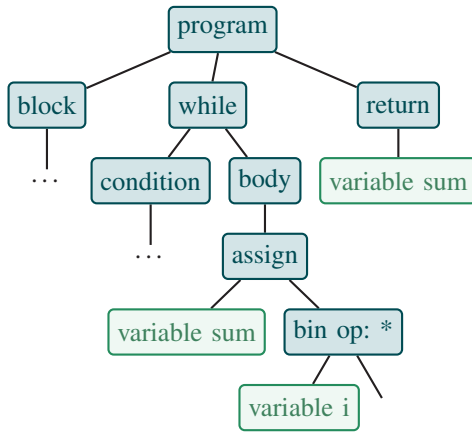


Figure 2: An example for an abstract syntax tree representing the structure of a simple program.

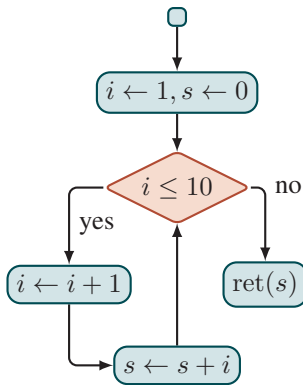


Figure 3: An example for a simple control flow graph to compute the sum  $\sum_{i=1}^{10} i$  in a loop.

the backend-phase. Thus it is necessary to have a very low-level machine abstraction for this purpose to keep this process simple. This means we need an abstract or *virtual* machine model to have the ability to use assembly-like representation for the control flow which is still flexible enough to be translatable into different real assembly codes. Such a human-readable assembly-language is called *intermediate language*. In real implementations equivalent byte-code-formats with a one-to-one mapping to human-readable assembly languages are used because this prevents additional needless parsing processes between consecutive phases[14].

The differences between assembler codes and the intermediate representations are missing restrictions on hardware like the size or number of available registers or that IR could pass additional informations that may be important for later executed machine-dependent optimizations. This means that hardware-dependent optimization like register-allocation can be separately implemented in the backend.

In practice there exists two different types of such machine models: stack- and register-based machines. In the following models memory or input-output will not be considered though real implementations as we will see later in section IV need

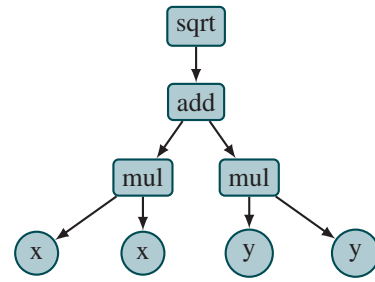


Figure 4: The abstract syntax tree for the expression  $\sqrt{x^2 + y^2}$ .

to handle memory and input.output-access.

#### A. Stack Machines

A stack machine consists of a pushdown-stack and an accumulator or *arithmetic logic unit* (ALU) which is connected with the top elements of this stack. It can put on or remove values from top of its stack and perform operations using the accumulator.

When performing operations like addition or multiplication the top values are taken from the stack, given to the accumulator, and the result is pushed back as new value on top of the stack. This means that for any computation on a stack machine an operations that has to be computed follows pushing or precomputing the values it depends on.

Programs for stack machines are sequences of instructions from a very reduced instruction-set including push and pop operations, some arithmetical functions and also branch instructions to allow non-sequential control flows. So for computations a program has to be written down in reverse polish notation.

While this might be an unfamiliar notation for humans this is a big advantage evaluating expressions given the abstract syntax tree. Because of the operation-nodes in a syntax tree are mostly root nodes for the subtrees of their dependent values the program for a stack machine can directly be generated by traversing the syntax tree in post order. For example computing the euclidean distance for which the expression is presented by the tree in Fig. 4 we would get the following sequence of instruction:

```

1 push x      ; push the value x
  push x      ; push the value x
  mul         ; x^2
6 push y      ; push the value y
  push y      ; push the value y
  mul         ; y^2
add          ; compute x^2+y^2
sqrt

```

Here one can see an advantage of stack based machine models. While register based models always need to reference the involved registers an for stack machines only the operations has to be referenced as the values are on top of the stack. So

for stack machines we can write shorter programs for the same computation steps because we don't have to reference target addresses as results are automatically pushed on top of the stack[13].

Programs for a stack machines could be extended with subroutines. When we compute something on the stack with the method we can see that the state of the stack only changes on top where all used values are replaced with the result of a subroutine and there is no reason why the rest of the stack should be touched. This makes it easy to perform subroutines on the same stack. Mostly the calling routine pushes the input variables on the stack before the call.

A big disadvantage of the stack-based model is that most real machines are register machines so the stack-based code has to be translated into register based code and this could mean that special optimization for register machines like register-allocation has to be done in a different step. Some implementations allow use temporal registers or operations on more than the only top values in the stack for this problem or also share special value like the maximum stack-size.

### B. Register Machine

A register-based machine has an arbitrary number of memory cells called *registers* that can store values. Operations, like addition, can be done on each (pair) of registers and the value will be stored in another defined register and computations only can be done using registers also if other memories as in real implementations are possible. Each register is available the whole runtime for operations. An Instruction for a register machine can be

- an assignment ( $r1 := r2$ )
- an unconditional jump (L1: **goto** L21)
- a conditional branch (L21: **if**  $r1 \geq 10$  **goto** L42)
- an arithmetical expression with an assignment ( $t1 := 2 * r2$ )

1) *Three-Address Code (3AC)*: Because arithmetical expressions can become complex it is a bad idea to allow any complexity or use parenthesis for arithmetical expressions. To prevent problems and simultaneously give the ability for an unique and simple byte-code mapping only one operation with at most two operands is allowed per assignment. Then this register-based assembly language is called *three-address code* (3AC or TAC) because each instruction can be related to at most three registers.

This means we can represent each instruction for example as quadruple like (opcode, regDest, regOp1, regOp2) in byte code. There are much more possibilities for example using triples like (opcode, regOp1, regOp2) where register-address for the result is chosen implicitly by the line number. Note that for such a case a register only may be assigned once which is a strong restriction to the power of a TAC as we will see later.

The following Listing gives us an example of a three-address code for computing  $x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ :

```
t1 := b * b
```

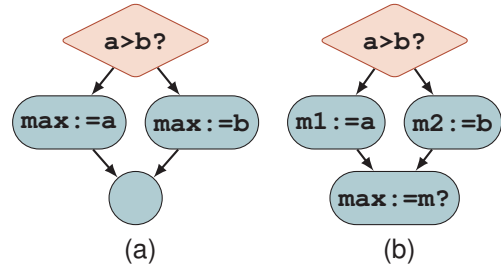


Figure 5: A Branch in the control flow with assignments to the same variable *max* (5a). Assignments in single paths to the same variable could be replaced to assign two different variables but then the final assignment depends on the chosen path in the CFG (5b)

```

t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
  
```

Generating the TAC for a given arithmetical expression is also possible by traversing the abstract syntax tree. Obviously we have to assume that every operation has at most two operands. This means the AST has at most 2 children per node. Now each time we reach a node with operation  $\langle \text{opp} \rangle$  in post-order we choose a new temporary register  $t_i$  and append  $t_i := \langle \text{opp} \rangle (t'_\ell, t'_r)$  to the program where  $t'_r, t'_\ell$  will be the temporary registers which contain the result for the subtrees.

2) *Static-Single-Assignment*: Three-Address Code is good for control flow analysis and other optimization techniques. For example it is possible to compute dependencies or maybe to detect constant propagation and prevent redundant computations. For these reasons it is useful to define another constraint.

A Three-Address Code is in *static-single assignment*-form (SSA) if each register gets assigned only once[14]. This can simplify optimizations for example for register allocation because the life-time of a register and its assigned value during runtime is the same and the dependencies of registers/their values can directly be computed thus the definition-use pairs are explicitly visible in the code.

Transforming general TAC into SSA-form is not trivial. Obviously if there is a register in a sequence of instructions which gets assigned twice we just use a different register since the new assignment and replace each reference to the register in the following part of the sequence until the SSA-condition holds. For example the following sequence of assignments

```
A0:=5; B0:=2; A1:=B0 + A0; B1:=A1 + A1; C0:=B1
```

is semantically equal to

```
A0:=5; B0:=2; A0:=B0 + A0; B0:=A0 + A0; C0:=B0
```

and can be generated by just renaming some registers in some statements.

Although this method works for sequences it is wrong in general because the control flow of a program might have branches and the result of a variable could depend on the selected path as you can see this visualized in Fig. 5. We could use two different registers for each branch, but the result for `max` depends on the taken branch so it is impossible to form this into SSA-form directly. One solution for this case it would be store the value in memory (if supported by the register-machine) at assignments in a branch and load it back from the same address after the join. However this could prevent optimizations so in general SSA-form allows a special function  $\phi$ , also called *phony function*[15].  $\phi$  is defined for the register machine and choses the result value from both operand-registers based on the branch from which this instruction was reached. Our example computation can now be rewritten in SSA-form like:

```
L1: if r_a < r_b then goto L3:
L2:  t_1 := r_a
      goto L4
5 L3:  t_2 := r_b
      goto L4
L4: max := phi t_1 [from L2], t_2 [from L3]
```

A temporary register for each branch gets assigned and finally the result depends on the previous instruction of L4.

Note that real machines have no such functions like phi and one has to choose how to abstract the variable selection and what happens with multiple variables or multiple  $\phi$ -instructions. But this is no problem in general because we can and need to transform back from SSA-form before final code-generation. Methods for efficient optimization for SSA are presented in [7].

#### IV. IMPLEMENTATIONS

There are several implementations of different kinds of intermediate representations for several virtual machines or compiler engines. In this section we will present two popular implementations. One which use a register-based model and one using a stack-machine.

##### A. LLVM

The *Low Level Virtual Machine*, known as LLVM, is an infrastructure for compilers. It provides a powerful intermediate representation, such that it is possible to create different frontends for any programming languages. One example for such frontend is clang for C-like programming languages like C, C++, Objective-C or Objective-C++[1].

Figure 6 shows the architecture of the LLVM infrastructure[4]. For each programming language LLVM only needs a frontend transforming the source code into the LLVM-Intermediate representation (LLVM-IR), an SSA-register based representation. With this representation LLVM has the ability to optimize and pass the code also in LLVM-IR

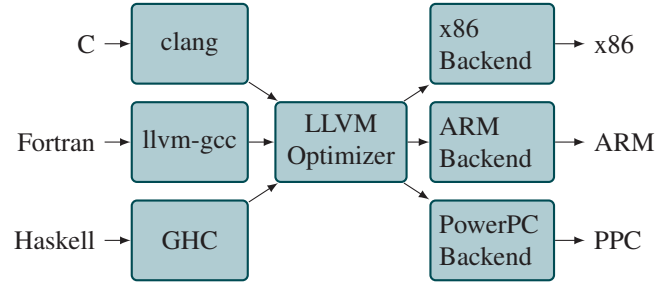


Figure 6: The Three-Phase Architecture of LLVM infrastructure [10]

to the backend for the specified compiler. This use of IR makes it easy and efficient to implement a new compiler for a new programming language or support a new processor architecture including GPUs[12], [4].

Different tools to perform different optimizations are available and also it is not necessary to directly generate the LLVM-IR from scratch in the frontend as we can use IR-Builder methods in the LLVM-library.

Given a c-program, we can use clang to generate LLVM code in human readable format. Assume we have the following C-Program:

```
#include <stdio.h>
2 void minmax(int a, int b){
    int max = 0; int min = 0;

    if(a>b)
        {max = a; min = b;}
7 else
        {max = b; min = a;}
    printf("_%d_<=_%d_", min, max);
}
int main(){
12 minmax(5,10);
    return 0;
}
```

Using clang with special options like `$ clang minmax.c -S -emit-llvm -o - | opt -S -mem2reg -o -` one can get the human readable format of the LLVM intermediate representation. Such that we receive the produced code for the register-machine:

```
1 define void @minmax(i32 %a, i32 %b) #0 {
    %1 = icmp sgt i32 %a, %b
    br i1 %1, label %2, label %3

; <label>:2                               ; preds = %0
6 br label %4

; <label>:3                               ; preds = %0
br label %4

11 ; <label>:4                               ; preds = %3, %2
    %max.0 = phi i32 [ %a, %2 ], [ %b, %3 ]
    %min.0 = phi i32 [ %b, %2 ], [ %a, %3 ]
}
```



```

%5 = call i32 @printf(i8*, ...) @printf(i8*
  getelementptr inbounds ([11 x i8], [11 x
    i8]* @.str, i32 0, i32 0), i32 %min.0,
    i32 %max.0)
ret void
16 }

declare i32 @printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
21 define i32 @main() #0 {
  call void @minmax(i32 5, i32 10)
  ret i32 0
}

```

Each register is given by a string with % as prefix followed by the register-name. We can see, that clang uses the variable-names from the C-file and generate new registers if required. In line 12 and 13 we can see that  $\phi$ -functions are support using labels for the branches.

Unlike a real machine instruction set we can see that LLVM uses a simple type system. `i8*` for example is a pointer to an octet. Another difference to the general TAC is that the LLVM-IR allows complex function calls and can pass special attributes that could be used by the optimizer or later optimization processes.

While some implementations like the Java virtual machine or the Microsoft.net Framework use a runtime framework on target systems also to provides additional libraries and just-in-time-compilation LLVM does not use this way. But the goal is to allow optimization in each layer of the compiling process including runtime-optimization in the backend. The solution for this is that the intermediate representation is used in each step. So after compiling c-files to .o-files it seems that after this we cannot use IR but generating a .o-file it just stores the IR inside instead of pure machine instructions. And LLVM gives has also the ability to optimize and pass IR-code after linking. This means that a resulting executable file can still contain intermediate which can be compiled and optimized again when the application gets started [4].

However with this architecture it is also no problem just implementing a backend for another high level programming language instead of a low level machine-language and just use LLVM to translate between different languages. This is done for example in a in the emscripten project which uses JavaScript as target language<sup>1</sup>.

### B. Common Intermediate Language

While LLVM provides an infrastructure to develop compilers efficiently by using the LLVM-IR and providing libraries for optimization, code generation and backends for various target systems there is a different way we can use intermediate languages. It is possible to share the software in a non-hardware-dependent format which contains an intermediate representation and move the whole backend phase on target machines. This method requires a special software often called

<sup>1</sup>Emscripten (<http://kripken.github.io/emscripten-site/>) is an LLVM-based project allowing the compilation of C or C++ code into JavaScript

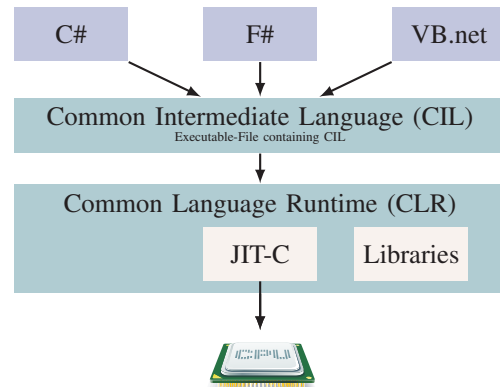


Figure 7: Architecture of the Common Intermediate Infrastructure realised in the Microsoft .net-Framework. The CLR is the implementation of the virtual machine which provides a standard class-library and performs the last compilation-phase.

*runtime environment* or *virtual machine* and it is the way the Java Virtual Machine (JVM) and Microsoft.net Framework works.

In the Microsoft .net-Framework applications are created and shared as PE-executable files (file-type: .exe). But those files are no pure PE-files[11]. They contain only a reference to load the main dll of the .net Runtime and payload data which is the intermediate representation that will be compiled just-in-time when the program runs.

The intermediate language for the Microsoft .net-Framework is called *Common Intermediate Language (CIL)* and is part of the ECMA-specification ECMA-335 for the *Common Language Infrastructure (CLI)*. This specifies an infrastructure for a runtime environment allowing applications, written in any language, to be executed without the need to rewrite the application for the target platform. The Microsoft .net Framework in this case is Microsoft's implementation of this specification as *Common Language Runtime* and a set of special libraries. Figure 7 shows the basic architecture of the .net-Framework which also contains class library[3], [8].

The CIL is a stack based intermediate language and has much more features than a general stack machine including a strict types and object orientation, and the fact that code can contain meta-informations.

```

public static int sum(int a, int b){
  int res=0;
  for(int i = a; i <= b; i++)
    res += i;
  return res;
}

```

This C# program would be translated into the following CIL-code:

```

.method public static hidebysig
default int32 sum (int32 a, int32 b) cil
managed
{
  .maxstack 2
}

```

```

.locals init (int32 V_0, int32 V_1)
IL_0000: ldc.i4.0    //
IL_0001: stloc.0    // sum = 0
IL_0002: ldarg.0    // load a on the
stack
9  IL_0003: stloc.1    // store a in first
var (i=a)
IL_0004: br IL_0011 // --+
IL_0009: ldloc.0    // | <--+
IL_000a: ldloc.1    // | |
14  IL_000b: add      // | |
IL_000c: stloc.0    // | |
IL_000d: ldloc.1    // | |
IL_000e: ldc.i4.1  // | |
IL_000f: add      // | | .
19  IL_0010: stloc.1    // | | .
IL_0011: ldloc.1    // <-+ .
IL_0012: ldarg.1    // load b |
IL_0013: ble IL_0009 // i<=b -+
IL_0018: ldloc.0
24  IL_0019: ret
}

```

With each method-call the *virtual execution engine* (VES) reserves part of the memory on top of an evaluation stack for the arguments and all local visible variables declared at the beginning of a method (see the `.locals` instruction in the listing). `ldloc.i` and `stloc.i` load and store the value of the  $i$ -th local variable (`ldarg` for argument) onto and from the stack.

As we can see variables are typed. CIL provides a huge set of instructions and also some for arrays, objects and structs. For example using `newobj` creates a new object instance and push its this-pointer on the stack. Then loading or storing fields is possible with `ldfld` or `stfld` which requires also the this-Pointer on the stack before.

## DISCUSSION

We just described how and where intermediate representations are used and we can see that the use in general allow us to have clean software-architecture for compilers with multiple layers or steps for the compilation processes. This gives us the possibility to build or extend good infrastructures for developing and optimizing software like LLVM and without the need to solve same problems for different environments. Although this in general simplifies the development of applications the big advantage is that it is not necessary . because one can develop software without regarding to the target system. In addition to that it is also possible to combine modules implemented in different programming languages. Designing new programming or domain specific languages gets faster and easier as developers only now need to implement a frontend which can produce the intermediate representation for the a virtual machine.

While this representations can be powerful we still need to be careful when we implement applications with real-time conditions. It might be difficult to estimate exact runtime when we use optimization methods. In addition if we have programs which should run on parallel systems we have to now if the IR is able to pass special information (or native

code binding) or if the IR can handle required features. For example there exists special projects to support OpenMP[2], [6] or also solutions by GPU-producers[12] for LLVM. We can also see the big advantage of the layer architecture as LLVM has already been extended with a separate IR in a new layer for the programming language Swift [9].

## REFERENCES

- [1] clang – language compatibility  
<http://clang.llvm.org/compatibility.html> (oct. 2015).
- [2] Openmp®/clang  
<https://clang-omp.github.io/> (oct. 2015).
- [3] Overview of the .net framework  
[https://msdn.microsoft.com/en-us/library/zw4w595w\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx) (nov. 2015).
- [4] CHRIS LATTNER AND VIKRAM ADVE. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.
- [5] CLICK, C., AND PALECZNY, M. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (New York, NY, USA, 1995), IR '95, ACM, pp. 35–49.
- [6] COWNIE, J. Openmp\* support in clang/llvm  
[http://openmp.org/sc13/OpenMPBoF\\_LLVM.pdf](http://openmp.org/sc13/OpenMPBoF_LLVM.pdf) (oct. 2015).
- [7] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.
- [8] EUROPEAN COMPUTER MACHINERY ASSOCIATION. *Standard ECMA-335: Common Language Infrastructure*, second ed., Dec. 2002.
- [9] GROFF, J., AND LATTNER, C. Swift intermediate language - a high level ir complement llvm  
<http://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf> (nov 2015), 2015.
- [10] LATTNER, C. The architecture of open source applications – llvm  
<http://aosabook.org/en/llvm.html> (oct. 2015).
- [11] MICROSOFT. Metadata and the pe file structure  
[https://msdn.microsoft.com/en-us/library/8dkk3ek4\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/8dkk3ek4(v=vs.100).aspx) (nov. 2015).
- [12] NVIDIA. Cuda llvm compiler  
<https://developer.nvidia.com/cuda-llvm-compiler> (oct. 2015).
- [13] SHI, Y., CASEY, K., ERTL, M. A., AND GREGG, D. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.* 4, 4 (Jan. 2008), 2:1–2:36.
- [14] STANIER, J., AND WATSON, D. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.* 45, 3 (July 2013), 26:1–26:27.
- [15] ZADECK, K. The development of static single assignment form  
<http://citi2.rice.edu/WS07/KennethZadeck.pdf> (oct. 2015).

# Static vs. Dynamic Typing

Toni Schumacher

Email: toni.schumacher@student.uni-luebeck.de

**Abstract**—The discord between static and dynamic typing has always been a controversial issue. In order to put away with the ambiguities and false arguments about this topic, this paper gives an overview of both techniques and compares them objectively. Supported by programming examples, and depending on programming concepts both typing mechanisms are weighed up against each other.

## I. INTRODUCTION

When asked for static or dynamic typing, there are many advocates of both camps. Most of these advocates answer the question, whether they prefer dynamic or static typing, reduced to their preferred known language [1]. The problem therefore is, that most programmers only know a single or only a small number of programming languages and thus only have a limited view of the totality of typing. In addition, only a few of these people know that widespread programming languages often combine static and dynamic mechanisms. The outcome of this is, that most people do not even exactly know what typing means or even have an incorrect definition of the terms. From this it follows, that these advocates then usually present false arguments to convince others of their point of view. They mostly only pronounce their known language and compare them with a (consciously) poorly chosen language in order to convince. To avoid this, we will objectively weigh up static and dynamic typing independent of programming languages. Only particular examples of implementations of certain languages are presented, but these languages are generally representative for all kinds of this typing.

But first we need to clarify what both terms stand for. Consequently, in chapter II, there will be an overview of the definitions of typing in general, followed by the precise definitions of static and dynamic typing. In chapter III we will then look at the advantages and disadvantages of static followed by dynamic typing. Afterwards we will take a look at advantages and disadvantages of both typing techniques applied on programming concepts. In chapter IV we will give a brief insight that static and dynamic typing also can coexist harmoniously. Then we will weigh in my own opinion on this topic. Finally, we will then draw a conclusion on the issue.

## II. TYPING

### A. What means typing?

Different values are interpreted by sequences of 0's and 1's in the memory. Only through type information the sequences are defined as a value, thus representing a certain variable. Therefore a type system, containing a type-checker, is required to classify phrases according to the kinds of values they compute [2] and assign the type information. But there are

also untyped languages that don't carry any tags for objects and don't distinguish different variables, such as Forth and assembly code. The term typing is often perceived quite differently. Generally accepted is, that a type-checker is built into a programming language to check values of objects for accuracy and to track type violation. Type-checkers can be distinguished in

- strong / weak,
- optional / explicit and
- static / dynamic typing.

So dynamic and static typing mean completely different techniques. However there are no clear boundaries between both. The consequences are that many people mix up typing and programming with languages and that it is often mixed up in articles and discussions about the topic. Then they assume that their used programming language, such as Java, is dynamic. They mistakenly expect, that, whenever a language contains dynamic elements, this language is a dynamically typed language and that statically typed languages only contain static type checking. But this assumption is wrong, since static and dynamic checking are mechanisms of programming languages and can not be equated with these. But let's first have a look in more detail what the terms mean.

### B. What is static typing?

Static typing is a technique of programming languages in which, *during the compile process*, the type checker tries to assign objects to their particular type. So static checking means that expressions and variables assigned to their values are checked for their accuracy before the program can be run. When doing this, type interferences occur or due to lack of type information no fixed expression or variable can be resolved, the compile attempt of the program code is canceled. Unlike dynamic typing, static typing check types at compile time and not at run-time. Although static and dynamic typing was introduced around the same time in programming languages, static typing is usually considered as the origin of dynamic typing. Because dynamically typed languages have finally come of age with Python and Ruby, while statically typed languages have been used much longer. Statically typed languages, for example are Ada, C, C++, Java, Fortran, Haskell, Meta Language (ML), Pascal, Perl and Scala.

### C. What is dynamic typing?

However dynamic typing means that the objects are checked for their types only *during the execution* of the program. Moreover classes, not types, are associated with their contained values and not with variables by tagging them with identifiers



such as `num`, `bool` or `fun` during run-time. The resulting runtime errors can then lead to partial or complete failure of the program. Therefore dynamic checking is inherently a restricted form of static typing with only a single type during compile-time, which also entails benefits. So dynamic typing is a special case of static typing. More than ever before dynamically typed languages are used to build large, widely used systems, but increasingly for domains that were previously the sole preserve of statically typed languages [3]. Dynamically typed languages, for example are Groovy, JavaScript, Objective-C, Perl, PHP, Prolog, Python, Ruby and Smalltalk.

In summary it turns out that static typing doesn't mean static language and dynamic vice versa. Unlike in most articles and discussions about this topic a static language uses static typing but can also use dynamic typing. Exactly the same applies for dynamic languages. However, the difference is that in static languages the main focus is the static type-checker and the dynamic typing (if existing) is not superficial. Conversely, this applies to dynamic languages. As we have seen, dynamic languages do perform class checks at run-time, but also static languages can do. Consequently, the already mentioned programming languages are not languages with the pure typing type. E.g. Haskell is called a static language, but Haskell also uses dynamic typing [4]. It is the point of different definitions that leads to most discussions about static vs dynamic typing.

### III. COMPARISON

Due to the fact that dynamic and static checking are almost opposite mechanisms as well as the huge technical and cultural gap between the respective typing communities, they can be compared very good. Most disadvantages of one typing are therefore benefits for the other one. Therefore we will deal with general advantages and disadvantages of both, static and dynamic typing. And for a general comparison we will depict the strength of both typing techniques in some programming concepts, independently of programming languages. Since we cannot evaluate static and dynamic typing for all programming concepts and important techniques for programming languages, we will take a look at significant mechanisms, that clearly favor a typing style. So far that some concepts are nearly impossible to implement with a certain type.

#### A. Advantages and Disadvantages of static typing

The greatest benefit static typing has, compared to dynamic typing, is the earlier detection of programming mistakes, because in software development process, the earlier you catch an error, the less expensive it is to fix the error. Advocates of static typing also prefer the increased runtime efficiency, which follows - among other things - from the more opportunities for compiler optimizations. This results in faster running statically typed programs than dynamic programs, because the compiler can optimize the choice of algorithms for operations on a specific value. For example many C compiler can use faster floating-point-specific microprocessor operations on a 32 bit

float data type [5]. But the C compiler must know that the specific value is a float and therefore it needs to be statically type checked. With static type-checking, programs can also be optimized by reducing their used memory. Because when the compiler knows the type of a value, it can optimize the storage it needs during runtime. Advocates of static typing also argue, that the benefits of static typing include better developing experience, because of the Integrated Development Environment (IDE) auto complete and name recognition. They also prefer the better documentation of statically type checked languages in form of type annotations, like in the well-known Java technical documentation from Oracle [6].

Often advocates also argue that 'well-typed programs cannot go wrong' which was coined by type theorist Robin Milner [7]. But it means something specific, it means that you can not exhibit undefined behavior from static typing and not that you can not imply that well typed programs do not contain bugs. Furthermore it only applies to a certain programming language, so it does not apply to all static typing languages. Because of properties that are not tracked by the type-checker of a static language, statically typed programs can still go wrong. Another disadvantage is, that statically typed programming languages are too rigid and can not handle changing requirements. Statically typed languages also can not handle a changing type of a variable. Following example would be illegal in a statically typed language:

```
employeeName = 9
employeeName = "Steve_Ferg"
```

Listing 1. Statically illegal checked code [8]

This is illegal, because, first, the variable `employeeName` would be an integer variable, containing 9 an integer. But in the second line `employeeName` is binded to a string and in statically checked languages variables can not change types. In a dynamically typed language this example would be perfectly fine and would not cause an error during compile-time nor run-time. Another drawback is, that statically typed code is less reusable than dynamically typed code and a programmer needs to write methods for every input the method could get. Advocates of dynamically typed languages also argue that statically typed languages are more verbose, but this is a fallacy and more dependent on the comparing programming languages than dynamic or static typing. Due to the fact that static typing should be more complete leads to complex and overly complicated concepts such as phantom types and wobbly types [9]. In order to be useful most statically typed languages compromise and define some exceptions as dynamic errors. An example for this is an array-out-of-bounds exception. In a programming language with statically checked array borders there must exist an array of length 1, an array of length 2, etc. Only thus it would be possible to check that all array accesses were in bounds. The drawback of such a system is, that it would not be possible to write functions over arrays of unknown size and this is not acceptable for practical programming. The original version of Pascal had array types

that fixed the size and this was probably one of the reasons that programmers rejected the language and Pascal version 1.1 introduced dynamic array length [10]. Nowadays all statically typed languages allow dynamic array size polymorphism, and check array bounds dynamically.

### B. Advantages and Disadvantages of dynamic typing

Advocates of dynamic typing argue that dynamic typing is better than static typing for prototyping systems with changing or unknown requirements, because you do not have to specify values and can handle different inputs. In addition dynamic typing allows programs to generate types and functionality based on run-time data, so it is much more flexible. For example, eval functions, which execute arbitrary data as code, become possible. An eval function is possible with static typing too, but requires advanced uses of complex algebraic data types. Furthermore dynamic typing is better for programs that interact with systems or modules with unpredictable changing output. It is widely accepted that more than 80% of all data is unstructured data [11]. Therefore it is a major advantage for dynamic typing towards static typing that it can manage with unpredictable generic data objects. Therefrom you can realise that dynamic typing is indispensable for dealing with truly dynamic program behavior. So on the one hand dynamic typing is more important for data intensive programming than static typing.

But on the other hand software development with dynamically typed languages leads to significantly more runtime errors and therefore to more costs in the development process. Because if no errors in the program should occur, much more tests need to be written to test all possible values of data the program needs to handle with. Unlike static typing, dynamic typing allows constructs that some static type checking would rule out as ill-typed. But this can also be seen as a disadvantage, because errors that could be rejected at compile time by static type checking are not found by dynamic type-checker until they run into this particular run-time error. To not crash the whole program you need to write exceptions, which causes much more effort to implement than in statically typed languages. Another big disadvantage of dynamic typing is, that errors can only be detected very late in the development process and consequently lead to complex troubleshooting and fixing errors. Usually these bugs are very difficult to locate, because they may occur long time after place where a programming mistake was made and a wrong type of data was passed into a variable it should not have. Often advocates of dynamically typed languages argue that dynamically typed program code is only interpreted and not compiled and therefore can be changed during runtime. However it depends on which programming language you use and is therefore no advantage of dynamic languages. The biggest advantage of dynamic typing is also one of the greatest disadvantage the type checking during runtime. It has many advantages but goes hand in hand with worse execution time, because the compiler must check all classes and associate their contained values by tagging them

with identifiers during run-time. Of course this is much slower than sign the objects to their particular type beforehand.

### C. Refactoring

Refactoring is the process of restructuring already written program source code while maintaining the observable program behavior. But that is the point why it is very complicated to implement with dynamically typed languages, because a benefit of dynamic languages is their flexibility and that is actually what makes the code very hard to maintain. However statically typed code is well structured code, every variable is known at compile-time and IDE's offer all sorts of options to semantically restructure the code. Thus it is much easier to refactor statically typed code than dynamically. Of course it can also be done with dynamic type checking but it makes considerably more work to write unnecessary tests and refactor your code by hand.

### D. Type inference

Most people mix up type inference and dynamic typing and think that you need to write static types like in Java, C, C++, Pascal etc. or even worse that you must write double type expressions like in Java:

```
Button b = new Button();
String s = "Doh!";
```

Listing 2. Usage of double typed expressions in Java [12]

But type inference and dynamic typing is something totally different, because type inference allows you to omit type information when declaring a variable and it can be used to determine which constructors to call when creating object graphs [12]. First implemented in ML and with completeness [13] type inference exist for more than 40 years only in statically typed languages like Standard ML (SML), Haskell, F# etc.

```
fun fak(n) = if n=0 then 1
             else n*fak(n-1);
```

Listing 3. SML function without any type specifications

This is an example from SML to calculate the factorial and it shows that even without any type specifications the compiler infers the correct type, in this example `int int`. In other words, static type checking allows programs to be more concise than dynamically typed ones, because this economy of notation effectively relies on the availability of static type information and thus is redundant for dynamic languages.

### E. Subtyping

With subtyping the programmer has the ability to override existing super types with a related datatype. Subtypes can also extend and/or specialize their super types with the overridden type. Overriding a super member sometimes completely changes the kind of that member. Thereby you can actually pass a value of any subtype into an object. Subtyping, or also called subtype polymorphism, is another technique that

facilitates that any term of a type can be safely used in a context where a term of the super type is expected. Subtyping should not be confused with the concept of inheritance from object-oriented languages, because subtyping is a relation between types whereas inheritance is a relation between implementations. With subtyping polymorphism the compiler can insert coercions to connect inferred and required types.

A static type-checker has the type information needed to automatically lift inferred to required types. E.g. C# has a type `int?` that donates nullable integers. In C# nullable integer extend normal integer and consequently enable subtyping polymorphism. Therefore in C# it is possible to subtype addition on normal integers to addition on nullable integers without throwing an exception, like in the following example:

```
int? a = null;
int? b = 1;
int? c = a + b;
```

Listing 4. Usage of subtyping in C#

With a dynamic checker it would also be possible to do automatic subtyping, but it would be very inefficient.

It would immediately exclude value types, since a dynamic type-checker does not produce type information during run-time, since it tries so associate values with classes. In conclusion, to add subtyping to a dynamic type checker you must rebuild the whole construct of dynamic typing.

#### F. Generics

Generics, or also called parameterized types, help avoid writing the same code multiple times and you just need to write a single function that handles arbitrary element types. To achieve that you create variables, methods, classes or interfaces (if existing). In dynamically typed programming languages any collection or method is automatically generic, since the type of the variable is at first available at runtime. Therefore it can be easily used to implement different behaviors for different types and it becomes possible to create highly reusable libraries. In statically typed languages generics are a lot harder to implement. In some statically typed programming languages without generics and subtype polymorphism you need to write a new function for any element type and any kind of collection.

```
new Set<object.getClass()>(object);
```

Listing 5. Generics in dynamically typed languages

This example of a Set of all possible variables needs to be replaced with following shortened code:

```
class Set{
    public Set(boolean b) { ... }
    public Set(int i) { ... }
    .. other constructors.
}
new Set<Object>(object);
```

Listing 6. Generics in static typed languages

It is evident that in this variant it is impossible to define the new Set of all types since there are endless types.

With dynamically scoped variables it is also possible to create generics in statically type-checked code [14]. It is much more complicated than in purely dynamically typed languages, but easier than in the example above. E.g. C# uses more complicated generic type definitions. So we can define classes of arbitrary type T as follows:

```
interface IEnumerator<T> {
    T Current { get; }
}
```

Listing 7. Usage of Generics in C# [15]

So on the one hand generics are not impossible in static typing. But on the other hand they are much more easier to implement in dynamically typed languages.

## IV. OUTLOOK AND OWN OPINION

### A. Hybrid languages

Static type checking focuses on complete checking of restricted specifications. Dynamic checking focuses on incomplete checking of expressive specifications. Thus, the goals of dynamic type checking and static type checking appear to be incompatible[16]. But there are different techniques of solving this misery. One possibility is to distinguish between statically typed and dynamically typed variables. E.g. C# uses this method, by checking all static variables during compile-time and all dynamic variables during run-time. In the following example you can see a Class Example and the Main method creating a dynamic and static instance of example without and with creating a compiler error:

```
class ExampleClass{
    public ExampleClass() { }
    public void exampleMethod1(int i) {}
}
static void Main(string[] args){
    ExampleClass ec = new ExampleClass();
    //would cause compiler error
    ec.exampleMethod1(10, 4);
    dynamic dynamic_ec = new ExampleClass();
    //no compiler error,
    //but run-time exception.
    dynamic_ec.exampleMethod1(10, 4);
}
```

Listing 8. Different ways so define variables in C# [17]

This shows that static and dynamic types are not opposed to one another and may coexist harmoniously.

### B. Own opinion

My own opinion is, that both dynamic and static typing are useful, but in different circumstances. Dynamic typing should be used for small programs and scripts, that must be developed very fast and on which no major safety requirements are placed. On the other side static typing mechanisms

should be used for applications relevant to security. Therefore both should be implemented in an high level programming language, because a fully expressive language is one that supports the delicate interplay between static and dynamic techniques. Some languages already exist that have both static and dynamic type-checking. The static type-checker verifies what it can, and dynamic checks verify the rest.

My point of view is, that all programming languages should implement such a type-checker. And not like now, where languages with static type-checking exist that provide a way to bypass the type checker. Because that is not the right solution of solving the problem. So my opinion about this discussion is that programming languages should use both static typing where possible and dynamic typing when needed.

## V. CONCLUSION

Static typing and dynamic typing is a topic of programming language design that is not always clearly defined and, as a result, is not very well understood. This article has given you an insight into the concepts of static and dynamic typing and we have considered several different advantages and disadvantages of static and dynamic typing. To complicate the matter even more, there are more ways, like mentioned before, of typing mechanisms, such as strong and weak typing, which would inflate this paper.

The question remains, if static typing is better than dynamic typing or vice versa. In this paper we had taken a look at both, advantages and disadvantages, and when to use them. It is obvious that both, static and dynamic typing, have more benefits than downsides and therefore both are better than no type checking. Now everyone should weigh the benefits only for their preferences and then decide which typing mechanisms, or maybe both techniques, he or she prefers.

## ACRONYMS

<b>IDE</b>	Integrated Development Environment
<b>PHP</b>	PHP: Hypertext Preprocessor
<b>ML</b>	Meta Language
<b>SML</b>	Standard Meta Language

## REFERENCES

- [1] [stackoverflow.com/questions/125367/dynamic-type-languages-versus-static-type-languages](https://stackoverflow.com/questions/125367/dynamic-type-languages-versus-static-type-languages) (November 2015)
- [2] [stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages](https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages) (November 2015)
- [3] Benjamin C. Pierce, *Types and Programming Languages*, ACM SIGACT News, Volume 37 Issue 4, December 2006 Pages 29 - 34
- [4] Laurence Tratt, *Dynamically Typed Languages*, Advances in Computers, Volume 77, 2009, Pages 149-184
- [5] John Peterson, *Dynamic typing in Haskell*, Technical Report YALEU/DCS/RR-1022, Yale University, 1993, Available: [cs.yale.edu/publications/techreports/tr1022.pdf](http://cs.yale.edu/publications/techreports/tr1022.pdf) (November 2015)
- [6] [en.wikipedia.org/wiki/TypeSystem](http://en.wikipedia.org/wiki/TypeSystem) (November 2015)
- [7] [docs.oracle.com/javase/8/](http://docs.oracle.com/javase/8/) (November 2015)
- [8] Robin Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, Volume 17, Number 3, December 1978, Pages 348-375
- [9] pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/ (November 2015)
- [10] Vincent Driessen, *The Power of Wobbly Types*, Radboud University Nijmegen, The Netherlands, 2006, Available: [cs.ru.nl/bachelorscripties/2006/ThePowerOfWobblyTypes.pdf](http://cs.ru.nl/bachelorscripties/2006/ThePowerOfWobblyTypes.pdf) (November 2015)
- [11] [freepascal.org/docs-html/ref/refsu18.html](http://freepascal.org/docs-html/ref/refsu18.html) (November 2015)
- [12] Seth Grimes, July 2005, Available: [informationweek.com/software/information-management/structure-models-and-meaning/d/d-id/1030187](http://informationweek.com/software/information-management/structure-models-and-meaning/d/d-id/1030187) (November 2015)
- [13] Seth Grimes, August 2008, Available: [breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/](http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/) (November 2015)
- [14] Erik Meijer and Peter Drayton, *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, Microsoft Corporation, 2004, Available: [ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf](http://ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf) (November 2015)
- [15] Luis Damas, *Type Assignment in Programming Languages*, PhD thesis, University of Edinburg, 1985
- [16] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields, *Implicit Parameters: Dynamic Scoping with Static Types*, 27. POPL 2000, Boston, Massachusetts, USA, Pages 108-118
- [17] [msdn.microsoft.com/de-de/library/58e146b7.aspx](http://msdn.microsoft.com/de-de/library/58e146b7.aspx) (November 2015)
- [18] Cormac Flanagan and Kenneth Knowles, *Hybrid type checking*, University of California at Santa Cruz, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 32, Number 2, January 2010
- [19] [msdn.microsoft.com/de-de/library/dd264736.aspx](http://msdn.microsoft.com/de-de/library/dd264736.aspx) (November 2015)

# Memory models

Gunnar Bergmann

**Abstract**—Resource handling is an important aspect of programming languages and related errors are still a common problem in today’s software. Managing it manually is hard and error prone, so safer alternatives are required. Garbage collection protects against most problems with memory handling but is not usable in programs with strict memory and time constraints nor can it handle other type of resources.

This paper presents alternatives to manual resource management that are safer to use but do not impose severe performance degradation. First it explores the concept of RAII and shows how it can be implemented, how typical classes look and then presents and solves limitations of RAII. Afterwards Rust’s ownership model will be explored. It helps preventing typical security issues currently present in C++ by applying additional checks at compile time.

## I. INTRODUCTION

In some languages, most notably C, the programmer needs to manage the memory by explicitly calling `free` whenever it is no longer needed. This leads to many problems:

- memory leaks when memory is never released
- use after free when a reference to an already freed location is accessed
- double delete when memory is deallocated twice
- repeated code because the memory needs to be released on every code path. As usual copied code often leads to other problems, especially when it is changed later.
- unreadable code when many nested *if*-statements are used to prevent repeated code.
- separation of allocation and release
- Adding exceptions is problematic because there needs to be a finalizer for every allocation.

Detecting and debugging memory related errors is hard. Leaked memory accumulates slowly and needs a certain amount before causing problems. In C *Use after free* and *double delete* lead to *undefined behavior*. It is possible that everything works as expected and just under rare circumstances cause crashes or security issues. There are programs for the detection of potential errors but they are rarely used and can not detect all issues.

Garbage collection solves memory management by tracing references and detecting which objects can be safely removed. It is not usable in software with strict memory constraints and the unpredictable frequency of collection cycles limits the benefit for software with real time requirements. Garbage collectors also can not handle other resources than memory because these are often unique, lack additional reserves and require a deterministic order of resource release.

Except for *use after free* all these problems can be solved by different classes using RAII [7]. Still *use after free* remains an

important issue that causes lots of problems that the ownership model introduced with Rust [4] solves.

The alternatives presented by this paper will also be able to create file handles or network sockets, that automatically close, as well as mutex locks that can unlock when no longer needed [7].

## II. LOCAL VARIABLES

The easiest form of automated memory management are local variables. Local variables can not outlive their scope and the memory is automatically reclaimed by the program [1]. They can be returned from functions by copying their values but pointers to the variables become invalid. It is easy for compilers to save them on the stack or hold them in registers, making local variables the fastest form of memory allocation, because deallocation takes places by decreasing the stack size at the end of each function and the locality of reference avoids cache misses. It is not possible to increase the capacity of stack allocated memory once it is no longer on the top of the stack and a stack allocated variable can not outlive the function call.

These limits make exclusive local variable impractical for most cases.

## III. RAII

When C++ added object oriented programming to C, it also provided a way to encapsulate initialization and release of member variables. Classes provide a better encapsulation of their data and thus may have a *constructor* for the initialization and a *destructor* for the resource release. [7]

A minimal string class may look like

```
class String {
private:
    char* data; // pointer to a character
public:
    // Constructor
    String(const char* s) {
        data = new char[ strlen(s)+1];
        strcpy(data, s);
    }

    // disable copying
    String(const String&) = delete;

    // Destructor
    ~String() {
        delete[] data;
    }
};
```

This concept is known by the name *Resource Acquisition Is Initialization* abbreviated as *RAII*, because the resource used by the object is allocated on its construction.

Instances of these classes can act as an extension of normal local variables. They can also be stack-allocated but unlike the primitive ones their destructor is run at the end of the scope.

The string class from above can be simply extended to allow dynamically growing strings. It just needs to allocate a new chunk of memory and copy the old data and the extension into a single array.

```
concat(const char* s) {
    char* old = data;
    int len = strlen(old)+strlen(s)+1;
    data = new char[ len]; // more memory
    strcpy(data, old);
    strcat(data, s);
    delete[] old; // free old memory
}
```

The memory can now grow dynamically in a completely safe way but is still bound to a scope and automatically deallocated by the destructor. Like other scoped variables the values can be passed out of the scope by creating a new object with the same content. Depending on programming language and implementation the content is either copied or transferred to the new object, leaving an empty one behind.

When an object is destroyed, members and base classes destructors are also called [1]. This allows safe composition of multiple classes, where each one uses RAII. Typically some or all of the operations are generated by the compiler, so that except for writing some basic data structures and wrappers around imported code from other languages it is not necessary to provide these for each class.

RAII can be used for many types of resources. Some examples from C++ are the classes `fstream` for files and

`vector` for dynamic growable arrays. The prime example is the usage of RAII for automatically releasing mutex locks:

```
{
    lock_guard<mutex> guard(some_mutex);
    // ... code inside the mutex
} // some_mutex automatically unlocked
```

#### A. Destruction order

An object is destroyed at the end of its lifetime, which can be automatically done for temporary objects and local ones when they leave the scope, in many languages visible as the closing brace `}`.

Member variables are destroyed by their parent's destructor from the last to the first and afterwards the base classes' destructors are called if existent [1].

In all the automatic destructor calls the variables are destructed in reverse of their creation order and independent from the code path [1]. RAII's destruction order is deterministic. The reverse order ensures that no object is destroyed while others depending on it are still alive.

There are additional ways of creating and destroying objects manually, which is needed for implementing some basic types that itself use RAII but need to manage their internal memory manually. For a simplified example the C++ type `vector` allocates a chunk of raw memory and constructs the objects internally. When elements are removed or the `vector` is destroyed, then all of the elements destructors are called before the object itself gets destroyed. You rarely have to do the management yourself unless you need some specialized and efficient datastructures that are not part of the standard library.

When an exception is thrown, the runtime library performs *stack unwinding* by going back all function calls and destroying every object until the according catch-statement has been reached. Destructors should not throw exceptions because throwing in a stack rewind would lead to two simultaneous active exception which most languages can not handle and immediately abort. [1]

#### B. Solved Problems

Under the assumption that the implementation of classes is correct, RAII solves some of the problems of manual memory management. RAII improves the code structure by reducing code duplication. It pulls both acquisition and release from the code using it into one class, that keeps the related code close together. By removing the need for calling `new` and `delete` outside of the class it solves *double deletes* and memory leaks. It does not solve *use after free* because there may still be references to memory after the owning object has released it. Also data structures require strict hierarchies so that there is an order for the destructors. Cyclic references are not possible although the section about *smart pointers* shows ways to relax these constraint.

### C. Containers

The most important building block for programs are *containers* [7]. These are generic classes that allocate memory for holding multiple objects and vary in their performance characteristics and usable operations. On destruction they call the destructors of all the elements in them and free the memory. The most important container is `vector` for storing a dynamically growable array and `map` or `unordered_map` for associative containers.

Some operations can remove, add or internally move elements, causing pointers to individual elements or *iterators* to dangle. In modern C++ this is a common error because it is often not visible from the outside that elements may move and under which circumstances, although some containers guarantee stable references.

### D. Smart Pointer

Unlike using objects directly, a pointer can be moved but the memory it refers to is not invalidated, so all other pointers remain valid. We use the term *owning pointers* to refer to those types of pointers that need to free the memory on release. Later we will explore the ownership in the Rust programming language that uses the same concept in a way the compiler can verify to prevent common sources of errors.

In addition to *owning pointers* there are other types of pointers that just refer to a memory location without ever acquiring or freeing the memory themselves. The pointers present in C are called *raw pointers* here.

Since raw pointers need to call the appropriate function to release the memory, it is safer to provide a wrapper class that uses RAII to release the memory automatically. These are called smart pointers. As a general rule owning raw pointers should be avoided and either replaced by using the object without indirection or with a smart pointer. There are typically three types of smart pointers [3]:

- `unique_ptr` (C++), `Box` (Rust) is the most simple type. It holds a reference and frees the memory on deletion. It can not be copied. It models exclusive ownership [3].
- `shared_ptr` (C++), `Rc` (Rust) provides shared ownership. It lets multiple smart pointers refer to the same memory location. When the last one is destroyed it frees up the memory. `shared_ptr` is implemented with reference counting. When creating a new object it allocates additional space for another integer and stores a counter there. When a `shared_ptr` is copied, it increases the counter and the destructor decreases it. When the counter drops to zero the inner object is destroyed. This smart pointer allows multiple references but the programmer needs to be careful to not create a cyclic dependency because that prevents the counter from reaching zero and may leak the whole cycle [3].
- `weak_ptr` (C++), `Weak` (Rust) refers to an object managed by shared pointers. It allows the pointer to dangle but that can be safely detected. It does not own the resource nor can be used directly but can grant temporary

ownership by being upgraded to a `shared_ptr`, which may fail if no associated object exists. The weak pointer can be used to break cycles [3].

For example in a tree structure each node has a list of shared pointers to child nodes and a weak pointer to its parent. It forms a hierarchy because when the last reference to the root node is dropped the whole tree will recursively be released. It is still possible for a node to safely access its parent.

Weak pointers are implemented by adding another counter to the one used by *shared pointer*. When the reference counter drops to zero the object is destroyed. The counter object is released on destruction of the last *weak pointer* [3].

Both containers and smart pointers store memory but they are used differently. Containers organize multiple objects of the same type, whereas smart pointers contain a single element, but vary in the access to it, and when inheritance is used, they allow downcasting to a base class.

## IV. RAII IN GARBAGE COLLECTED LANGUAGES

A garbage collector is an easily usable protection against memory-related issues but lacks support of other resources. Classes often have *finalizers* that can close leaked resources but many of them, including file handles and network sockets, need to be closed as soon as possible so that other programs can use them. Often the release order matters or it takes an unpredictable time span to the next collection cycle so it sometimes takes too long and may cause nearly undetectable errors. Relying on *finalizers* therefore is strongly discouraged.

Traditionally garbage collected languages use `finally` to run specific code for resource release at the end of the scope, although that requires the discipline to surround every resource usage with a `finally` block and as a consequence is rarely used. Also resource handling occurs more often than defining a class containing a closable resource, so RAII reduces code duplication in comparison to `finally` [6].

Some languages like *D* support a garbage collector for memory and RAII for other resources [2]. Other languages were extended with special keywords that enable RAII-like behavior. For example *Python 2.5* [5] introduced the methods `__enter__` and `__exit__` that can be used with the new keyword `with` to support automatic closing of objects:

```
with open("test.file") as f:
    content = f.read()
```

## V. RUST

Rust is a new language that aims at memory safety, abstractions without overhead and multithreading. [4]

Traditional RAII can resolve some common errors but it does not prevent against dangling references. Modern programs heavily rely on iterators and modifying a container while iterating over it may cause the iterator to point to now invalid memory similar to a dangling reference or skip elements or visit them twice.



Accessing the same memory location from to different threads may easily lead to data races. Even when a RAII ensures safety for a variable, another thread may access in an intermediate state that is not safe to use.

Rust uses the concepts of ownership, borrowing and lifetimes to eliminate these bugs without introducing additional overhead.

#### A. Ownership

Ownership means that there is always a single variable that can access the value and return it. The ownership can be transferred to other variables but not shared. [4]

The strict semantics of ownership ensure that RAII is used correctly and that no variable can be shared across threads.

Variables have ownership over the resources they access. You can create a heap-allocated array with three elements with

```
let a = vec![1, 2, 3];
```

The variable `a` has ownership over the content of the array. When `a` goes out of scope it automatically reclaims the memory by using RAII. You can transfer ownership to another value.

```
let b = a;
```

The array is moved to `b`. Accessing it afterwards is a compiler error.

```
a.push(4); // append 4
```

will not compile because the content of `a` has been moved to `b` and is no longer accessible.

When multiple control paths exist and a value has been moved in one, it is no longer accessible afterwards until a new value has been assigned. Functions can take a value by move, which makes the code more efficient than copying values and also prevents accidental modifications.

Some types are an exception from the ownership rule and are copied instead of moved, because move and copy are equal for them.

#### B. Borrowing

Ownership alone is not useful because just a single source can access a value and you can not even pass it to a function without losing the ownership. [4] Instead you can temporarily borrow the value. It is technically just a reference like in C++ but with additional checks at compile time.

Rust employs a concept similar to a read-write-lock: Only one mutable reference at a time is allowed but multiple immutable ones. Borrowing makes it impossible to transfer ownership or modify a container while a reference or an iterator to it exists.

Having multiple read-only references is save because no one can change anything.

```
fn foo() -> &i32 {
    let a = 12;
    return &a;
}
```

will not compile because the local variable `a` goes out of scope at the end of `foo` and invalidates the returned reference.

```
let x = vec![1, 2, 3];
let reference = &mut x;
println!("{}", x);
```

will not compile because `x` can not be used by the print function while there is still a mutable reference to it.

This is not an actual lock and does not make the types thread-safe. Sending a reference to another thread is not allowed. It just prevents creation of dangling references. Still the same mechanism is also used for thread-safety to prevent unlocking a mutex while still holding a reference to its content:

```
{
    let guard = mutex.lock().unwrap();
    modify_value(&mut guard);
}
```

#### C. Lifetimes

The compiler enforces and validates the borrowing and ownership rules with lifetimes. These are bound to the scope and every type has an implicit lifetime on it. [4] References are represented by a leading single quote: `'lifetime`. There is a special lifetime `'static` for items that life as long as the process exists.

```
struct Foo {
    x: i32,
}

// the lifetime 'a is a generic parameter
// it returns a reference with the same
// livetime as the input
fn first_x <'a>(
    first: &'a Foo,
    second: &Foo)
    -> &'a i32 {
    &first.x
}
```

In this example the parameter `first` and the returned reference are annotated with the generic lifetime parameter `'a`. It is not possible for the caller to let the reference outlive the object or modify it while the reference still exists. Taking immutable references is still possible, but mutable ones are forbidden.

Although the compiler generates and checks the lifetime for every object, they can be automatically generated based on the context except for some ambiguous cases.

#### D. Anonymous functions

Like most modern languages Rust supports the creation of anonymous functions, also called lambda functions. These are created locally and can access the local variables. To obey the safety rules without introducing additional overhead there are three basic types of anonymous functions.

The simplest type accesses the local variables by reference. Explained in the terms of ownership it borrows its environment. It can not be returned from the function because then references could no longer access the data.

The second type takes the variables by move and forbids the access after the function definition, which means it takes the ownership of the variables. For taking copies of the value first store a copy and then use the copy inside of the function.

The last type can be called only once. This allows the function to consume the inner data instead of leaving it in a valid state. To build similar behaviors for own types these can take the self-reference by move. `self` accesses the object on which the method is called, similar to the `this` by other types. Rust's ownership ensures that the object is moved inside a method and then destroyed at the end of the call. [4]

```
fn do_something(self) {
    // ...
} // self is destroyed
```

C++ in comparison lets the programmer decide for each value. There are two default capture modes which take all variable by value or by reference. There are also individual choices for each variable. The C++14 standard also added *generalized captures* that can bind any expression to a name. [3]

```
[ x, // by value
  &r, // by reference
  p = make_unique<int>(0)
    // generalized capture
] (auto parameter1) {
    // the code in the function
}
```

There is no mode that supports moving values inside a function but generalized capture can be used for that.

The C++ lambdas are more flexible, but also more dangerous because no checks occur. Accidentally using the wrong capture mode, especially when the default ones are used, can easily lead to dangling references. They are also more verbose and you can not build the self-destructing ones.

### E. Limits of ownership

It is not possible to protect against all kinds of bugs even with the strict ownership rules. Especially the value inside of a shared smart pointer has an unpredictable lifetime. While it can be guaranteed that the value has at least the pointers lifetime, aliasing mutable references are needed for many programs. There are some types in Rusts standard library that allow safe access e.g. by falling back to runtime checks.

### F. Ownership in C++

Ownership is nothing that was invented for Rust. As mentioned in the smart pointer section it is often useful to think about owning references for structuring a program [3]. Only the application of additional checks to enforce correct ownership semantics is new.

At CppCon 2015 a new programming guide and a tool, that can check for rules similar to the rust compiler, although less strict, were announced. Some of Rust's rules for example those that protect against data races in multithreaded programs are also not covered. [8]

## VI. CONCLUSION

This paper has shown how RAII can create a safer and simpler alternative to manual memory management and how some internally unsafe classes can provide useful building blocks for creating quite safe programs. Afterwards it has presented the ownership concept of Rust that prevents common problems of RAII.

In comparison to garbage collection RAII can not handle cyclic references. The programmer always needs to use an adequate container or smart pointer, structure the code and break cyclic references manually. On the other hand this forces a good structure for the program and allows a deterministic and immediate destruction which enables the use of RAII for other types of resources.

Whereas garbage collection is a general solution for all kinds of memory related issues, RAII is one for all kinds of resources except those ordered in cyclic dependencies.

Ownership can eliminate many issues at compile time and provides a fine grained control over the destruction of objects. Other languages have to detect problems at runtime or don't protect against them at all. This does not just prevent some sources of memory issues but also other classes of problems that need to be found by extensive testing otherwise.

On the other hand it is a lot to write. Simply passing a reference around is not possible because you need to ensure that no invalidated variable can be used and sometimes extra lifetime annotations are required so that the borrow checker can verify the code. Even when you sacrifice performance and make extensive use of copying objects or just use reference counting, there is still a lot work involved in manually creating copies every time and you loose the ability to use mutable objects, which need other verbose workarounds.

It remains open if the compile time checks and improved code structure will benefit the creation of large scale and reliable software or if the amount of work and the costs are too high for productive software development.

## REFERENCES

- [1] Working Draft, Standard for Programming Language C++. Technical report, 2015. N4296.
- [2] Ali Cehreli. *Programming in D: Tutorial and Reference*. CreateSpace Independent Publishing Platform, 1 edition, 8 2015.
- [3] Scott Meyers. *Effective modern C++ : 42 specific ways to improve your use of C++11 and C++14*. O'Reilly Media, Sebastopol, CA, 2014.
- [4] The Rust Project. The Rust Programming Language. <https://doc.rust-lang.org/stable/book/>, 2015. [Online; accessed 12-November-2015].
- [5] Python Software Foundation. Python 2.7.10 Documentation. [docs.python.org/2/](https://docs.python.org/2/), 2015. [Online; accessed 12-November-2015].
- [6] Bjarne Stroustrup. FAQ. <https://isocpp.org/faq>. [Online; accessed 15-November-2015].
- [7] Bjarne Stroustrup. Foundations of c++. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2012.

- [8] Herb Sutter. Writing good C++14 ... by Default. <https://github.com/isocpp/CppCoreGuidelines/blob/master/talks/Sutter%20-%20CppCon%202015%20day%202%20plenary%20.pdf>. [Online; accessed 12-November-2015].

# Stack Based Programming Paradigms

Timo Luerweg

**Abstract**—Within the field of Programming Languages, Stack based ones are allot less noticed than most other. Stack based Systems are commonly used, be it as intermediated Language for virtual machines, or as a foundation for meta programming. Within this paper, I will give an overview of the topic of stack based programming. It discusses the most important Paradigms such as variable elimination . Afterwards there will be a short overview of some languages implementing the presented Paradigms and individual characteristics.

## I. INTRODUCTION

Stack based languages are widely used, yet mostly unnoticed. The reasons behind this could be the uncommon still efficient syntax, the implicit passing of parameters or the fact most languages of this type are highly specialized. However within the filed of their respective usage they are highly used. The first part of the paper will deal with this topic in more detail. Afterwards I'll discuss paradigms implemented by most of the languages, beginning with the special syntax(postfix) and the conversion to this syntax. Then I will pay special attention towards variable elimination. The reason behind this, is it works totally different form comparable mechanism in commonly used 'modern' languages. This part will be split in two main parts: the local and the global variable elimination. Afterwards there will be a small overview of some frequently used or conceptual interesting languages.

## II. CONCEPTS

### A. Syntax

Nearly all stack based languages use the Reverse Polish Notation(RPN). This is a mathematical notation where the operator is placed behind its arguments, in contrast to the commonly used infix notation where a operator is placed between its operands. This notation provides advantages for stack based languages, due to the fact that it's similar to their way of operating. Furthermore this notation provides a faster way to calculate, because it passes on using parenthesis on the given term. To transform from one to another an algorithm called 'Shunting-yard algorithm' exists. It uses a stack to store the operators (like +/ - ·) and emit them to the expression, when needed. lets look at an example:

$$4 + 5 \cdot 8 \cdot (3 + 6) + 2$$

Is in commonly known infix notation. The next line would represent the same function in RPN:

$$4 5 8 \cdot 36 + \cdot + 2+$$

The above would be the way, you would calculate the function, in e.g. Forth. Of cause this can be done, not only with mathematical operations, but with all kind of operations(even self-written ones). There for stack programs are written as a sequence of values and instructions.

## III. FIELDS OF USE

Stack based languages are not falling short in terms of usability in contrast to 'modern' languages. However they are mostly used in field where they possess a natural advantage over the last. This would be the area hardware-close implementations( e.g. Forth is frequently used here) or as software for postfix using calculators.

### A. Intermediate Language

An Intermediate Language describes on an abstract level, hardware or platform independent, data structures for programs. This is done, to later translate this abstract code to a hardware specific code, without the need to compile the code for each individual pair of hardware and compiler. However this topic was covered earlier within this seminar, by Malte Skambath so if you need to refresh your knowledge regarding the topic I recommend to reread his paper.

In addition to his coverage of the topic, Stack based languages are frequently used as intermediate languages. Two examples would be EM, Java Byte code or Elisp byte-code. The reason behind this is, the characteristic, to be condensed to very compact binary code in addition to the reduced set of operand, which will speed up the syntactical analyse, leading to a faster execution.

### B. Meta Programming

Meta programming is the topic of automatic code generation and manipulation. Again this topic was already covered within our seminar, by Alexander Schramm. Stack based languages (he took Java and Java byte code as an example) can be used within runtime meta programming, because they are able to build and execute programs at run-time. [2].

## IV. IMPLEMENTATIONS

### A. Forth

Forth is a imperative stack based programming language. It originated from Charles Moores work in the beginning of 1960. Forth possesses two stacks: one stack where data is stored and manipulated(data stack) and another one where return addresses are stored(return stack). While the data stack is used to pass data between the words of a program, the return stack is used to keep an overview of where to return to(in case of

executing a subroutine) and stores local variables which would otherwise cause problems. Although Forth is not specialized, today it's mostly used in the field of embedded systems and micro controllers. One reason for this could be, the possibility to generate and optimize and compile code before transferring it to the specific machine. 'Forths are often characterised by the size of an item on the data stack, which is usually 32 bits or 16 bits. Several 64 and 8 bit Forths also exist, as well as 4, 20 and 24 bit systems. Forth is an untyped language, so there is nothing to stop you adding a character to a number. No casts are necessary. The amount of memory needed to store a stack item is called a cell.' ([5])

If you are further interested in this Language you can take a look at [5] for further details and an introduction to the topic.

### B. Joy

Joy is a functional language. Other than most other functional languages, Joy is not based on the lambda calculus. It is rather based on the composition of functions. All functions take a stack as argument and produce a stack as value. This however allows a Joy program to be significantly more powerful than e.g. a Forth program due to the possibility of passing whole data structures(or program structures) instead of a limited amount of cells. Furthermore a Joy program, in theory, can take and returns an unlimited amount of parameters. E.g. a '5' would not be interpreted as the integer 5, rather it is interpreted as an program, that pushes the '5' onto the stack. It was designed and developed by Manfred von Thun, currently working at the La Trobe University in Melbourne, Australia. if you are more interested in the details of this language, you can download the current version and all papers about this language here [11].

### C. PostScript

PostScript is a programming language used to generate (vector) graphics. Printers often use PostScript to generate their documents. PostScript can be considered as an intermediate language, because a majority of the code, is generated by programs to convert text files e.g. .doc to a postscript file which then can be used by a printer.

## V. OPTIMIZATIONS

### A. Variable elimination

The concept of variable elimination is to eliminate as many variables(and with this memory accesses) used in an program as possible. This is ultimately done, to speed up the execution of programs. To accomplish this goal with variable elimination, there are some requirements to be met. First of all the (register-based) program needs to be split-able in (equally large)basic code blocks. In addition blocks of different branches (IF-THEN-ELSE) 'the IF and ELSE clauses must both leave the same temporary variables in the same locations on the stack when control transfers' ([3]). Another (in some cases optional) requirement for those blocks would be, after the execution of each basic block the Stack its preformed on, needs to be empty. With this conditions in Mind, there are two possible ways to

perform variable elimination: local(intra-block) or global(inter-block). While there exist algorithms to perform local variable elimination [3] [1], there aren't any algorithms for global elimination so far, mostly because its very hard to eliminate variables with a lifetime longer than a basic block(atleast with an algorithm), but it can be applied manually after the local elimination took place. Note this two concepts are discussed in the context of Forth and while most parts discussed stay available for other Languages, there are steps which need modification in order to work in those languages.

1) *Local variable elimination*: The first step is called **Raw input processing**. Within this step either the intermediate code structures or the source code itself(again it depends on the language) is transferred to a list of stack-based instructions. However this transformation is only happening on a basic level and the result is later used for further elimination and processing and is not executable(naive processing).

The second step is **Code clear up**. At this point, the transformation to stack-code is completed. This includes the modification of subroutines, so they take their input from the stack and return it back to the stack, instead of using registers. Also condition code is modified to use stack-based comparison, instead of register-based.

Next an initial phase of **Peephole optimization** is performed. This leads to a more consistent code.

This step is the key of the whole algorithm the **Stack scheduling**. The purpose is instead of using local variables in the code, the needed values are on top of the stack(or atleast reachable) when they are needed. In detail the code is optimized so the number of fetch and load operations is minimized. This is accomplished by firstly searching for local pairs of fetches and stores of the same variable, ranking them by distance and secondly try to eliminate these pairs in order of their distance. Stack scheduling can be performed, if the local variable is used more than once(example below).

The next step is another round of **Peephole optimization**. This time however after the elimination took place the code the code is cleared form complex stack operation, which can be simplified e.g.:

DUP SWAP becomes DUP  
(DEAD) LOCAL! becomes NOP

The last step **Code generation** converts the so far generated stack code to the individual machine code. Within this process non-standard words used by the algorithm are also translated to normal Forth words.

*Example of a local variable elimination*: First let me explain the notation: the brackets represent the Stack we would preform our operations on, while everything on the left side of the delimiter – represents the stack before we perform the operation, the right side represents the stack after we performed the operation(to simplify the code this part is excluded form the example).

The **LOCAL@** represents a (non-standard)load in Forth<sup>1</sup>, so e.g. in the first line, the variable **b** is loaded from the local register **76** and pushed on the stack.

Along with this definition, the **LOCAL!** represents a store in Forth<sup>2</sup>, as seen in line four, where the top item of the stack is saved at the local variable **75**, in our case **a**.

The line with the comment (*DEAD*) at the end, identifies this load statement as useless and is a remainder of step 1 and will be eliminated in step 5.

Now to the example. Lets assume this is a part of the source code we want to optimize with stack scheduling.

```

1  a = b * c ;
2  b = a / 8 ;
3  c = a - b ;
    
```

This would be the naive translation to forth-like code(step 1).

```

1  (      ---) 76 LOCAL@  \ b @
2  (   76 ---) 77 LOCAL@  \ c @
3  (76 77 ---) *
4  (   75 ---) 75 LOCAL!  \ a !
5  (      ---) 75 LOCAL@  \ a @
6  (   75 ---) 8          \ literal 8
7  (75 88 ---) /
8  (   76 ---) 76 LOCAL!  \ b !
9  (      ---) 75 LOCAL@  \ a @ (DEAD)
10 (   75 ---) 76 LOCAL@  \ b @
11 (75 76 ---) -
12 (   77 ---) 77 LOCAL!  \ c !
    
```

Afterwards the distance between pairs of fetches and the nearest use of the variable on the stack is measured. In our example this would be in the following lines:

In the lines 4 and 5 the variable **a** is loaded, immediately after it is saved. So the distance is 1. It then is used again in line 7 and loaded in line 9. in this case the distance is 2.

The variable **b** is stored in line 10 and reloaded in line 12. The distance again is 2.

After all distances are measured the pairs are ranked by their distance, beginning with the lowest.

For each of the pairs, its evaluated if the variable of interest can be copied to the bottom of the stack using a single stack word<sup>3</sup>, otherwise the pair is ignored and the next is evaluated. If the condition is met and the Stack depth at the point if reuse is 2 or lower (so the variable at the bottom of the stack can be changed on top if needed). If both conditions are met, depending on the status of the stack, a word to duplicate is added in front of the 'use' case and a word to place the result instead of the 'reuse' case is added instead of the 'reuse'.

<sup>1</sup>to be more specific, a @ fetches and returns the cell at the memory address, however in this case a numeric input is added and can be interpreted as the 'name' of the variable.

<sup>2</sup>the ! Stores the cell on top of the stack at memory address.

<sup>3</sup>like DUP(Duplicates the top stack item) or TUCK(Insert a copy of the top stack item underneath the current second item. Further details at the end of this section.)

After we perform this on our first found pair the code would look like this:

```

1  (      ---) 76 LOCAL@
2  (   76 ---) 77 LOCAL@
3  (76 77 ---) *
4  (   75 ---) DUP          \ copy of 75
5  (75 75 ---) 75 LOCAL!
6  (   75 ---) NOP          \ 75 LOCAL@
7  (   75 ---) 8
8  (75 88 ---) /
9  (   76 ---) 76 LOCAL!
10 (      ---) 75 LOCAL@
11 (   75 ---) 76 LOCAL@
12 (75 76 ---) -
13 (   77 ---) 77 LOCAL!
    
```

Now the first of the pairs with a distance of 2 is evaluated. Note there doesn't seem to be a great change in performance or result by choosing one over the other.

```

1  (      ---) 76 LOCAL@
2  (   76 ---) 77 LOCAL@
3  (   76 77 ---) *
4  (   75 ---) DUP
5  (   75 75 ---) 75 LOCAL!(DEAD) \ s .
        below
6  (   75 ---) NOP
7  (   75 ---) 8
8  (   75 88 ---) UNDER
9  (75 75 88 ---) /
10 (   75 76 ---) 76 LOCAL!
11 (   75 ---) NOP          \ 75
        LOCAL@
12 (   75 ---) 76 LOCAL@
13 (   75 76 ---) -
14 (   77 ---) 77 LOCAL!
    
```

becomes DEAD, because its never used afterwards.

The **UNDER** is a non standard word and defined as following:

>R DUP R>

the >R pushes the top element of the data stack onto the return stack.<sup>4</sup> The R< pushes the top element of the return stack onto the return stack.

Finally our last pair is evaluated:

```

1  (      ---) 76 LOCAL@
2  (   76 ---) 77 LOCAL@
3  (   76 77 ---) *
4  (   75 ---) DUP
5  (   75 75 ---) 75 LOCAL!(DEAD) \ *
    
```

<sup>4</sup>Forth uses two stacks,other than most other stack based languages. The stack used so far is the data stack and all programs are performed on it. The return stack is used for storing return addresses and temporary data which would get in the way if cept on the stack [5].

```

6 (      75  ---) NOP
7 (      75  ---) 8
8 (    75  88  ---) UNDER
9 (75 75  88  ---) /
10 (    75  76  ---) TUCK    \ copy of 76
11 (76 75  76  ---) 76 LOCAL!
12 (    76  75  ---) NOP
13 (    76  75  ---) SWAP    \ 76 LOCAL@
14 (    75  76  ---) -
15 (      77  ---) 77 LOCAL!

```

This would be our result after step 4 of the algorithm. However this code still can be optimized. As you can see lines 4 and 5 are not needed for the algorithm to work in its intended way. This is because the value of **a** use in the next step of computation is already located on the step, instead of being stored and loaded again from a register. Lines 11 and 12 can be optimized from **TUCK LOCAL!** to **DUP LOCAL!**. The reason behind this, is the following operation is a subtraction, which is associative. For the same reason the **SWAP** can be eliminated. The last optimization is to delete all **NOP** operations still left in the code. The resulting code would look like this:

```

1 (      ---) 76 LOCAL@
2 (      76  ---) 77 LOCAL@
3 (    76  77  ---) *
4 (      75  ---) 8
5 (    75  88  ---) UNDER
6 (75 75  88  ---) /
7 (    75  76  ---) DUP
8 (76 75  76  ---) 76 LOCAL!
9 (    75  76  ---) -
10 (      77  ---) 77 LOCAL!

```

As you can see, the new code only contains 4 variable references instead of 8 in the beginning and overall is two lines shorter.

Figure 1 shows the results of this algorithm on some example programs. As you can see the efficiency is around 90%.

To this point we implicit assumed the accessibility of the stack to be one or two. What is meant by this is not the actual depth of the stack, rather the depth of variables available for the algorithm to copy to the bottom of the stack. Earlier in this example we set the instruction to check, if a variable could be copied with a single word<sup>5</sup>, to make it available when needed. Furthermore, because we are in a stack based language we can define our own words, to accomplish even deeper access to the stack. However if the depth greater than three, the algorithm tends to produce suboptimal code with many unnecessary stack operations. For the purpose of comprehensibleness the example above only shows stack manipulation with a depth of one.

2) *Global variable elimination*: In contrast to local variable elimination, we now try to eliminate variables throughout

all given program blocks. To achieve this with an algorithm, one would need to analyse the whole program to find global variables matching each other throughout the code. Other than one would think, this is not a trivial task. Meanwhile this still can be accomplished manually and the results can be seen in figure 2. As you can see, not all algorithms presented in the example can be optimized to their fullest, this however can be reasoned with the usage of many frequently changing variables in loops, or repeated usage of recursive function calls.

<sup>5</sup>a word in Forth is equivalent to a function in e.g. C

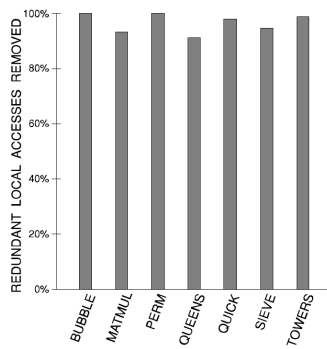


Figure 1. Intra-block stack scheduling removes most redundant accesses to local variables. [3]

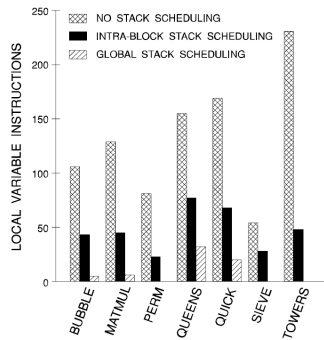


Figure 2. The number of local variable instructions in the compiled code reduces dramatically with stack scheduling. [3]

## VI. CONCLUSION

Stack based languages provide advantages, not covered by other types of programming languages. As they are conceptually simple, automatic code-generation(meta-programming) becomes rather easy. Further more because of the simplicity they are also easy extendible, which is another reason they are frequently used as intermediate languages or as interpreters. Furthermore Stack based programs can be compactly saved in byte code format. However they aren't used as frequently as other languages. The reasons for this could be their exclusive use of RPL(postfix syntax). Further they possess implicit parameter passing. This reduces the readability of programs, not written with good style, by a lot. Also most of the languages are specified to a certain field e.g. PostScript to vector graphics. This limits their attractiveness to new programmers, but those working within the respective field of use.

## REFERENCES

- [1] C. BAILEY, R. SOTUDEH, and M. OULD-KHAOUA. The effects of local variable optimisation in a c-based stack processor environment. In *Proceedings of the 1994 EuroForth Conference*.
- [2] M. A. Ertl. *Implementation of Stack-Based Languages on Register Machines(1996)*. PhD thesis, Technische Universität Wien, Technisch-naturwissenschaftliche Fakultät.
- [3] P. K. Jr. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 6, 1992.
- [4] O. Patashnik. *Designing BIBTEX Styles*, 1988.
- [5] S. Pelc. *Programming forth*, 2003.

- [6] T. Shpeisman and M. Tikir. Gerating efficient stack code for java. Technical report, University of Maryland Department of Computer Science, 1999.
- [7] M. von Thun. *Mathematical foundations of Joy*. Available from the author, 1994.
- [8] M. von Thun. *The algebra of Joy*. Available from the author, 1995.
- [9] M. von Thun. *Joy compared with other functional languages*. Available from the author, 1996.
- [10] M. von Thun. *Lambda calculus vs. combinatory logic vs. Joy — a very very gentle introduction*. Available from the author, 1997.
- [11] M. von Thun. <http://www.latrobe.edu.au/humanities/research/research-projects/past-projects/joy-programming-language>, 2015.



# Evaluation Strategies

Moritz Flucht

Email: moritz.flucht@student.uni-luebeck.de

**Abstract**—Strict vs non-strict, applicative-order vs. normal-order, lazy vs eager—terms commonly used in debate about the evaluation behavior of programming languages. In this paper, the differences and similarities are presented, use cases are named and respective examples are provided. The possibility of improving the efficiency of inherently eager languages by using lazy traits is discussed, as well as *optimistic evaluation* as a further improvement on lazy evaluation.

## I. INTRODUCTION

“At a conference on programming languages you might hear someone say, ‘The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation.’”

(H. Abelson)

Given this quote from Abelson’s classic *The Structure and Interpretation of Computer Programs* [1], also known as the wizard book, you quickly realize there exists an abundance of terms and phrases surrounding the field of evaluation strategies. Some of these are more commonly used than others and while some are synonymous, others differ in nuances depending on who you ask.

This paper seeks to do away with some of the confusion caused by conflicting and inconsistent definitions. For that purpose, in section II, the most commonly used terms are explained and contrasted with each other.

To give an insight into the inner workings of the various evaluation strategies and outline some of their use cases, lazy and eager evaluation techniques are presented in section III and section IV. Their implementations are discussed, their benefits and drawbacks in certain applications are clarified and mitigating strategies illustrated. Also, some practical examples are given that motivate usage in real life software.

As a proposed improvement to the sometimes resource-hogging lazy evaluation, optimistic evaluation is introduced as an alternative approach the the benefits of lazy evaluation in section V.

## II. TAXONOMY

In a field with as broad and rich a spectrum programming languages, researchers seldom reach a mutual consensus regarding common nomenclature. Add to that the rapid progression in the last decades and you are guaranteed to end up with a variety of definitions and interpretations of similar concepts. To mitigate the multiplicity of such interpretations, a hopefully sane and approachable compromised definition of the most common terms is given.

### A. Semantics vs. Evaluators

If you were to begin reading up on the topic of evaluation strategies, you would quickly encounter the terms *applicative-order* and *normal-order*.

As to what the difference between those terms is, one has to look at the parameters of a function and the way in which they are evaluated. Using applicative-order, all function parameters are evaluated before the procedure’s body is entered, whereas normal-order languages *defer* the evaluation until the moment the value is actually required and used [1][2].

Consider, for example, the following definition of a `try` function in listing 1. In Scheme, fully parenthesized prefix syntax is used for all function applications and operators (as for example the boolean `=`). To globally bind a function to a name we use `define` which takes two parameters; the function’s signature and its body.

---

```
(define (try a b) (if (= a 0) 1 b))
```

---

Listing 1: Applicative-order and normal-order in Scheme

Running `(try 0 (/ 42 0))` would yield a division-by-zero error in an applicative-order language because the parameters are evaluated before the function is applied. Using normal-order evaluation, the second parameter `b` is never evaluated because of the conditional statement and, thus, the function returns `1`.

The terms of applicative-order and normal-order are closely related to eager and lazy evaluation strategies. They are often, though not always, used synonymously. Nevertheless, it is possible to make a distinction, as is done in [1]. It is suggested that applicative/normal-order be used to describe the semantics of a programming language, whereas eager/lazy would refer to a specific evaluation strategy—the former thus being more or less independent from the latter. We will see, for example, that while Haskell is an inherently normal-order language, it can both be evaluated lazily (section III) and optimistically (section V) without changing its semantics. Another illustration would be an evaluator for a normal-order language which evaluates subexpressions eagerly on another thread and throws away the result if it is not needed.

### B. Strict vs. non-strict

Having a second glance at listing 1, it becomes obvious that the `if`-clause very much behaves like a function evaluated in normal-order. That is to say, it is *non-strict* in its arguments. Similarly, a procedure is said to be *strict* in its parameters, if they are evaluated before the body of the functions is entered.

So, the terms strict and non-strict are used when referring to parameters and procedures, whereas applicative-order and normal-order are used when referring to languages.

Now, knowing this, have another look at the quote in the introduction (section I) and see whether it makes any more sense.

### III. LAZY EVALUATION

According to [1], *lazy evaluation* refers to a class of evaluation strategies that will delay the evaluation of an expression that is being passed to a function as a parameter until the last possible moment. That time has typically come when the value of the given expression is actually needed, e.g., the respective variable is used in a primitive operation. Notably, a parameter might thus never be evaluated, if it is not used in the function’s body (as seen in section II-A).

A common example of an advantage is the possibility of defining a conditional function yourself (see listing 2) [1].

---

```
(define (unless condition _then _else)
  (if condition _else _then))
```

---

Listing 2: Defining `unless` in Scheme

This is only viable because the procedure is *non-strict* in its arguments, so the function’s body is entered before the parameters have been evaluated. This way, the code given in `_then` and `_else` is only executed, if the premise of the corresponding condition holds.

#### A. Call-by-name vs. call-by-need

As mentioned previously, the definitions for terms regarding evaluation strategies are, at best, used fuzzily. In the wizard book<sup>1</sup> *call-by-name* refers to a lazy evaluation strategy, where a value is computed only when it is needed. In contrast to *call-by-need*, that computation is done *every time* a value is called for. Intuitively, this is similar to code rewriting macros as found, for example, in C. Apart from the difference in underlying compiler mechanisms (see section III-D), there is also a distinction to be made in semantics. Consider a (not very useful) macro for multiplication (see listing 3).

---

```
#define mult(x, y) x*y
```

---

Listing 3: Multiplication macro in C

Calling `a = mult(2+3, 1+4)` would of course yield `a = 9` because of the operator precedence of `*` over `+`, whereas true call-by-name would result in the more intuitive `a = 25`. So, with correct use of parentheses, call-by-name can be *simulated*.

To circumvent the repeated evaluation seen in call-by-name and the thereby caused computational overhead, with *call-by-need* the initially computed value is stored and referred to instead—if the value is required a second time. Thus, call-by-need can be thought of as a *memoized* version of call-by-name. This is, of course, only practical when pure functions are used,

<sup>1</sup>Colloquial pseudonym for SICP [1] because of the wizard used on the front cover.

where a call to a function with the same parameters yields the same output and does not result in any side effects. In that case, however, it is likely to cause a considerable speedup, which is why call-by-need is used by default in most of purely functional languages like Haskell, Miranda and Clean.

Why use the less efficient call-by-name, if you can use the memoized version? Often the use case is an application where side effects are a wanted feature of repeated evaluation or immutability cannot be guaranteed (see section IV-B).

#### B. Motivation

Quoting the most cited paper in the field yields the most general motivation for lazy evaluation:

Lazy evaluation makes it practical to modularize a program as a generator that constructs a large number of possible answers, and a selector that chooses the appropriate one [3].

Because of the deferred nature of the evaluation, lazy evaluation is mostly used in contexts of functional programming languages. That is to say, in languages which respect the paradigm of pure functions—functions that don’t destructively update an entity, i.e. execute without causing side effects. In this context, variables refer to immutable values and, because of this, the program usually does not deal with mutable states. This way, the order of evaluation is not essential to the computed result, which is important for lazy evaluation, since, of course, the execution is inherently not in order of code but in order of usage.

#### C. Working on infinite structures

One of the most often mentioned advantages of lazy evaluation is the fact that, because of the non-strict nature, the programmer can work with infinite data structures.

---

```
magic :: Int -> Int -> [Int]
magic m n = m : (magic n (m+n))
```

---

Listing 4: Infinite lists in Haskell

Consider listing 4 where we define a function `magic` that recursively computes an (infinite) list of the Fibonacci numbers. Of course, calling `(magic 0 1)` in an eager evaluation context would yield no result at all, as the recursive computation would continue until the system runs out of memory, i.e. causing a stack overflow. However, with lazy evaluation the above piece of code can be executed without further ado, since the call would return something like a *promise* that a value will be produced if it is actually needed. In that, we receive a reference to a deferred evaluation (see section III-D).

---

```
doubleList :: [Integer] -> [Integer]
doubleList [] = []
doubleList (n:ns) = (2 * n) : doubleList ns
```

---

Listing 5: Operations on infinite lists in Haskell

Naturally, we cannot expect an operation that requires all the values in the list to terminate. That means, we cannot

print or sort an infinite list, but we are able to run operations like `doubleList` on them (see listing 5) [4]. The result of `doubleList (magic 0 1)` can then be passed to another function that only does its work on the elements it requires.

In functional programming, it is quite common to work with the aforementioned generate-and-filter paradigm, while hoping lazy evaluation will avoid an overly large intermediate list.

#### D. Implementation

A group working on Algol 60 coined the term “think” [5], which is one of the main concepts of any laziness implementation. Arriving at a parameter expression, as stated before, that expression is not immediately evaluated and passed (see section IV), but delayed instead. For that purpose, it is safely tucked away in a *think* which contains all the information necessary to compute an actual value, if need be. Therefore, the think has to contain not only the expression but also the environment in which the call is evaluated (namely bound variables). It is conceptually similar to a *closure* with the addition of the ability to save the returned value.

If the time finally comes and the value is needed, the think is *forced* and the value computed. In listing 6 an example think wrapper for Python is presented [2].

---

```
class Think:
    def init (self, expr, env):
        self._expr = expr
        self._env = env
        self._evaluated = False
    def value(self):
        if not self._evaluated:
            self._value = force_eval(self._expr,
                                     self._env)
            self._evaluated = True
        return self._value
```

---

Listing 6: Building thinks in Python

Upon creation, it receives the expression whose evaluation is to be deferred, which, at this point, can be thought of as in form of an *abstract syntax tree*. The think also receives the environment in which the passed expression should be evaluated in. It is initialized with its evaluation state set to `False`. The class exposes a `value` method which will produce the value of the expression, either by forcing the think or returning an already computed value.

Forcing can stem from three situations:

- *Passing a value to a primitive function.* If a primitive operation is being evaluated, it requires the operands to be fully evaluated, i.e., it is strict, and therefore forces any associated thinks.

---

```
doubleMe x = x + x
let y = 40 in (doubleMe 1) + y
```

---

Listing 7: Haskell: Forcing primitive Operations

In listing 7 the `(doubleMe 1)` call is forced immediately when the outer `let` is forced.

- *Using a value in a condition.* In order for the program to correctly determine its control path, the boolean expression

used in a conditional statement needs to be evaluated and its value made known to the runtime component. Though, depending on the language and the actual condition, not all operands are necessarily evaluated to make a decision on branching (see section IV-B1).

- *Using a value as an operation.* In functional languages where functions are “first class citizens” [6], they can be passed as parameters and assigned to variables. Should such a variable then be applied as a function, its value is needed and the respective think is therefore forced.

#### E. Handling Exceptions

Because the order of evaluation cannot always said to be transparent, comprehending how and *when* an error was caused can become a problem. In Haskell exceptions can only be caught inside of the I/O monad part of code, so that the evaluation order is well-defined. Consider, for instance, a pure function that was able to catch and distinguish different exceptions. If this were the case, its result would be dependent on the order of execution—which is up to the runtime component and is due to change, possibly between different executions of the same system. Because a pure function’s result should only be affected by its arguments, catching exceptions is not allowed in the pure segment of the code. You are encouraged to leave throwing and catching exceptions to the impure portion of your code instead and use the optional types `Either` and `Maybe`.

The implementation of exceptions on the other hand, is relatively straight-forward [7] and only differing from that commonly found in eager evaluation by specific optimizations (see section VI-A). When sighting a `catchException`-statement, an “exception handler frame” is pushed onto the stack. Should an error be encountered, frames are torn off the stack until the first such handler is found, whereafter the associated callback is executed.

#### F. Optimization

A significant amount of research has been dedicated to the efficiency drawbacks that come with lazy evaluation techniques. Acceptable execution time is achieved by means of *static analysis*. The idea is to look for expressions that are safe to evaluate immediately and, thereby, refrain from allocating a think. This resulted in a mean speedup of factor two [8] for a given set of test programs.

In some cases, *strictness analysis* can expose more locations, where such optimization is viable, e.g., for lazy lists [9]. This is done by analyzing how often an expression is likely to be evaluated by its environment [10].

Another approach is *cheap eagerness* where expressions that are probably cheap in computation time and safe to execute are evaluated before their value is needed [11]. So while strictness analysis attempts to find sub-expressions that were going to be evaluated anyway, cheap eagerness looks for those expressions that would not cause any serious cost, even if they were not evaluated using pure call-by-need. The inner workings of these optimizations are complex and implemented differently

depending on the programming language and the compiler [12]. All use data and control flow analysis to determine, for example, whether or not all variables needed for evaluating a given expression were already computed—in which case the expression can be speculatively evaluated (see section V). For further information and materials see section VI-A.

#### IV. EAGER EVALUATION

As one might imagine at this point, *eager evaluation* (a term probably first used in [13]) describes a class of evaluation strategies, where a parameter expression's value is known before the function's body is entered. In particular, it is computed as soon as it is assigned to a variable. This technique is commonly used by conventional programming languages like C, Java, Scala and Python—to name only a few. This way, the programmer can be sure at which point his code will be evaluated and plan accordingly. Anticipating how other expressions are affected when evaluating a specific sub-expression is essential when dealing with side effects.

##### A. Call-by-value and call-by-sharing

Again, inconsistent use of terminology across different programming communities hinders understanding what is actually meant. Usually *call-by-value* refers to any evaluation strategy where a sub-expression is fully evaluated before its value is bound to its respective variable. In the context of method parameters, this is usually done by copying the value and thereby allocating new memory. By that reasoning, any changes made to the variable from within an applied function, are not visible from outside that function (as they were done to the copy). This is the case, for example, for primitive data types in Java (see listing 8).

```
class Operation {
  void change(int data) {
    data=data+42; }

  public static void main(String args[]){
    Operation op = new Operation();
    int x = 0;
    System.out.println("before "+x);
    op.change(x);
    System.out.println("after "+x); } }
```

Listing 8: Call-by-value in Java.

Of course, both times  $x$  will yield zero. Unfortunately, some will argue that Java uses call-by-value for object references, too. While this may be technically true (if you assign a new object to a passed reference variable, the variable outside of the current scope will not be subject to the change), it is reasonable to make a semantic distinction. A mechanism where changes to a passed parameter are observable from outside the applied

function, are called *call-by-sharing* [14]. This requires the passed object to be mutable to make a difference.<sup>2</sup>

##### B. Lazy Evaluation in applicative-order languages

In most applicative-order languages, there already exist constructs that mimic non-strict behavior, namely `if`-clauses. Not only are the arguments (code blocks) evaluated after the premise is evaluated (see section III), but the evaluation of the condition itself is dependent upon being evaluated lazily.

1) *Short circuit evaluation*: This is what is commonly referred to as *short circuit evaluation*. The evaluation of boolean operators is done in left-to-right order and yields a result as soon as the premises value can be determined. Arguments that have not been visited up to that point are not evaluated at all. This can sometimes lead to unintended behavior, if the programmer is not aware that such mechanisms are in place and depends on an argument to produce a side effect.<sup>3</sup>

2) *Picking the goods*: Lets have a look at another example to see how an applicative-order language like JavaScript can still benefit from lazy concepts. Consider a list of values (see listing 9) that needs to be filtered according to some given function and limited to a number of results thereafter. With the JS framework Lodash [15] this can be accurately modeled with the provided pipelining functions, where the actual result is computed only when the `value()` function is called.

```
function ageLt(x) {
  return function(person) {
    return person.age < x; }; }
var people = [
  { name: 'Anders', age: 40 },
  { name: 'Binca', age: 150 },
  { name: 'Conrad', age: 200 },
  { name: 'Dieta', age: 70 },
  { name: 'Finnja', age: 30 },
  { name: 'Geras', age: 130 },
  { name: 'Hastig', age: 20 },
  { name: 'Ickey', age: 200 }];
var alive = _(people)
  .filter(ageLt(100)).take(3).value();
```

Listing 9: Lazy pipelining in JavaScript with Lodash

If you were to use eager concepts for pipelining, `filter()` would have to loop through all eight objects in `people` when, in the end, only three are really needed. This is, of course, because `filter()` is completely oblivious to the chained `take()` method, since its value is fully evaluated before passing it on. Taking advantage of lazy evaluation, the execution can be delayed until all pipelined functions are registered and can be applied to each element individually, stopping when the third filtered element is received. This way only five elements need to be processed.

<sup>2</sup>It is worth mentioning that there exists another term called *call-by-reference* which is not listed here because its meaning varies even more greatly in diverse communities. This is to be expected, as *reference* is not dogmatically defined. While, for example, a Java user would refer to the mechanics of passing an object reference, in a C world, reference is often used as a synonym for a pointer and would therefore allow to change the way variables are dereferenced *outside* of the current scope.

<sup>3</sup>This type of programmer is, of course, frowned upon.



Python developers also recently discovered lazy evaluation as a means to saving computation time. In Python 2.x, for instance, the `range()` function would return a fully evaluated list, whereas in Python 3.x a special range object is returned. If a value of that range is actually needed, it is only computed at that moment, therefore avoiding computation of unused values.

3) *Scala*: Scala is another applicative-order language that offers some lazy features for convenience. Both call-by-name and call-by-need mechanics can be exploited [16]. A `def` can be used to define a variable as a parameterless method. This way, that `def` is then only evaluated when it is used (call-by-name) and not at the time the variable would be initialized normally using a simple `val`. Additionally, `val`-expressions can also be made to behave lazily by prepending the keyword `lazy`. In contrast, though, these are evaluated at most once, whereafter their computed value is stored and reused. This makes sense, as a `val` is immutable in Scala.

## V. OPTIMISTIC EVALUATION

As we have previously established, lazy programs can make a programmer's life easier. But, as often is the case in computer science, the trade-off for concise and intuitive code, sadly, is performance in both time and space. A speculative evaluation strategy, titled *optimistic evaluation* [17], attempts to partially alleviate this drawback. It was implemented as an extension to the Glasgow Haskell Compiler but was not, as of this writing, merged into to the main branch, "due to its significant complexity, and the fact that performance improvements tended to be inconsistent." [18]

### A. Motivation & Idea

When using lazy evaluation (see section II.A), for every function's argument a thunk is allocated in the heap. We've already seen some compiler counter measures to overcome this performance overhead (see section III-F). However, these approaches have to be *conservative* in nature, meaning thunks that might never be evaluated have to be kept because they are not automatically "provably unnecessary". Given a handful of realistic test programs, in most cases, the generated thunks are either "always used" or "cheap and usually used" [17]. It is therefore evident, that a lot of thunk overhead can be avoided, if thunks that are "always used" were to be evaluated eagerly, with the rest remaining lazy. In order to achieve this, a run-time component is introduced that decides which chunks should be evaluated using call-by-value. It also contributes an abortion mechanism that can suspend a speculative execution of a thunk, in case of too long a computation.

### B. Compiler modifications

In order to provide the afore mentioned aspects, the back end of the compiler has to be slightly modified. When compiling a program's code, the compiler will run its course as usual, applying analyses, optimizations and transformations as it goes along. It's only after this process, that the modifications come in. [17] also supplies an operational semantics that works with a boiled down version of the Haskell language. This

simplified rendition still possesses the same expressiveness, but allows for much easier demonstration of the evaluation strategy. The program is therefore transformed in such a way, that all thunks are allocated by `let` expressions and the following modifications are less complicated to implement. In effect, this component works as a code generator for the intermediate language.

1) *Let expressions*: For every `let` expression `let x = <rhs> in <body>` the compiler will insert a code snippet that is essentially a switch (see listing 10).

---

```
if (LET42 != 0) {
  x = value of <rhs>
} else {
  x = lazy thunk to compute <rhs>
    when needed
} evaluate <body>
```

---

Listing 10: Code generated `let` switches

Now, by setting LET42 accordingly, either a thunk is allocated or the right-hand side of the `let` expression is evaluated speculatively. In order to circumvent code bloat, the right-hand side is extracted into a new function. The behavior of the switches are adjustable during run-time, starting in a configuration in which all `lets` are speculated.

2) *Abortion*: This evaluation strategy is called *optimistic* because it will initially assume a given thunk should be speculated, in any case. Of course, if that thunk turns out to be too expensive, a change in strategy is needed. With the *abortion* mechanism, a speculative evaluation can be suspended and, if needed, continued when the thunk is actually forced. To do so, a suspension which contains the started computation is allocated in the heap. This implementation is equivalent to that used for dealing with asynchronous exceptions, where a thunk is "frozen" [19].

---

```
if (SPECDEPTH < LIMIT42) {
  SPECDEPTH = SPECDEPTH + 1
  x = value of <rhs>
  SPECDEPTH = SPECDEPTH - 1
} else {
  x = lazy thunk for <rhs> }
evaluate <body>
```

---

Listing 11: Recursion counters extension for `let` switches

3) *Chunky evaluation*: Considering self-referencing or recursive data structures, an *abortion* would quickly occur because of too much computation time spent in a particular `let` expression and, therefore, not make use of the often more efficient speculative evaluation. To circumvent this, a *chunky* evaluation technique is used. The idea is to switch over to lazy evaluation after a given number of recursions and, thus, gaining the speed-up of speculative evaluation while still retaining the ability to work with infinite data structures. This is implemented with the means of a simple recursion counter (see listing 11) that keeps track of the current nested speculation depth. This snippet is an extension of the `let` switch shown previously.

Again, the allowed speculation depth can be fine-tuned for each `let` during run-time. Arguably, the deeper nested a

speculation is already, the less probable is a useful computation during the next speculative evaluation.

4) *Dealing with errors:* Because code is evaluated speculatively and, most probably, not in any meaningful order, it is unacceptable to halt the execution, should an error (or exception) occur during that evaluation. Instead, the error is caught and prevented from escaping at that point. The variable of the corresponding `let` expression is then bound to a thunk that will re-raise the caught exception, given that the initial expression that caused the thunk allocation in the first place is even forced.

5) *Dealing with I/O:* Certain I/O operations are marked *unsafe* in the otherwise pure language Haskell, i.e. their evaluation may produce side effects. Were, for example, an expression speculatively evaluated that wrote out a character to stdout, that character would be visible to the user, whether or not the expression would actually be reached. Obviously this is no tolerable behavior, which is why speculative evaluation is not allowed for these functions.

### C. Run-time component

The configuration for the `let` switches has to be set in such a way that an expression is speculated only if the amount of work that could possibly be wasted on the evaluation outweighs the cost of a lazy thunk allocation. This is implemented by the means of an online profiling component. A so-called *wastage quotient* is computed, which is the amount of work wasted divided by the times an expression has already been speculated. These, of course, are really just heuristical means and will always return mixed results, depending on the practical application.

Optimistic evaluation was able to produce a mean speed-up of 15%, with no tested program slowing down by more than 15%. These are encouraging figures, considering the maturity of the GHC.

## VI. CONCLUSION

We have seen that both lazy and eager evaluation come with benefits and drawbacks. While eager evaluation is likely to evaluate expressions that are never actually used, the lazy approach produces a considerable computational overhead by the additional bookkeeping necessary to keep track of what has already been evaluated. The terminology presented and used in this paper is listed again in table I. Although research in this field has come a long way, optimizations that make mixing the two strategies viable are still a topic of recent discussion. In fact, we saw an example (see listing 9) where using lazy strategies in the context of an applicative-order language would lead to a significant speed-up.

Therefore, we may conclude that both strategies can enhance the programmers experience and the program's efficiency if used symbiotically.

### A. Further reading

Concerning the semantics and the implementation of lazy evaluation strategies, not all relevant aspects could be discussed.

TABLE I  
TERMINOLOGY OVERVIEW

Evaluator	Language	Procedures	Strategies
eager	applicative-order	strict	call-by-value, call-by-sharing
lazy	normal-order	non-strict	call-by-name, call-by-need

Especially, the commonly used technique of graph-reduction (as used in the GHC with the STG-machine) would have went well beyond the scope of this paper [20].

## REFERENCES

- [1] H. Abelson, G. J. Sussman, and J. Sussman, *The Structure and Interpretation of Computer Programs*. Cambridge, Mass., USA: MIT Press, 1985, vol. 9, no. 3.
- [2] D. Evans, "Introduction to computing," 2011.
- [3] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. April 1989, pp. 98–107, 1989.
- [4] Duplode, "Haskell/Lists II," 2015. [Online]. Available: [https://en.wikibooks.org/wiki/Haskell/Lists\\_II#Infinite\\_Lists](https://en.wikibooks.org/wiki/Haskell/Lists_II#Infinite_Lists)
- [5] P. Z. Ingerman, "Thunks: a way of compiling procedure statements with some comments on procedure declarations," *Communications of the ACM*, vol. 4, pp. 55–58, 1961.
- [6] R. Burstall, "Christopher Strachey—Understanding Programming Languages," *Higher-Order and Symbolic Computation*, vol. 13, no. 1-2, pp. 51–55, 2000.
- [7] A. Reid, "Handling exceptions in haskell," *submitted to Practical Applications of Declarative Languages (PADL'99)*, 1998.
- [8] J. Fairbairn and S. C. Wray, "Code generation techniques for functional languages," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86. New York, NY, USA: ACM, 1986, pp. 94–104.
- [9] P. Wadler and R. J. M. Hughes, "Projections for strictness analysis," *Proc. of a conference on Functional programming languages and computer architecture*, no. September, pp. 385–407, 1987.
- [10] P. Wadler, "Strictness analysis on non-flat domains," in *Abstract interpretation of declarative languages*. Ellis Horwood Chichester, UK, 1987, pp. 266–275.
- [11] K.-f. Fax, "Cheap Eagerness : Speculative Evaluation in a Lazy Functional Language," in *ACM Sigplan Notices*, vol. 35, no. 9. ACM, 2000, pp. 150–161.
- [12] A. L. D. M. Santos, "Compilation by Transformation in Non-Strict Functional Languages," Ph.D. dissertation, 1995.
- [13] R. Bubenik and W. Zwaenepoel, "Performance of Optimistic Make," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '89)*, vol. 17, pp. 39–48, 1989.
- [14] B. Liskov, E. Moss, A. Snyder, R. Atkinson, J. C. Schaffert, T. Bloom, and R. Scheifler, *CLU reference manual*. Springer-Verlag New York, Inc., 1984.
- [15] J.-D. Dalton, "lodash," 2015. [Online]. Available: <https://lodash.com/>
- [16] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*, 2nd ed. Artima Inc, 1 2011.
- [17] R. Ennals and S. P. Jones, "Optimistic evaluation: An adaptive evaluation strategy for non-strict programs," in *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '03. New York, NY, USA: ACM, 2003, pp. 287–298.
- [18] "The 2005 GHC survey." [Online]. Available: <https://www.haskell.org/ghc/survey2005-summary>
- [19] S. Marlow, S. P. Jones, A. Moran, and J. Reppy, "Asynchronous exceptions in haskell," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 274–285.
- [20] S. L. P. Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming*, vol. 2, no. 02, p. 127, 1992.

# Logic Programming

Gilian Henke

**Abstract**—The idea to use the concepts of logic in the programming language is an interesting concept, which today has just a small specialised usage. This paper describes the foundations for logic programming for itself, which includes basic definitions, unification and SLD-Resolution. Also we will discuss some of the problems which exist in the implementation of logic programming languages. To further enlarge upon this theme examples in the most common logical language Prolog are given.

## I. INTRODUCTION

Logic Programming is based on the idea to have a fully declarative programming language, where the programmer just has to write what the program should do, without him caring about in which way the program will be realised. Therefore this program will also have no side effects. And also everything is written in a clear logical way, with a clear reference to the mathematical principles of logic. Also this language shall be expressive, the programmer should be protected from common mistakes, a high enough abstraction and the program itself is efficient through its implementation. In the logic programming languages many of this are tried to be used in, but to make things easier for the programmer, in nearly every logic programming language there are some procedures which subvert these goals. To fully apprehend the logic programming languages the basics of the logic programming are explained. And to better understand these abstract concepts examples in the most relevant language Prolog are also presented.

## II. HISTORY

The concept of logic programming is based on the lambda calculus, developed in the 1930s. In these years the idea to use the clausal form of logic to represent programs came up. But the usage in these times was vastly different then the usage today. The logic programming known evolved from the automation of theorem proving by Herbrand and from the usage in the artificial intelligence. In the programming of artificial intelligence was a conflict between declarative and procedural procedures. On the procedural side the language Planner was developed. In the 1970s Kowalski tried to rebase this in pure logic. He developed SLD resolutions and together with Colmerauer he developed the most important logic programming language Prolog, which until today is the most important logic programming language. [1]

## III. PRINCIPLES

The algorithm of a logic program can always be seen as two parts: logic and control. The logic part determines the solution, which can be derived from it. This is the part for which the programmer is responsible. The control part is about in which way this solution can be found. This part is not changing for

a given Prolog compiler. But the logic part alone determines the solutions. A change of the control part will not change the solution. But you can use different compiler on the same logic program to optimize the solution process. [9]

## IV. LOGIC

Logic Programming is based on First Order Predicate Logic. In a logic program every line has to be a so called Horn Clause. A Horn Clause is clause, i.e. a disjunction of literals, which contains at most one positive literal. These can be further divided into definite clause with exactly one positive literal, unit clause with just one positive literal and no negative ones and goal clauses with no positive literals. In logic programming these are written as seen in table ?? . Hereby  $A$  is called head,  $B = B_1 \wedge B_2$  is called body and the  $G_i$  are called subgoals. A rule can also be read in the following way: To show that  $A$  holds true, show that all  $B_1, \dots, B_n$  hold true. The rules and Facts are written in a database, the program. On this database a query can be applied to get an answer from the program. Going by the definition above this answer can just be yes or no. But above we have not used predicate logic to describe our program. In the predicate logic a literal is an atomic formula. The atomic formula consists of a predicate symbol  $P$  and terms  $t_i$ ,  $P(t_1, t_2, \dots, t_n)$ . Every term can either be a constant, a variable or a compound term. Constants are particular individuals, which can be for example integer or strings. Variables are a single individual but none in particular. There can be multiple instances of the same variable in one clause. A compound term consists of a functor and its arguments. If we would write the Horn Clauses as a First Order Predicate Logic Formula we also would have to add the quantifiers. In the case of a rule with an variable we would have to write an universal quantifier with this variable in front of it. If we then have a Query with a variable in it, this means that we want a instantiation of this variable, so that the Query evaluates to true. [3], [8], [9]

## V. CONTROL

The easiest to understand implementation of the control part is done with Proof-Trees, which are also called Search-Trees or Execution-Trees. A Search-Tree displays the solution as a tree, where the root is the goal, and its children are the subgoals. Each subgoal can in the same way have children which then are calculated with the rules. When a leaf is reached and this leaf is a fact of the program this path leads to a result. The search in this tree can be implemented with backtracking, but also every other search strategy can be used.

### A. Substitution

Let  $X$  be a variable and  $t$  be a term. For a given set of tuples  $(X, t)$  the substitution  $\theta$  denotes that if  $\theta$  is applied to a

Normal name	Normal logic	Logic programming	Logic programming name
Definite clause	$A \vee \neg B_1 \vee \neg B_2 \vee \dots \vee B_n$	$A \leftarrow B_1 \wedge B_2 \wedge \dots \wedge \neg B_n$	Rule
Unit clause	A	$A \leftarrow$	Fact
Goal clause	$\neg G_1 \vee \neg G_2 \vee \dots \vee G_n$	$\leftarrow G_1 \wedge G_2 \wedge \dots \wedge \neg G_n$	Query

term  $s$  every  $X_i$  is replaced by its corresponding  $t_i$ . Then  $s\theta$  is called an instance of  $s$ .

A Substitution is basically the mapping from variables to terms. Terms can as described earlier be constants, variables or compound terms. It is possible to substitute a variable to itself. Also multiple substitutions can be composed, so that the substitution  $\theta = \theta_1\theta_2$ .

### B. Unification

Let  $s$  and  $t$  be terms. If there exists a Substitution  $\theta$  so that  $s\theta = t\theta$ , then  $\theta$  is called the unifier of  $s$  and  $t$ . This is also called that  $t$  and  $s$  unify.

For a given  $s$  and a given  $t$  it is always possible to calculate  $\theta$  if it exists. But a  $\theta$  does not always exist. In the case of logic programs, if the unifier applied to the term evaluates to true, then the unifier is called the solution for the term. And if there exists no unifier for two terms then there exists no solution. The most common used algorithm to compute the unifier is the Herbrand Unification Algorithm. [5]

### C. Resolution

The resolution in logic programs is based on the modus ponens from the traditional logic. For given rules  $A \leftarrow B$  and  $B \leftarrow C$  it is possible to derive  $A \leftarrow C$  from this. Therefore to show that  $A$  holds true it is no longer necessary to show that  $B$  is true, it has just to be shown that  $C$  holds true.

### D. SLD-Resolution

The combination of unification and resolution results in the SLD-Resolution. SLD stands for selecting a literal, using a linear strategy, restricted to a definite clause. This is the formal correct way of building Search-Trees. Given a goal clause, which contains the literal  $\neg B_i$  and a rule which contains  $A$ . When  $B$  unifies with  $A$  the SLD-Resolution gives another goal clause where  $\neg B_i$  is replaced by every negative literal from the rule. After that the unifying substitution is applied to the output. The complete way for a trivial way to compute this is shown in Algorithm 1.

In this algorithm we have a query  $Q$  which has to be resolved and the substitution  $\sigma$  which in the end is the solution for the query. The query can be split in subgoals, and we try to eliminate the subgoals one after another. In the beginning a subgoal has to be chosen, the order in which these are chosen does not matter for the correctness of the solution, but in the case of multiple solutions this determines which is chosen first, and it can in some instances prevent problems, which are shown later. If the chosen goal is a fact it can be eliminated. If there exist no possible unification for  $G_i$  then the Resolution fails. If there are multiple substitution possible than this trivial algorithm expects that a correct one is chosen or it would output a failure. This problem would in more complex Resolutions

be resolved with the usage of recursions. When a substitution is found,  $G_i$  will be replaced. Afterwards the substitution will be applied to the Resolvent and to the substitution  $\sigma$ . This continues until we get a failure or the Resolvent is empty, which means we have found a substitution  $\sigma$  which applied to  $Q$  evaluates to true. [1]

**Data:**  $Q=G_1, G_2 \dots G_n$

**Result:** substitution  $\sigma$  and failure

Resolvent =  $Q$ ;

$\sigma = \{ \}$ ;

failure=false;

**while** Resolvent  $\neq \{ \}$  **do**

    select  $G_i \in$  Resolvent;

**if**  $G_i = \text{true}$  **then**

        delete  $G_i$ ;

        continue;

**end**

    select rule  $A \leftarrow B_1 \dots B_m$  where  $A$  and  $G_i$  unify with

**if**  $A$  does not exist **then**

        failure = true;

        break;

**end**

    replace  $G_i$  with  $B_1 \dots B_m$ ;

    apply  $\theta$  to Resolvent;

$\sigma = \theta\sigma$ ;

**end**

return  $\sigma$ , failure;

**Algorithm 1:** SLD-Resolution

## VI. PROBLEMS

With this definitions there are some problems which persists while using a fully declarative logic programming language. In Prolog for example these were averted with the addition of some procedural strategies. An always persisting problem is the non-determination of Prolog-programs. If a program has more than one solution it is never determined which solution will be selected. This is further complicated with a high reliance on the order of rules and facts in the database. If this order is changed the complete run of a program can be changed.

### A. Negation as failure

In the standard definition of logic programs the negation is not implemented. For normal usage a not can be build in the way, that this goal succeeds if every path fails. This for itself would not be a problem but combined with completeness and termination, this leads to problems. But the hereby constructed not is not the same as the negation of a variable in logic. For most cases this can be ignored, but there are some cases in which this makes a difference. It is possible to build a not,



which is equivalent to the negation, as long as the negative formula is ground, i.e. the formula does not contains variables. [6]

### B. Completeness

A given number of rules is complete, if they are consistent, i.e. it is not possible to derive a contradiction, and extensions of it are not consistent. For example there has not been found a way to implement the negation in such a way that it is complete and efficient. [7]

### C. Termination

It is easily possible to program a program which does not terminate even if the answer could be easily computed. Just the change of some lines in the code, which on the logical side do not change anything can trap the program in an endless loop. In this case we would have a search tree with its left side infinite and a never reached right side, which is finite. An example for this is mentioned later. In general problems with the termination exist then when the body of a rule uses a variable which has not been used in the head of the rule. Programs which frequently use this rule are often of the form "Divide and Conquer" or "Generate and Test". Also it is really difficult to prove that a logic program terminates for every given input. For nearly every program there exist a certain way to attain this. Even the programs which are included in a standard Prolog application can be manipulated to not terminate for a given input. [2]

## VII. PROLOG

The by far most common used logic Programming language is Prolog. Before we look on some examples we have to get to know the syntax used in Prolog. For the most part it is the same as the definition above. The small changes are : – instead of ← and . instead of ∧. Constants are written beginning with a small letter and variables are beginning with a capital. The Query is written in the form ? –  $G_1, \dots, G_2$ , where  $G_i$  are terms.

Below are two Hello World programs in Prolog. The first example in Listing VII is the more generalised Version of the second one, which with minimal adjustments can run in any logical language. The second one is using predefined methods which are Prolog exclusive. If we look on the first program we see that the first line is a fact and the second line is the query to get its solution as a solution on the console.

```
hw(helloworld).
?-hw(X).
```

Listing 1: Hello World

```
?- write('Hello_world!').
```

Listing 2: Hello World in Prolog

Next in Listing2 an extended database which not only contains facts but also rules. This will be database used for the following examples. Now too look further on the database.

This database represents a basic family structure, where the fact parent(sophie,frank) mean that the first person, sophie, is a parent of the second one, frank. The rules are defined analogue. For example the rule grandparent(X,Z) means that X is the grandparent of Z. This then holds true if there exists an Y, so that X is the parent of Y and Y is the parent Z.

Some queries which could be used on this database would be ?-parent(steve,ben) which is true, or ?-parent(ben,steve), which would be false. The addition of a variable would lead to Queries like ?-parent(claire,X), which would lead to X=ben and as an alternative solution X=sophie.

```
parent(sophie,frank).
parent(sophie,gary).
parent(steve,ben).
parent(steve,sophie).
parent(claire,ben).
parent(claire,sophie).
parent(alice,carl).
parent(ben,carl).
parent(tom,frank).
parent(tom,gary).
grandparent(X,Z):-parent(X,Y),parent(Y,Z).
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(Z,Y),ancestor(X,Z).
```

Listing 3: Example Database

### A. Search Tree

For the listing 2 and the query ?-grandparent(X,carl) we get with a trivial search the search tree in Figure 1. Trivial search because, we conduct a depth-first traversal, with no optimization. The root of the search tree is the goal. The Nodes are the subgoals. There exists an edge between two Nodes if they unify. Leaves are success nodes, if a fact has been reached or failure nodes if the selected goal can not be reached anymore.

In this example the goal grandparent(X,carl) is first redefined to the subgoals parent(X,Y) and parent(Y,carl), with the usage of the resolution. Then the substitution with X=sophie is applied. This guess is arbitrary, any other guess for X could have been done here. It is done because the first line of the database is parent(sophie,frank). And this is also the way in which most Compiler would evaluate this. With the same argumentation the second substitution is applied. Because parent(frnk,carl) can not be fulfilled this branch leads to failure. Therefore we backtrack and try another substitution for Y, but this also leads to failure. Now all possible substitutions for Y have lead to failure, therefore we try another substitution for X. With this substitution we come to a solution where all subgoals are fulfilled. We get the solution X=steve. It is possible for extend the search to get other possible solutions. It is also possible to use another search strategy to get another solution first. [9]

### B. Example

1) *Non-Determination*: The Query ?-grandparent(X,gary) would give for database 2 the solution steve. If you change

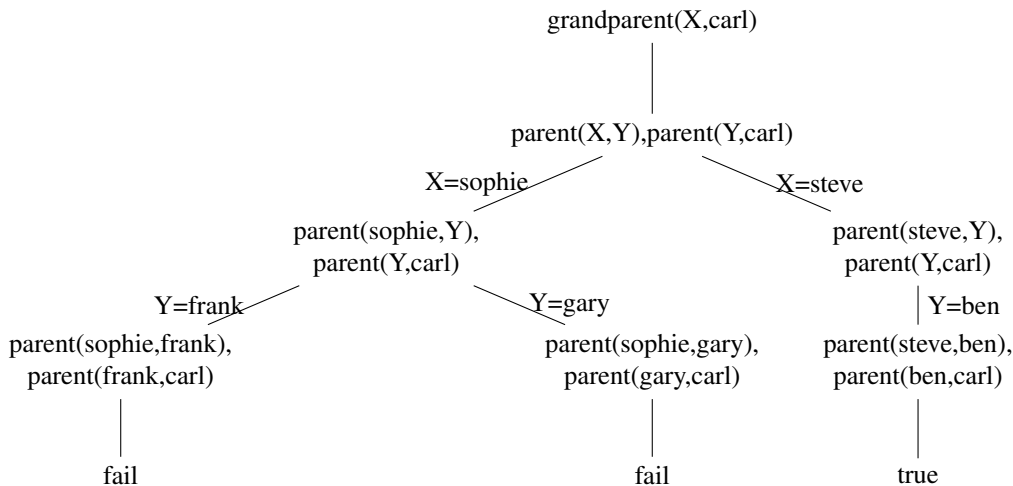


Figure 1: Search Tree

the order, so that parent(claire,sophie) stands on top then the solution would be claire. In Prolog if multiple substitutions can be made, will always choose the one on top of the database. The other solution can in the next step be calculated, but for the same query, come just with a different order different solutions. This should not be possible, because in logic the order does not matter.

2) *Non-Termination:* The Query ?-ancestor(sophie,frank) would on the database 2 give the correct solution and the program will terminate. If you would change some lines to VII-B2 than the same Query would lead to a non ending loop. This happens because ancestor(sophie,frank) is replaced with ancestor(sophie,Z),parent(Z,frank). Then the first guess for Z would be frank. Then the same would go on and on, and the loop will never end.

```
ancestor(X,Y):-ancestor(X,Z),parent(Z,Y).
ancestor(X,Y):-parent(X,Y).
```

Listing 4: Non-Terminating Database

3) *Recursion:* It is also possible to use recursions in Prolog. One of the possible most simple recursions are the natural numbers. Hereby 0 is defined as a natural number and every number, which is the successor of a natural number is also a natural number. The succ function which is true if the second integer is the successor of the first one, is implemented in most Prolog compilers and can therefore be used.

```
nat(0).
nat(X):-succ(Y,X),nat(Y).
```

Listing 5: Natural Numbers

Next a bit more complex program in Prolog. This is the common known Quicksort in Prolog. Hereby is new the possibility to use lists in Prolog. A List can be either in the form [a,b,c], where a,b and c are elements of the list, or it can be in the form [a|b] where a is an element and b is another list. This

program is called with a query ?-quicksort([4,6,8,34,63,2,1],X) to get the sorted list as an output.

```
quicksort([X|Xs],Ys):-
    partition(Xs,X,Left,Right),
    quicksort(Left,Ls),
    quicksort(Right,Rs),
    append(Ls,[X|Rs],Ys).
quicksort([],[]).

partition([X|Xs],Y,[X|Ls],Rs):-
    X < Y, partition(Xs,Y,Ls,Rs).
partition([X|Xs],Y,[X|Ls],Rs):-
    X = Y, partition(Xs,Y,Ls,Rs).
partition([X|Xs],Y,Ls,[X|Rs]):-
    X > Y, partition(Xs,Y,Ls,Rs).
partition([],Y,[],[]).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

Listing 6: Quicksort

C. *Non-Uniformity*

A problem specific for the language Prolog is the non-uniformity of Prolog-Compiler. There exist a lot of different Prolog-Compiler but they are all different. They all use different solving-schemes, have different built-in-functions, so that a given Prolog-Program even if it is very basic and runs on one Prolog Compiler, there is no guaranty that it will be able to run on another Prolog-Compiler the same way as on the first. There does in fact exist an ISO-standard for Prolog Compilers, but many Prolog implementations do not use this. There are intentions to create a common compatibility framework with YAP and SWI-Prolog, but there still exists many other different Compilers. [10]

### VIII. VARIANTS

Even if Prolog is the most common logic programming language there exist a lot of derivatives. There are for example Parlog, which is Prolog for parallel computers and Datalog, which is a truly declarative programming language. It is a subset of Prolog, but has many restrictions, to example a restriction of the recursion and that every Variable which appears in the body has to appear in the head of a clause.

In concurrent logic programming the Horn Clauses are extended to guarded Horn Clauses. The rules then are of the form  $A \leftarrow B_1 \wedge B_2 | B_3 \wedge B_4$  where  $B_1$  and  $B_2$  would be the guards. This is done to reduce the non-determinism of the program, because there will be the rule chosen which guards evaluates to true. For example the quicksort algorithm would be more determined if the body of the partition rules would contain guards, which for example would be  $X > Y$ . With this there would be more determination which resolution should be performed for a given instance. In this case the guards can be interpreted as cases.

Then there is also constraint logic programming where the Horn Clauses are in the form  $A \leftarrow B_1 \wedge B_2 \diamond B_3 \wedge B_4$  where  $B_1$  and  $B_2$  would be the constraints. This constraints can be of any kind which constrains some term, e.g. an integer is only allowed in a specific interval.

### IX. USAGE

The usage of logic programming is in automated theorem proving and mostly various usages in artificial intelligence. Datalog for example also has usage in cloud computing and databases. There are also applications to use logic programs for logic-programming-based model checking. [4]

### X. SUMMARY

The idea to have a programming language build upon logic is an interesting idea. There are many advantages of using a logical language, which mostly come from its declarative upbringings. Therefore its relatively easy to write a working code, which solves problems which derives from the logic with relative ease. But the implementations of the control structures have some negative influences on the behaviour of the program. For example we have seen that for a bit more complex programs it will be difficult to handle problems like non-termination and non-determination. Even for small programs there can be big problems, which consists of different solving schemes, which are used in different Prolog Compilers. Therefore it can be said that logic programming has its definite uses in some specialized fields, but for the general usage for every problem, there are often easier to use methods.

### REFERENCES

- [1] An introduction to prolog. In *An Introduction to Language Processing with Perl and Prolog*, Cognitive Technologies, pages 433–486. Springer Berlin Heidelberg, 2006.
- [2] K. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J. Lloyd, editor, *Computational Logic*, ESPRIT Basic Research Series, pages 150–176. Springer Berlin Heidelberg, 1990.
- [3] W. Clocksin and C. Mellish. The relation of prolog to logic. In *Programming in Prolog*, pages 221–242. Springer Berlin Heidelberg, 1987.
- [4] B. Cui, Y. Dong, X. Du, K. Kumar, C. Ramakrishnan, I. Ramakrishnan, A. Roychoudhury, S. Smolka, and D. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 1998.
- [5] P. Deransart, A. Ed-Dbali, and L. Cervoni. Prolog unification. In *Prolog: The Standard*, pages 11–17. Springer Berlin Heidelberg, 1996.
- [6] T. Ling. The prolog not-predicate and negation as failure rule. *New Generation Computing*, 8(1):5–31, 1990.
- [7] J. Moreno-Navarro and S. Muñoz-Hernández. Soundness and completeness of an “efficient” negation for prolog. In J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence*, volume 3229 of *Lecture Notes in Computer Science*, pages 279–293. Springer Berlin Heidelberg, 2004.
- [8] A. Nerode and R. Shore. Prolog. In *Logic for Applications*, Graduate Texts in Computer Science, pages 159–220. Springer New York, 1997.
- [9] L. Sterling and E. Shapiro. *The Art of Prolog*. 1999.
- [10] J. Wielemaker and V. Costa. On the portability of prolog applications. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin Heidelberg, 2011.

# Parser

Larissa von Witte

Email: larissa.vonwitte@student.uni-luebeck.de

**Abstract**—Parsers are essential for the compiling process by checking the syntax of the input. They also build the foundation of the semantic analysis. This paper will explore the basics behind the modern parsers. At first we will have a closer look at the recursive descent parsers which are commonly used to create a parser by hand. Afterwards we will focus on the shift reduce parser. Since those are rather complex we will explore them on the basis of an example. Later on we will have a short look at parser generators and parse trees, which are essential for the semantic analysis.

## I. INTRODUCTION

Parser are not only used in conjunction with programming languages, but everywhere where grammar and language checks are important (e.g Calculator).

Due to the fact that users can always make syntax errors, it is necessary to check the correctness of the input. In case of an error, the user expects a meaningful error message in order to comprehend and correct it. This leads to more complex parsers.

Furthermore the syntax analysis is relevant in the field of compiler construction. The parse tree, which is produced during the syntax analysis, is fundamental to perform the successive steps.

Since there are different scopes of application for parsers, many different parser types exist. Some of them are so complex that parser generators are necessary to reduce the effort for the developers. Nowadays the development of a compiler is much easier and faster, because in the past parser generation posed a huge part of the compiler development process.

## II. TAXONOMY

Before we can have a closer look at the topic, we need to define some terms in order to understand the basics behind the parsers.

### A. Parser

A parser analyses the syntax of an input text with a given grammar or regular expression and returns a parse tree. A compiler could use the resulting parse tree to perform the semantic analysis. The parser is not to be confused with the scanner, which only checks every word of the input for its lexical correctness and groups the input symbols to tokens. [1]

### B. Lookahead

In the following definitions we're going to need the term "lookahead". The lookahead  $k$  are the following  $k$  tokens of the text, that are provided by the scanner.

Subsequently we will define a notation for the grammars we are using.

### C. Formal Grammars

A formal grammar [2] is a tuple  $G = (T, N, S, P)$ , with:

- $T$  as a finite set of terminal symbols
- $N$  as a finite set of nonterminal symbols and  $N \cap T = \emptyset$
- $S$  as a start symbol and  $S \in N$
- $P$  as a finite set of production rules of the form  $l \rightarrow r$  with  $l, r \in (N \cup T)^*$

### D. Context-free Grammars

A grammar  $G = (N, T, S, P)$  is called context-free [2] if every rule  $l \rightarrow r$  holds the condition:

$l$  is a single nonterminal symbol, so  $l \in N$ .

### $LL(k)$ Grammar

One type of context-free grammars is known as the  $LL(k)$  grammar. To establish the understanding of  $LL(k)$  grammars we will first have a look at the special case  $k = 1$ .  $LL(1)$  grammars are defined by means of the following two expressions [6]:

$$\text{First}(A) = \{t | A \Rightarrow^* t\alpha\} \cup \{\varepsilon | A \Rightarrow^* \varepsilon\}$$

$$\text{Follow}(A) = \{t | S \Rightarrow^* \alpha A t \beta\}$$

with  $S, A \in N, t \in T$  and  $\alpha, \beta \in (N \cup T)$ .

A context-free grammar is called  $LL(1)$  grammar if it holds the following conditions for every rule  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  with  $i \neq j$

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset \quad (1)$$

$$\varepsilon \in \text{First}(\alpha_i) \rightarrow \text{Follow}(A) \cap \text{First}(\alpha_j) = \emptyset \quad (2)$$

Condition 1 indicates that it has to be clear which rule was used to create the text only by looking at the next token (lookahead = 1). The second condition indicates that if the rules contain  $A \rightarrow \varepsilon$  it has to be distinguishable with the lookahead whether it was created by  $A$  or by another nonterminal that can occur after  $A$  in the word. Since  $\varepsilon$ -productions are often used, it would be a too strong restriction to forbid  $\varepsilon$ -productions [3]. The  $k$  in  $LL(k)$  grammar represents the size of the lookahead.

### III. RECURSIVE DESCENT PARSER

A recursive descent parser is the most straightforward form of a parser. It is a top-down parser, which means that the parser attempts to verify the syntax of the input stream by starting with the startsymbol  $S$  and then matching the observed lookahead terminal to one of the expected nonterminals.

The basic idea of a recursive descent parser is to create an own parser  $parse_A$  for every nonterminal  $A$  [6]. All these  $parse_X$  with  $X \in N$  compose the recursive descent parser. The parser begins with  $parse_S$  and decides based on the lookahead which parser needs to be called next. In order to do this  $parse_S$  checks which terminal symbols can be produced by the start symbol  $S$  and compares them to the lookahead. This comparison needs to be distinct, which is the reason why a recursive descent parser can only be used for  $LL(k)$  grammars. To reduce the complexity of the parser an  $LL(1)$  grammar is usually used. It must not be left recursive because otherwise the recursive descent parser might not terminate when it always calls itself without parsing any other symbols [4].

The  $parse_A$  parser is basically a method which consists of case-by-case analysis where it compares the lookahead with the expected symbol of the current case. If none of the cases match the parsing fails and the parser should throw an exception.

The example in Fig. 1 is based on the following  $LL(1)$  grammar.

**expression**  $\rightarrow$  number | "(" expression operator expression ")"

**operator**  $\rightarrow$  "+" | "-" | "\*" | "/"

### IV. SHIFT REDUCE PARSER

A shift reduce parser uses a push-down automaton to analyse the syntax of the input [7].

In the following we will use the notation  $\alpha \bullet au$  to represent the state of the parser [6]. In this notation  $\alpha$  stands for the tokens which are already read and partially processed. These tokens are placed on a stack.  $au$  stands for the tokens which are not yet analysed by the parser. The parser has to choose between two different operations:

- *shift*: read the next token in the input and switch to the state  $\alpha a \bullet u$
- *reduce*: detect the tail  $\alpha_2$  of  $\alpha$  as the right side of the production rule  $A \rightarrow \alpha_2$ , remove  $\alpha_2$  from the top of the stack and put  $A$  on the stack. This way  $\alpha_1 \alpha_2 \bullet au$  is transformed into  $\alpha_1 A \bullet au$  with the production rule  $A \rightarrow \alpha_2$ .

The parser performs a rightmost derivation since the reduce operation always tries to reduce the top of the stack. This and the fact that the parser reads from the left side of the input to the right side leads to the name  $LR(k)$  grammar.  $L$  stands for left-to-right and  $R$  for rightmost derivation. Every

```
boolean parseOperator(){
    char op = Text.getLookahead();
    if (op == '+' || op == '-' || op == '*' || op == '/' ){
        Text.removeChar(); //removes the operator from
                           //the input
        return true;
    }else{
        throwException();
    }
}

boolean parseExpression(){
    if (Text.getLookahead().isDigit()){
        return parseNumber();
    }else if (Text.getLookahead() == '(' ){
        boolean check = true;
        Text.removeChar();
        check &= parseExpression() && parseOperator()
        && parseExpression();
        if (Text.getLookahead() != ')' ){
            throwException();
        }else{
            return check;
        }
    }else{
        throwException();
    }
}
```

Fig. 1. Part of a recursive descent parser for the used grammar [5]

grammar that produces a definite shift reduce parser is a  $LR(k)$  grammar.

Shift reduce parsers are often better for modern programming languages because the grammar of many programming languages is usually already in  $LR(1)$  form. In contrary creating a recursive descent parser often requires to change the grammar into  $LL(1)$  form. This is a very important advantage of the shift reduce parser in comparison to the recursive descent parser.

The shift reduce parser is a *bottom up parser*. A bottom up parser starts with the input text and tries to match the tokens and found nonterminal symbols to nonterminals until it is at the end of the input [8]. If the last remaining nonterminal is the startsymbol  $S$  and there are no terminal symbols left the input is accepted.

To determine which operation needs to be used the shift reduce parser uses a *parser table*. The parser gets the upper state of the stack and the lookahead as input and returns the operation that has to be used.

To create the parser table it is necessary to first create a non-deterministic automaton. Therefore we need to add the production rule  $S' \rightarrow S \text{ eof}$  with  $S'$  as the new start symbol. The states of the non-deterministic automaton consist of items, which are productions of the grammar where the right side has the mark  $\bullet$ . This is necessary to point out at which position on the right side of the rule the parser is. In the following we will use an example to ease the understanding

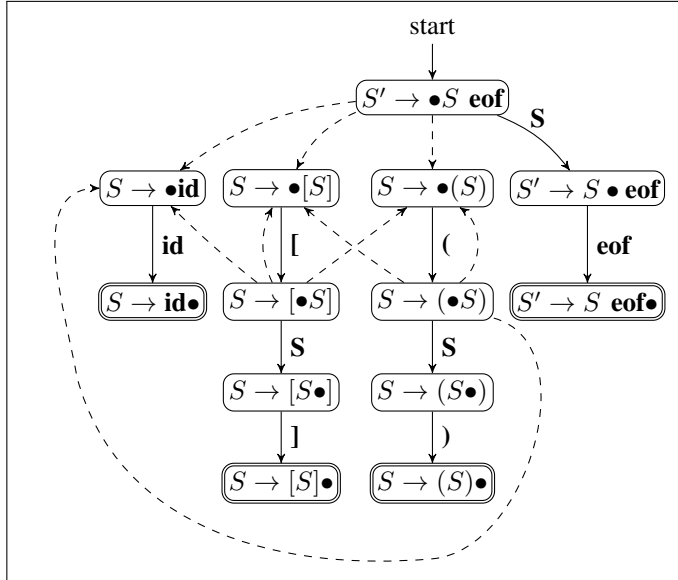


Fig. 2. Non-deterministic automaton for the used grammar

of the transition from a grammar to a parser table.

- |                                |     |
|--------------------------------|-----|
| $S' \rightarrow S \text{ eof}$ | (1) |
| $S \rightarrow (S)$            | (2) |
| $[S]$                          | (3) |
| $\text{id}$                    | (4) |

In this example we have the following items:

- |  |  |  |                             |
|--|--|--|-----------------------------|
| $S' \rightarrow \bullet S \text{ eof}$ | $S' \rightarrow S \bullet \text{ eof}$ | $S' \rightarrow S \text{ eof} \bullet$ |                             |
| $S \rightarrow \bullet (S)$            | $S \rightarrow ( \bullet S)$           | $S \rightarrow (S \bullet )$           | $S \rightarrow (S) \bullet$ |
| $S \rightarrow \bullet [S]$            | $S \rightarrow [ \bullet S]$           | $S \rightarrow [S \bullet ]$           | $S \rightarrow [S] \bullet$ |
| $S \rightarrow \bullet \text{id}$      | $S \rightarrow \text{id} \bullet$      |  |                             |

The non-deterministic automaton has the transitions

$$[A \rightarrow \alpha \bullet \kappa \gamma] \rightarrow [A \rightarrow \alpha \kappa \bullet \gamma]$$

with  $\kappa \in T \cup N$  and

$$[A \rightarrow \alpha \bullet B \gamma] \rightarrow [B \rightarrow \bullet \beta]$$

for every production rule  $B \rightarrow \beta$ .

The second type of transition is called  $\varepsilon$ -transition. The resulting non-deterministic automaton for the example can be seen in Fig. 2.

The dashed lines represent the  $\varepsilon$ -transitions and the states with a bullet as the last symbol are *reduce items*. The reduce items correspond to the detecting of a nonterminal symbol. The next step to create the parser table is to transform the non-deterministic automaton into a deterministic automaton because the choice of the operations to be executed must be definite.

The first state  $A$  of the deterministic automaton is formed by  $A = \delta(S', \varepsilon)$ , with  $\delta$  as the transition function of the non-deterministic automaton. The other states are formed

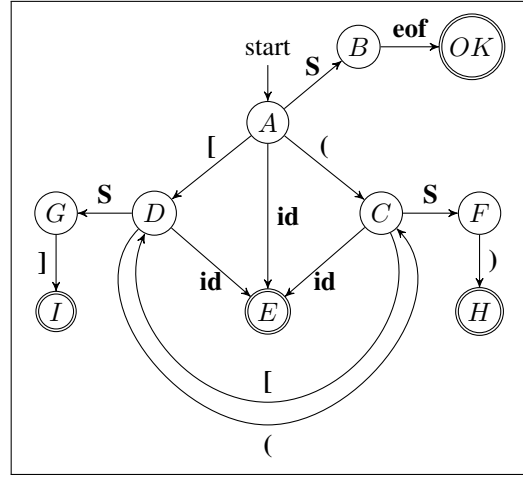


Fig. 3. Deterministic automaton for the used grammar

by  $\delta(X, \alpha)$  with  $X$  as an already existing state of the deterministic automaton and  $\alpha \in N \cup T$ . The resulting table of new states is called a *goto-table* and the deterministic automaton is also called the *deterministic goto-automaton*.

The same state can contain both shift and reduce items, but only if the lookaheads of both operations do not overlap.

The resulting states in the example are:

- |      |   |
|------|---|
| $A$  | $= \{S' \rightarrow \bullet S \text{ eof}, S \rightarrow \bullet (S), S \rightarrow \bullet [S], S \rightarrow \bullet \text{id}\}$ |
| $B$  | $= \{S' \rightarrow S \bullet \text{ eof}\}$  |
| $C$  | $= \{S \rightarrow ( \bullet S)\}$  |
| $D$  | $= \{S \rightarrow [ \bullet S]\}$  |
| $E$  | $= \{S \rightarrow \text{id} \bullet\}$   |
| $F$  | $= \{S \rightarrow (S \bullet ), S \rightarrow \bullet (S), S \rightarrow \bullet [S], S \rightarrow \bullet \text{id}\}$           |
| $G$  | $= \{S \rightarrow [S \bullet ], S \rightarrow \bullet (S), S \rightarrow \bullet [S], S \rightarrow \bullet \text{id}\}$           |
| $H$  | $= \{S \rightarrow (S) \bullet\}$   |
| $I$  | $= \{S \rightarrow [S] \bullet\}$   |
| $OK$ | $= \{S' \rightarrow S \text{ eof} \bullet\}$  |

We form an automaton out of the states from above. The transitions are adopted from the non-deterministic automaton. So for instance  $A$  has every transition which a production rule  $\alpha \in A$  had in the non-deterministic automaton.

For our example this leads to the automaton that can be seen in Fig. 3. Every missing transition in the automaton leads to an error state, which represents a wrong syntax.  $H$ ,  $I$ ,  $E$  and  $OK$  contain reduce items.

Two possible types of conflicts can occur. The first one is the *shift-reduce conflict*. It occurs when one state contains a shift item  $A \rightarrow \alpha \bullet t \gamma$  and a reduce item  $B \rightarrow \beta \bullet$  and it holds that  $t \in \text{Follow}(B)$ . The second conflict is the *reduce-reduce conflict*. It appears when one state contains at least two reduce-items  $A \rightarrow \alpha \bullet$  and  $B \rightarrow \beta \bullet$  and it holds  $t \in \text{Follow}(A) \cap \text{Follow}(B)$ .

After creating the deterministic automaton it is possible to

	(	)	[	]	id	S	eof
A	C		D		E	B	
B							OK
C			D		E	F	
D	C				E	G	
E		r(4)		r(4)			r(4)
F		H					
G				I			
H		r(2)		r(2)			r(2)
I		r(3)		r(3)			r(3)

Fig. 4. Parser table for the used grammar

create the parser table. The rows of the parser table correspond to the states of the automaton and the columns to the terminal and nonterminal symbols. In row  $X$  and column  $\kappa$  is the subsequent state  $\delta(X, \kappa)$  if  $X$  contains a shift item of the form  $A \rightarrow \bullet \kappa \gamma$ . If  $X$  contains a reduce item of the form  $A \rightarrow \alpha \bullet$  and  $\kappa \in \text{Follow}(A)$  we write  $r(x)$  with  $x$  as the number of the production rule that can be used to reduce the previous token or tokens.

This results in the parser table that can be seen in Fig. 4.

The empty fields in the parser table represent error states where the parser detected wrong syntax.

Usually shift reduce parsers are created by parser generators because it is too complex for big grammars to produce them by hand. The example grammar has only 3 production rules in the beginning but the parser table already has 9 rows and 7 columns. So the size of the parser table is relatively big in comparison to the small grammar. This is the reason why it is advised to develop a parser as a recursive descent parser if one wants to create it by hand.

## V. PARSER GENERATORS

Parser generators automatically generate parsers for a grammar or a regular expression. These parsers are often *LR* or *LALR* parsers, which are like *LR* parsers but they merge the states that are similar and only differ in the lookahead [6].

Two famous parser generators are *Yacc* and *Bison*. *Yacc* (“yet another compiler compiler”) and *Bison* are *LALR*-parser generators [9]. They require an input file which consists of three parts that are separated with `%%`. The first part contains the declarations of the tokens, the second part holds the production rules and the third part could contain a C-function that executes the parser.

For the example grammar from the shift reduce parser the file may look like Fig. 5 for both parser generators. *Bison* can produce a file which contains the grammar where it adds the rule  $\$accept \rightarrow S \$end$ , the occurrence of the terminal and nonterminal symbols and the formed parser table. For our example input it produces the file that is shown in Fig. 6. As you can see the produced parser table conforms with the parser table of Fig. 4. The other output file of the parser generator is executable code, which is often in C.

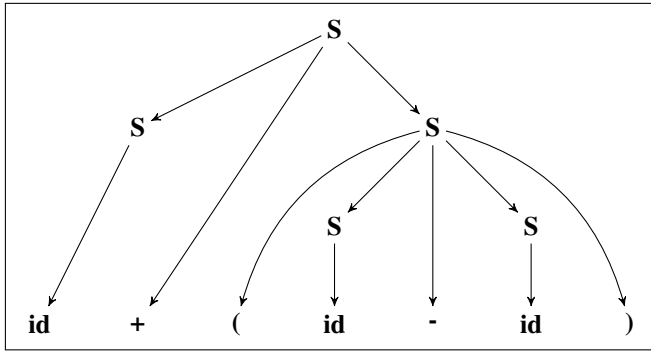
```
%% token ID
%%
S : '(' S ')',
  | '[' S ']',
  | ID
  ;
%%
```

Fig. 5. Example input file for yacc

```
Grammar
  0 $accept: S $end
  1 S: '(' S ')',
  2 | '[' S ']',
  3 | ID
[...]
State 0
  0 $accept: . S $end
  ID shift, and go to state 1
  '(' shift, and go to state 2
  '[' shift, and go to state 3
  S go to state 4
State 1
  3 S: ID .
  $default reduce using rule 3 (S)
State 2
  1 S: '(' . S ')',
  ID shift, and go to state 1
  '(' shift, and go to state 2
  '[' shift, and go to state 3
  S go to state 5
State 3
  2 S: '[' . S ']',
  ID shift, and go to state 1
  '(' shift, and go to state 2
  '[' shift, and go to state 3
  S go to state 6
State 4
  0 $accept: S . $end
  $end shift, and go to state 7
State 5
  1 S: '(' S . ')',
  ')' shift, and go to state 8
State 6
  2 S: '[' S . ']',
  ']' shift, and go to state 9
State 7
  0 $accept: S $end .
  $default accept
State 8
  1 S: '(' S ')' .
  $default reduce using rule 1 (S)
State 9
  2 S: '[' S ']' .
  $default reduce using rule 2 (S)
```

Fig. 6. Example Output file from yacc



Fig. 7. Parse tree of the term  $\text{id} + (\text{id} - \text{id})$ 

## VI. PARSER COMBINATORS

Parser combinators basically combine two parsers into one new parser. There are two common types of parser combinators [10].

The first one is the *sequential combinator*. It takes two parsers and creates a new one which matches the first and the second in order. It fails if one of the transferred parsers fail. This combinator represents the logical AND operation between two parsers.

The second type is the *disjunctive combinator* which corresponds to the OR operation. If the first parser fails the result of the second parser represents the final result, otherwise the final result is a success.

## VII. PARSE TREE

The parse tree is a tree that describes the syntax of the input. It is the output of the parser and can be used during the further compiling process. In the following the structure [3] of the parse tree will be presented.

The root of the parse tree will be the start symbol  $S$ . The interior nodes are the nonterminal symbols of the grammar and the leafs are the terminal symbols. Concatenating the leafs from the left to the right side results in exactly the same sequence of tokens as in the input. The children of a node correspond to the symbols of the right hand side of a production rule.

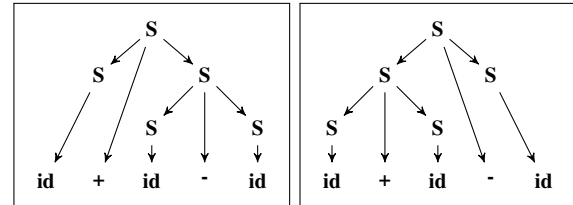
Fig. 7 illustrates a parse tree for the expression  $\text{id} + (\text{id} - \text{id})$  and the grammar:

$$S \rightarrow S + S \mid (S - S) \mid \text{id}$$

The tree in Fig. 7 belongs to an *unambiguous* grammar because there is only one way to illustrate it. If one would change the production rule  $S \rightarrow (S - S)$  to  $S \rightarrow S - S$ , the grammar would become *ambiguous* and so does the tree. This would result in two possible trees for one single expression. Those are represented in Fig. 8.

## VIII. CONCLUSION

Parser theory is complex, since the implementation as well as the transformations of the grammars are extensive. There are multiple different approaches to solve the issues of parser construction. The recursive descent parser is comprehensible

Fig. 8. Parse trees of the term  $\text{id} + \text{id} - \text{id}$ 

and simple to create, which facilitates the work of the developer. The disadvantage of it is its lack of efficiency and the massive restrictions of the grammar, which enforces considerable transformations of the grammar. In contrast, shift reduce parsers are rather complex, but they provide more efficiency and don't require extensive grammar transformations. Their complexity leads to the need of parser generators, which have multiple different generation strategies. A big drawback of parser generators is their rigidity, which conducts to problems considering their usage in combination with experimental features. In addition, the developer has to learn to handle the mostly unintuitive functioning of parser generators in order to use it.

## REFERENCES

- [1] Dr. C. Herrmann, Universität Passau, <http://www.infosun.fim.uni-passau.de/cl/lehre/funcprog05/fohlen/s09.pdf>, 2005 (Abruf 14.12.2015)
- [2] T. Tantau, Universität zu Lübeck, *Vorlesungsskript Theoretische Informatik Wintersemester 2009*, 2010
- [3] Dr. R. Wilhelm and Dr. D. Maurer, *Übersetzerbau Theorie, Konstruktion, Generierung*, p. 317-323, 2nd ed. Springer Verlag, 1997
- [4] Dr. R. Völler, Hochschule für angewandte Wissenschaften Hamburg, <http://users.informatik.haw-hamburg.de/vöeller/fc/comp/node16.html> (Abruf 14.12.2015)
- [5] D. J. Eck, *Introduction to Programming Using Java*, Chapter 9, 7th ed. Hobart and William Smith Colleges, 2014
- [6] H. P. Gumm and M. Sommer, *Einführung in die Informatik*, p. 710-735, 9th ed. Oldenbourg Verlag München, 2011.
- [7] R. Wilhelm, H. Seidl and S. Hack, *Übersetzerbau Syntaktische und semantische Analyse*, p. 114-115 Springer Vieweg, 2012
- [8] N. Wirth, *Grundlagen und Techniken des Compilerbaus*, p. 26-29, 2nd ed. Oldenbourg Verlag München Wien, 2008
- [9] A. V. Aho, R. Sethi and J. D. Ullmann, *Compilerbau Teil 1*, p. 313-318, 2nd ed. Oldenbourg Verlag München Wien, 1999.
- [10] D. Spiewak, <http://www.codecommit.com/blog/scala/the-magic-behind-parser-combinators> (Abruf 20.12.2015), 2009

# Array programming languages

Alexander Harms

**Abstract**—Array programming languages are an important research topic for science or engineering purposes of numerical mathematics and simulations, data analysis and evaluation. This paper provides an overview of different array programming languages and their major use.

Besides it will clarify the main concepts of array programming languages with examples based on MATLAB, APL and QUBE.

## I. INTRODUCTION

The term array programming, also known as vector or multidimensional languages, is used for different programming languages with array oriented programming paradigms and multidimensional arrays as their primary data structures [1]. Those arrays could be vectors, matrices, tensors or scalar values, which are isomorphic to arrays without any axes. The main purpose is the use in science, university or engineering for numerical simulations, collection of data, data analysis as well as evaluation. Although, there are concepts such as classes, packages or inheritance. It is not uncommon, that array programming language one-liners would require more than a couple of pages in other programming languages. The main idea of array programming languages is; that operations can apply to an entire set of data values without resorting to explicit loops of individual scalar operations [2]. Therefore, array programming languages are not very similar to better known programming languages like C++, Java or Python.

Outline: In chapter II it is provided an overview of what array programming languages are for and the idea behind it. In Chapter III it will be explained the array programming language MATLAB with the help of a brief introduction in its concepts and main field of application. In Chapter IV an introduction of APL which is a very old and still used array programming language will be given. Chapter V is about QUBE, a language developed at the institute of software engineering and programming languages at the University of Lübeck.

Chapter VI discusses the conclusion with a short outlook on the future of array programming languages.

## II. WHAT IS THE REASON FOR ARRAY PROGRAMMING?

Using array programming languages, it is easier and faster to solve technical computing problems than with traditional programming languages, such as C, C++, and Java. Their powerful function libraries allow them to avoid loops. The

progress at multicore processors combined to implicit parallelization provides a fast and efficient way of solving numerical problems.

Array programming languages is a powerful and specialized tool, mostly used by research laboratories, universities, enterprises and engineering settings. A big advantage of array programming languages is that they can be used mostly without having a broad knowledge about computer architecture. Many array programming languages handle type declaration and memory management by their own and are very similar to common known math notation. The programmer can think and operate on the data, without thinking about how to handle loops of individual operations. Dr. Iverson described array programming (here referring to APL) as follows [2]:

Most programming languages are decidedly inferior to mathematical notation and are little used as tools of thought in ways that would be considered significant by, say, an applied mathematician. [...]

The thesis [...] is that the advantages of excitability and universality found in programming languages can be effectively combined, in a single coherent language, with the advantages offered by mathematical notation. [...] it is important to distinguish the difficulty of describing and of learning a piece of notation from the difficulty of mastering its implications. For example, learning the rules for computing a matrix product is easy, but a mastery of its implications (such as its associativity, its distributivity over addition, and its ability to represent linear functions and geometric operations) is a different and much more difficult matter.

Indeed, the very suggestiveness of a notation may make it seem harder to learn due to the many properties it suggests for explorations.

[...] Users of computers and programming languages are often concerned primarily with the efficiency of execution of algorithms, and might, therefore, summarily dismiss many of the algorithms presented here. Such dismissal would be short-sighted, since a clear statement of an algorithm can usually be used as a basis from which one may easily derive more efficient algorithm.

### III. MATLAB

MATLAB (MATrix LABoratory) is a multi-paradigm, numerical computing environment, which was developed by MathWorks. Its main features are matrix manipulations, plotting of functions and data, implementation of algorithms and interfacing with programs written in other languages like C++, JAVA or Python). MATLABs initial release, was in 1984, published by the commercial enterprise MathWorks [2]. The MATLAB system consists of five parts. First, there is the language. Then there is the MATLAB working environment, the graphics system, the mathematical function library and the application program interface.

Common usage of MATLAB involves the usage of the command window, characterized by the command prompt '>>'. Here it can be coded all commands like interactive calculations or executing text files, which are containing MATLAB Code in m-files. MATLAB code contains of a sequence of commands, which can be interactively coded in the command window.

The fundamental data type in MATLAB is an n-dimensional array of double precision. Variables are defined by the assignment operator '=' and are case sensitive. Often, it is not aimed to get the printed result of a calculation. Therefore, it is necessary to use a semicolon at the end of the command.

In most of the cases, it is not necessary to think about data types or array and matrices dimensions because MATLAB sets them itself [7].

```
% variable declaration
>> x= 3
x =
    3
% suppressed output
>>string = 'Hello';
```

Listing 1: MATLAB code example of declaring variables

Simple arrays are defined using the syntax *initialization:increment:terminator* (initialization value, increment value, terminator value). As an exception, the increment value can be left out.

```
% standard notation of array declaration
>>array = 1:5:21
array =
    1    5   10   15   20
% array from 0 to pi in 7 steps
>>array = linspace(0 , pi , 7)
array =
    1    0.5236  1.0472  1.5708  2.0944  2.6180  3.1416
```

Listing 2: MATLAB code example of declaring arrays

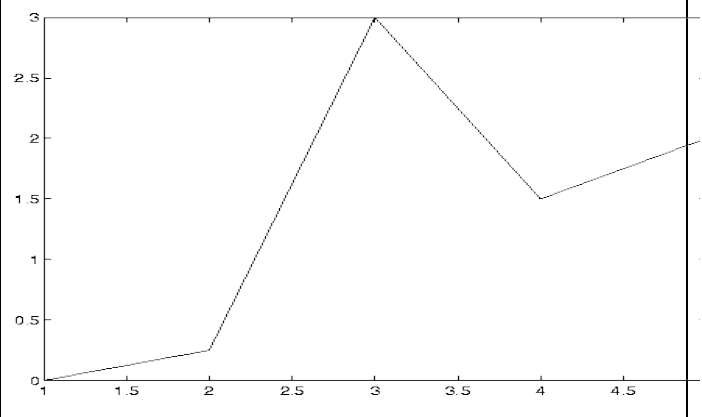
Matrices can be defined by using the syntax [*val11 val21; val12 val22; val13 val23*]. There are also abbreviations to shorten code up.

```
% standard matrices declaration
>>A = [1 1 1; 2 2 2; 3 3 3]
A =
    1    1    1
    2    2    2
    3    3    3
>>A(3,2)
ans =
    3
% declaration of m x n matrices
% additionally instead of 'zeros': 'ones', 'eye', 'rand'
>>B = zeros(3:3);
>>B(2,:)
ans =
    0    0    0
%Matrix multiplication
>>A*B
ans =
    3    3    3
    6    6    6
    9    9    9
>>A.*B
ans =
    1    1    1
    2    2    2
    3    3    3
```

Listing 3: MATLAB code example of working with matrices

As it can be seen in Listing 3, with MATLABs powerful function library, there is no need for loops in MATLAB. Matrices can be multiplied and read out easily with given commands [4]. This prevents off-by-one mistakes, caused by wrong loop notation. MATLAB can handle graphics. It includes commands for two- and three dimensional data visualization. Furthermore, it is possible to create graphical user interfaces on your MATLAB applications.

```
>> x = [1;2;3;4;5];
>> y = [0;0.25;3;1.5;2];
>> figure % opens new figure window
>> plot(x,y)
```



Listing 4: MATLAB code example of creating plots

All in MATLAB created programs are saved as 'm-files',

characterized by the ending ‘*name.m*’. If they are located in the current workspace, they can be called by writing the name in the command window. There are two types of MATLAB-files, namely scripts and functions. Scripts contain a collection of commands like in the command window. Thus, scripts can call and save variables in the workspace. Functions mostly handle parameters, given by the user, run calculations as well as return the result. Besides there are various other file extensions to fit a large range of different requirements [4].

MATLAB commands are easy to handle. The functions are typically named by their use, so reading and understanding MATLAB code is very easy. The programs are typically very short, which allows to get a quick overview about unknown projects. The included documentation allows it to find the needed commands and demonstrate how they are used. All in one, handling arrays and matrices is very easy, since most of the functions are specialized for them. Thus MATLAB is one of the easiest to use and most successful array programming languages.

#### IV. APL

APL was named after the in 1964 published book called ‘A Programming Language’ by Dr Kenneth E. Iverson(1920-2004). In the beginning the function of APL was a notation for expressing mathematical procedures.

Dr. Iverson was a mathematician and had special interest in mathematical notation. He disliked the standard notation and developed a new system in 1957. He began to work for IBM in 1960, where he wrote an extended version and made it possible to be used for the description of systems and algorithms. The result was known as ‘Iverson’s Better Math’. IBM disliked this name, thus Dr. Iverson named it ‘A Programming Language’ (APL). It allows certain data manipulations such as matrix manipulation as well as recursion functions with a special non-ASCII set of symbols. There are different ways to use those symbols with a QWERTY keyboard. The most common is to use a keyboard mapping file.



Source [5]

Each key of a QWERTY keyboard with APL-symbols overlay has up to 4 symbols. As an usual keyboard, alphabetic characters are lower case character and, when pressed with ‘Shift’, upper case character. To type APL-symbols of the lower right of a key, it is necessary to use the ‘AltGr’ instead of ‘Shift’.

‘AltGr’ combined with ‘Shift’ are printing the upper right

symbols.

APL code is evaluated from right to left without hierarchy of function precedence. So  $3 \times 4 + 5$  is 27 using APL, not 17 as one might think. There are just a few rules like parentheses to group subexpressions for changing evaluation order. APL is an interactive language. As datatypes, APL can handle *char*, *bool*, *int* and *float*. As in MATLAB, there is no need to declare variables, the interpreter changes the datatypes whenever it is needed. A *float*-variable can be changed it to a *string* without any special commands. The basic data structure of APL is the vector. Instead of using loops, APL transforms everything into a vector/matrix whenever it is possible. Since 1986 it is possible to build ‘Nested Arrays’. Every Index of a nested array can consist of any datatype. Thus a single array can be built with strings, integers and even new arrays at once. There is no prompt or at APL. The input is indented, the output is not.

```
⊙ Code is interpreted from right to left
3x4+5
27
(3x4)+5
17
⊙ Creating the vector 1 2 3, add the scalar 5,
⊙ add this to the variable A
A ← 5 + 1 2 3
6 7 8
```

Listing 5: APL code example

APL is very powerful and code can be incredible short. To calculate the sum of all integers from one to five, a simple C-solution would be:

```
Include <stdio.h>
Int main()
{
    int I = 0;
    int sum = 0;
    for (I = 1; i<=5; sum += i++);
    printf("Sum: %d/n", sum);
    return 0;
}
```

Listing 6: C-solution example

The APL solution would be:

```
+/ι5
```

In this APL-example, the ‘ι5’ stands for every natural number from one to five. The command ‘/’ means to put the command left between every number of an array right of it [9].

A popular example of an APL “one-liner” is John Conway’s “Game of Life”:

```
life←{↑1 ωV.^3 4=+/,-1 0 1○.⊖-1 0 1○.⊕⊂ω}
```

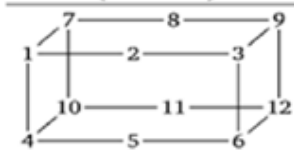
APL is interpreted, thus its performance is low, if it has to handle ‘key pressed events’. On the other hand, APL has a high amount of specialized functions for array programming thus large datasets can be executed very fast. APL works best on multicore processor machines combined with implicit parallelization.

APL is also criticized because the amount of unusual symbols, leads to a cryptic code, which is hard to decode. Often, even the author of a code has trouble to understand his own code later. The challenge to write an APL ‘one-liner’ is hard to resist. Later on, Dr. Iverson created the programming language J, which is similar to APL, but uses exclusive ASCII-symbols. Some call this a great advantage, others criticize, that the self-explanatory of the APL-symbols got lost. Unlike other programming languages, APL was not designed to work like a computer internally, instead it should help the user define procedures for solving problems [11].

Nowadays, APL is mostly used in small projects of specialized software producer, universities and research institutes and has its own wiki.

## V. QUBE

QUBE is an array programming language, developed at the Institute for software engineering and programming languages of the University of Lübeck, the inaugural dissertation of Kai Trojahnner in the year 2011. Usually, rank-generic operations require that the arguments of the ranks, shapes and elements fits in certain constrains. As an example, the element wise addition  $A + B$  in MATLAB dynamically checks that both arrays have the same shape, otherwise the entire program aborts with an error message. The advantage of QUBE is the use of dependent array types to check programs at compile time[6]. QUBEs dependent array types can separate between arrays of different shapes [6]. This allows array operations to precisely specify the allowed arguments and how the type of the result is affected on them. QUBE uses a combination of type checking and automatic theorem proving to statically rule out large classes of program errors like ‘array out-of-bound’ violations. Multidimensional arrays are characterized by two essential characteristics: their rank and shape vector. The rank of a multidimensional array is a natural number that denotes its number of axes. The shape vector is a vector of natural numbers that describe the extent of each axis. Scalars such as five are rank zero arrays with an empty shape vector.

Array	Rank	Shape vector
1	0	[]
[ 1 2 3 ]	1	[3]
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	2	[2 3]
	3	[2 2 3]

Source [6]

In the following paragraph, it is given a short impression of  $\text{QUBE}_{\text{core}}$ . QUBE extends  $\text{QUBE}_{\text{core}}$  with a richer expression of syntax, more base types and others like an I/O.

$\text{QUBE}_{\text{core}}$  comprises three layers,  $\text{QUBE}_{\lambda}$  (applied  $\lambda$ -calculus with dependent types),  $\text{QUBE}_{\rightarrow}$  (integer vectors) and  $\text{QUBE}_{\square}$  (multidimensional arrays) [6].

The language fragment  $\text{QUBE}_{\lambda}$  (QUBE fun) forms the basis of QUBE. As most significant feature,  $\text{QUBE}_{\lambda}$  provides dependent types and refinement types. A refinement type  $\{x:T|e\}$  describes the subset of values  $x$  of type  $T$  that satisfy the boolean expression  $e$ . As an example the type  $\text{nat}$  are all integers with an integer value bigger than 0

$$\text{type nat} = \{x:\text{int} \mid 0 \leq x\}$$

Since the expression  $e$  is true by any  $x$ , the type  $\{x:T|\text{true}\}$  is equivalent to  $T$ . The dependent function type  $x:T1 \rightarrow T2$  binds the variable  $x$  of the domain type  $T1$  in the codomain type  $T2$ . This allows the result type of a function to vary according to the supplied argument [6]. As an example, the code shows an addition of two integer numbers

$$\text{val}+: x:\text{int} \rightarrow y:\text{int} \rightarrow \{v:\text{int} \mid v = x + y\}$$

$\text{QUBE}_{\rightarrow}$  (QUBE vector) adds support for integer vectors. These vectors are important as array shapes and as index vectors into multidimensional arrays.  $\text{QUBE}_{\rightarrow}$  includes syntax for defining, accessing and manipulating integer vectors.

‘ $\text{intvec } e$ ’ is a new type, where the expression  $e$ , a natural number, describes the length of the vector. The vector constructor  $[\bar{e}]$  defines a vector with elements  $\bar{e}$ . If all elements evaluate to integers, the resulting vector is a value. The vector selection  $e.(e_i)$  selects the element at index  $e_i$  from the vector  $e$ . The vector modification  $e.(e_i) \leftarrow e_e$  yields a modified version of the vector  $e$  in which the element at the index  $e_i$  is replaced with the new element  $e_e$ .

```
let a = [1,2,3] in
let b = a.(2) <- a.(0) in      (* b = [1,2,1]*)
a.(2)                        =>* 3
```

Listing 7: QUBE vector code example

Vector  $b$  is defined as a modification of vector  $a$ . The element of  $a$  stays unchanged, while  $b.(2)$  is replaced.

Unlike the vector constructor  $[\bar{e}]$  which creates a vector of fixed length with distinct elements, the constant-value vector expression  $\text{vec } e_n$ ,  $e_e$  defines a vector of non-constant length  $e_n$  that only contains copies of the element  $e_e$ . If the length evaluates to a non-negative integer  $n$  and the element evaluates to a value  $v$ , the entire expression evaluates to a vector that contains  $n$  copies of  $v$ .

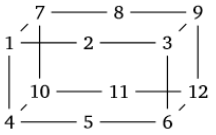
```
let n = 1 + 3 in
vec n 0                        =>*[0,0,0,0]
```

Listing 8: QUBE vector code example

The third language layer  $\text{QUBE}_{\square}$  (QUBE array) adds support for multidimensional arrays and rank-generic

programming.

QUBE array type system classifies arrays with types of the form  $[T|\bar{e}]$ .  $T$  describes the array elements.  $\bar{e}$  is an integer vector, representing the array shape. As an example, a  $2 \times 3$  integer matrix has type  $[int|[2,3]]$ , but also type  $[int|[2],[2+1]]$ , because it evaluates to  $[int|[2,3]]$  [6]. The array constructor  $[\bar{e}:T|[n]]$  defines a multidimensional array with elements  $\bar{e}$  of the element type  $T$  and shape  $\bar{n}$  (and thus rank  $|\bar{n}|$ ). As a data type invariant, the array shape has to be a natural number and the number of array elements must equal to the product of the shape vector. Since these restrictions go beyond mere syntax, they are enforced by the type checker.

Array	Uniform array representation
1	$[1: int   []]$
$[1\ 2\ 3]$	$[1,2,3: int   [3]]$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$[1,2,3,4,5,6: int   [2,3]]$
	$[1,2,3,4,5,6,7,8,9,10,11,12: int   [2,2,3]]$

Source [6]

The following example shows two equivalent selections into a matrix of shape  $[2,2]$ , one uses a single vector, the other a structured vector.

```
[1,2,3,4 :int |[2,2]].[[0,1]]    =>2
[1,2,3,4 :int |[2,2]].[[0],[1]] =>2
```

Listing 8: QUBE array code example

The array modification  $e_a[\bar{e}] \leftarrow e_e$  modifies the array  $e_a$  by replacing the element indexed by the index vector  $\bar{e}$  with the new element  $e_e$ .

```
[1,2,3,4 :int |[2,2]].[[0,1]] ← 0 => [1,0,3,4 :int |[2,2]]
```

Listing 9: QUBE array code example

QUBE<sub>core</sub> also supports type checking. QUBE<sub>core</sub> is type-safe. This means, that no well-typed program can go wrong.

## VI. Conclusion

To sum it up an overview about different array programming languages have been provided.

Array programming languages are powerful for solving numerical problems and working with big matrices and datasets. So MATLAB and APL are providing a large set of functions for array and matrices manipulation. Thus, program code consists mainly of combinations of those functions. The use of loops, which are very error-prone, is not needed anymore. Array programming code is typically very short. This allows the developer to focus on the numerical problem without spending too much time at checking the program code for mistakes or implementing difficult formula.

Especially MATLAB provides a large set of specialized

commands with a high number of external libraries for nearly every situation, whereas APL focuses on its clearness. Also APL code is optimized to write the shortest code possible but in exchange for its comprehensibility. QUBE code is not this short and self-explanatory than the other. It mainly focuses on ruling out as many mistakes of array and matrices manipulation at compile time as possible.

It has to be clarified, that those languages do not fit for every problem and programmer. Often it is needed to invest a lot of time to learn and master those. Also the possibility of coding programs with a fitting and user friendly graphic interface is not always given. For those specific requirements, array programming languages are best choice to use for.

Paired with the great technological progress of multicore processors and implicit parallelization, array programming languages can show their power. In the future, array programming languages can have a bigger influence on daily work as well as research life. Combined with other programming languages, even faster and more efficient programs are possible than nowadays it is. It is a possible research field for imaging processing. Since images are large datasets of matrices, calculations operated on them, could be speed up significant.

All in all, array programming languages are an interesting field of research.

## REFERENCES

- [1] Trojahner K., Clemens Grellck "Dependently Typed Array Programs Don't Go Wrong" [https://staff.fnwi.uva.nl/c.u.grellck/publications/2009\\_JLAP\\_qube.pdf](https://staff.fnwi.uva.nl/c.u.grellck/publications/2009_JLAP_qube.pdf) [online] 2015
- [2] MathWorks [https://www.mathworks.com/help/pdf\\_doc/matlab/getstart.pdf](https://www.mathworks.com/help/pdf_doc/matlab/getstart.pdf) [online] 2015
- [3] Iverson, K. E. (1980). "Notations as a Tool of Thought."
- [4] Houcque D., "Introduction to Matlab for engineering students" <https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-matlab.pdf> [online] 2015
- [5] [http://microapl.com/images/aplx\\_keyboard.jpg](http://microapl.com/images/aplx_keyboard.jpg)
- [6] Trojahner K. (2011) "QUBE – Array Programming with Dependent Types" <http://www.zhb.uni-luebeck.de/epubs/ediss1099.pdf>
- [7] MathWorks, "Language Fundamentals" <http://de.mathworks.com/help/matlab/language-fundamentals.html> [online] 2015
- [8] Iverson K.E., "A programming language" <http://www.jssoftware.com/papers/APL.htm> [online] 2015
- [9] Iverson K. E., "A personal view of APL" <http://web.archive.org/web/20080227012149/http://www.research.ibm.com/journal/sj/304/ibmsj3004O.pdf>
- [10] MathWorks "Matlab The Language of Technical Computing" <https://web.stanford.edu/class/ee262/software/getstart.pdf> [online] 2015
- [11] MicroAPL Ltd "Introduction to APL" [http://microapl.com/apl/introduction\\_chapter1.html](http://microapl.com/apl/introduction_chapter1.html) [online] 2016

# Just-In-Time Compilation

Thiemo Bucciarelli

thiemo.bucciarelli@student.uni-luebeck.de

**Abstract**—In this paper the Just-In-Time compilation technique is examined and compared to interpreters and Ahead-Of-Time compilers. It comprises possible profiling methods and representations, as well as a look at the Java Virtual Machine, its structure and their optimization techniques, based on the HotSpot JVM and JRockit JVM.

## I. INTRODUCTION

Nowadays, software is mostly written in so-called high-level programming languages in order to facilitate the work of the programmer. These languages can't directly be executed by the system, therefore compilers or interpreters are needed. The different types of compilers have evolved through the years and are highly optimized. One sort of compilers are the Just-In-Time Compilers, below often abbreviated as JIT, which we will take a closer look at.

## II. DEFINITIONS

To improve the basic understanding of the topic, the following terms have to be defined.

### A. Compiler

The main task of the compiler is the translation of source code from one programming language to another. Generally, the purpose of the compilation is to translate the code into a code which is either directly executable by the underlying processor, or causes less overhead during the further processing (for example the translation of *Java Sourcecode* to *Java Bytecode*). The resulting code does not necessarily have to be directly executable machine code. A typical compiler includes the following steps [13] to translate the code:

- 1) **Lexical Analysis** The lexical analysis is done by a part of the compiler called scanner. The scanner splits the input into atomic units (called *tokens*) in order to facilitate the following analysis. Every token is a part of the language (for instance the keyword **if**).
- 2) **Syntax Analysis** The syntax analysis involves the parser. The parser uses the list of tokens generated by the scanner and checks if it represents an instance of the given grammar. In general, the result of the syntax analysis is an abstract syntax tree.
- 3) **Semantic Analysis** After the syntactical analysis, the compiler checks the semantics of the program. This includes for instance the type checking, using the syntax tree. This typically results in an intermediate representation of the code.
- 4) **Optimization** The optimization aims to increase the effectivity of the code without affecting the logical

meaning of it. One simple part of the optimization would be the *dead code elimination*.

- 5) **Code Generation** In the last step, the compiler creates the resulting code from the optimized intermediate representation.

### B. Ahead-Of-Time Compiler

One common implementation of compilers is the Ahead-of-Time compiler (abbreviated **AOT**). The AOT compiler does not compile on runtime, but is called by the programmer. It often produces native code which can then be distributed by the developer and be directly executed on the machine. The generation of native code usually makes these compilers strictly platform dependent.

### C. Interpreter

An interpreter does - in opposition to a compiler - not create a translated or executable output file. It performs the analysis at runtime, and then directly executes the given code on the processor. The advantage of interpreters over AOT compilers is their platform independence, since the interpreted program can run on every system which is compatible with the interpreter.

## III. JUST-IN-TIME COMPILATION

A JIT compiler [11] tries to fill the gap between AOT compilers and interpreters, by providing the platform independence of interpreters and combining it with the efficiency of compilers. It performs the compilation on the user's system during runtime. In a first attempt, one would doubt the benefit of this method, since it is implying more work during runtime. The main advantage of JIT compilers is their platform independence. The system generally just has to be compatible with the compiler, which then is able to compile the code for execution on this machine. In most of the cases, the JIT compilation is used in *virtual machines*. These virtual machines can then make advantage of the JIT compilation, providing platform independence, and giving the possibility to perform a dynamic analysis and further optimization of the program during runtime.

There exist two main types of JIT compiler techniques: *trace-based* [12] and *method-based*. A method-based JIT compiler performs the compilation method by method. This proceeding is similar to static compilers, with the difference that the methods are only compiled when needed. So the compiler will not analyse the internal paths inside the methods, but only their calls.

A more sophisticated and time-consuming method is the trace-based JIT compilation. This sort of compiler is usually

used in combination with an interpreter. In a first attempt, the code will be interpreted and analysed during runtime. This analysis allows then to identify which paths (*traces*) of the program are often executed. This makes it possible to determine which parts of the code should be compiled and where to focus during the optimization. The most important property of JIT compilers is their performance. They have to be as time-efficient as possible, since everytime when the compiler works, the program is constrained in its execution. In order to improve the time-efficiency of a JIT compiler, there are multiple possibilities given through the fact that it performs during runtime. A JIT compiler does not have to create localizable code or write the compiled code in a file. Also, the JIT compiler can use runtime analysis to allow better optimizations of the code. Another advantage of a good JIT compiler is that it knows exactly what hardware it is running on. This allows it to create highly optimized machine code which is tailored to the given processor.

#### IV. COMPARING INTERPRETERS, JIT AND AOT COMPILERS

One advantage of AOT over JIT compilers is their performance during execution. JIT-Compilers have some additional work to do during runtime, which makes the execution of JIT compiled code slower than AOT compiled code. Another advantage of AOT compilation is the compile time needed. Regarding AOT compilers, one could say that the compilation speed can be arbitrary slow, since it is only performed once. JIT compilers don't have that privilege, and so they can't spend too much time on compilation, analysis or optimization. On the other hand, the JIT compilation permits to generate highly platform independent code. Since the code is compiled on runtime on the same system where it is also executed, the compiler can generate code which is highly optimized for the given processor, which is an optimization method the AOT compilers can't use. Basically every system which is able to handle the compiler can run the code. There are some optimization techniques for JIT compilers to improve their speed. For instance, a JIT compiler usually analyses the execution of the program in order to optimize it, which increases the effectivity of the program during longer runtimes. Interpreters have the advantage that they don't have to generate any output, and thereby they are faster than the compilers. The advantage of compilers is that they usually have to compile the code only once, which results in a faster execution afterwards. Compilers also have more optimization techniques at their disposition, which improves the efficiency of the generated code. Generally, interpreters are only useful for code parts which are not often executed, because in this case the compilation would eventually need more time than it gains.

#### V. PROFILING

To make the optimization techniques possible, it is necessary to use *profiling*. Profiling means analysing the program and collecting as much relevant information as possible. A

tool which with the ability to perform profiling is then called *profiler*. This part is crucial for the effectivity of the optimizations, since a bad profiling leads to weak optimizations. Basically, there are two types of profilers: *static* and *dynamic*. Static profiling is performed by analysing the program code, while dynamic profiling analyses the program during runtime. Dynamic profiling provides more information and thus permits better optimization possibilities. A very well-known software performing profiling is open-source project *gprof*, which uses a sample-based profiling and can be used for any language supported by *gcc*.

Profiling the program dynamically during its runtime is time-consuming. For that reason there exist multiple strategies to accomplish effective profiling. Especially for JIT compilation, it is important to have a profiler which does not produce too much overhead, since the compilation and optimization are also performed during runtime. Below are explained some methods [6] how a profiler could be implemented.

##### *Sampling-based profiling*

This method uses statistical approaches to reduce the overhead produced by the profiling. Instead of constantly profiling the execution of the program, this profiler activates at specified intervals to check the state of the program. This results in less precise profiles, but in return it is very efficient. Another advantage of statistical profiling is their independence from the program and their sporadic activity. For this reason, they don't affect memory management, caching or performance very much. This causes them to produce less adulterate results than other profilings.

A drawback of the sample-based profiling is that it is possible for a method to coincidentally execute always during the inactive phases of the profiler.

##### *Event-based profiling*

This type of profiling focuses on important events which occurred during runtime. This profiling method stays inactive during runtime until one of the specified events is triggered. If a trigger is activated, the profiler starts its analysis. This is a very specified form of profiling, which can deliver tailored profiles accorded to the users needs.

##### *Instrumentation*

Instrumentation describes the modification of the code by directly inserting code to gather information during runtime. This gives the program the ability to profile itself during runtime. This is more time-efficient than permanent profiling by a parallel running profiling software. The code can directly be injected in the given source code, but it is also possible to add it during a later state of compilation, for instance in the intermediate representation or in the machine code. The instrumentation can also take place during runtime, which then gives a dynamic aspect to the code injections, since they can be adapted to the needs during the execution of the program.

The output of the profilers, the profiles, can have several different formats, according to the needs of the user. In the



%	cumulative	self	self	self	total		
	time	seconds	seconds	calls	ms/call	ms/call	name
	33.34	0.02	0.02	7208	0.00	0.00	open
	16.67	0.03	0.01	244	0.04	0.12	offtime
	16.67	0.04	0.01	8	1.25	1.25	memccpy

Fig. 1. Example flat profile resulting from gprof

index	% time	self	children	called	name
[1]	100.0	0.00	0.05		<spontaneous> start [1]
		0.00	0.05	1/1	main [2]
		0.00	0.00	1/2	on_exit [28]
		0.00	0.00	1/1	exit [59]
-----					
		0.00	0.05	1/1	start [1]
[2]	100.0	0.00	0.05	1	main [2]
		0.00	0.05	1/1	report [3]

Fig. 2. Part of a call graph from gprof

following will be listed some profiles used by gprof [7], differing in the amount of information provided.

### Flat Profile

This profile illustrates how much time every function of the program consumed during runtime, formatted as a list. Every element of the list contains information like the name of the function, the amount of calls, and the needed time. The exact structure of the flat profile of gprof is illustrated in Fig.1.

### Call Graph

This format gives more information on the order of the calls and how much time was used for specific functions. The call graph in gprof has a textual form where every element contains a list, separated by dashed lines. The exact representation can be seen in Fig.2. Every of those elements can be interpreted as a node of a graph, with the calls as the edges and the amount of calls as weights. It gives more detailed information about spent time, caller/callee-dependence and recursive calls. This simplifies the decision which sub-methods should be optimized.

### Execution trace

The execution trace is a profile with a very vast information content for the optimization. It basically contains the complete execution in chronological order, including timestamps. Thereby it builds a representation of the program. If the collected informations include control flow, value, address and dependence histories, it is called *whole execution trace* [2], since that provides a *complete* representation of the execution of the program. So it can be exactly detected which method was called at which time, and which methods were called before or after it. This basic trace can be expanded with additional information like for instance values of variables, memory addresses or the control flow. This type of profile takes a lot of time to generate, and can also consume a significant amount of memory. Especially for JIT compilation, this profile is often not the best to be used.

## VI. OPTIMIZATION

Optimization aims to improve the performance of the application. Compared to static compilers, JIT compilers have a major diversity of optimization techniques at its disposal. But as already stated before, the optimization phase of JIT compilers has to be fast, which excludes some techniques. In the next section we will get a short glimpse at static optimization techniques which can also be used in JIT compilers because of their time efficiency. Afterwards, we will take a closer look at the *adaptive* or *dynamic optimization*.

- 1) Dead code elimination [9] checks for code which will obviously never be executed, for instance a code part in an *else* case where the *if* case always returns true.
- 2) Constant folding and constant propagation [8] Constant folding evaluates constant terms on compile time, e.g.  $x = 3 * 7 + 5$  can directly be substituted by  $x = 26$ . Constant folding takes it further and propagates these constants through the code, substituting every appearance of constants whose value is known by the value itself.
- 3) Loop-invariant code motion [10] This is one of several loop optimization techniques. It checks the loop for code which is not affected by the loop itself, and therefore can be moved outside the loop.

Adaptive optimization describes optimization and recompilation techniques which are performed during runtime, based on the profiling and its analysis. We will take a closer look to the adaptive optimization techniques using the example of the *Java Virtual Machine (JVM)*, which will be analysed in the following chapter.

## VII. THE JAVA VIRTUAL MACHINE

In this part, we are going to examine the structure of the Java compiling process and its optimization methods, using the *HotSpot JVM* and *JRockit JVM*. Both of those JVM implementations are owned by Oracle.

When using Java, the developer compiles its Java code to a bytecode which is stored in a *.class* file. This step aims to improve the performance of the JVM during runtime. The bytecode is an assembly-like code, which is easier to process in the subsequent steps. The compilation to bytecode is static, so the static optimization methods can already be applied here, which leaves the dynamic optimization to the JVM. An example to show how Java bytecode looks like can be seen in Fig.3.

### A. The JVM architecture

This bytecode can then be run by the JVM. The JVM specification does not include the use of Just-In-Time compilation, for this reason the first JVMs were pure interpreters, but that lead to a very bad performance. The HotSpot and JRockit JVMs both make use of Just-In-Time compilation to improve their performance.

Fig.4 represents the structure of the JRockit JVM. In a very

```

1 Compiled from "test.java"
2 public class test {
3     public test();
4     Code:
5         0: aload_0
6         1: invokespecial #1 //
7         Method java/lang/Object.<init>:()V
8         4: return
9
10    public static void main(java.lang.String []);
11    Code:
12        0: iconst_5
13        1: istore_1
14        2: return
15 }

```

Fig. 3. Example of Java-Bytecode, generated with the `javap`-command

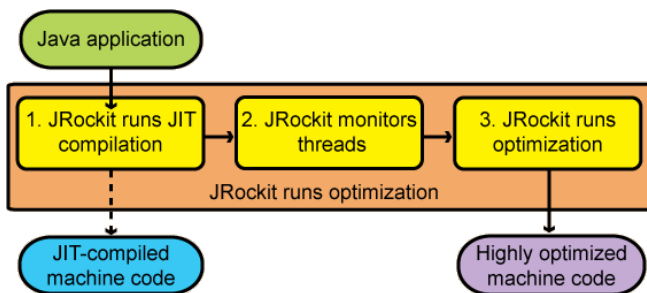


Fig. 4. The illustrated structure of JRockit [1]

first step, the given bytecode is compiled to executable machine code. [1] This step is relatively time consuming, which leads to a rather slow startup of the application. During runtime, the JRockit uses a background thread which periodically activated and checks for often used methods. These methods are favorable to be optimized and marked by the thread. In the last step, several optimization techniques are used to improve the previously generated code. The decision if a code part should be optimized is necessary, since the optimization does also cost time and the time saving of the optimized code does not always equalise the costs.

The HotSpot JVM uses a slightly different approach compared to the JRockit JVM. Instead of always using Just-In-Time compilation, it chooses between JIT and interpreting, based on heuristic approaches. The idea is that nearly unused code parts consume less time being interpreted than being compiled and then directly executed. This results in a faster startup compared to JRockit, but as drawback, HotSpot is usually somewhat slower regarding longer runtimes. Furthermore, HotSpot consumes less memory, since it does not need to compile everything.

### B. Optimizations of the HotSpot JVM

The HotSpot JVM uses several optimization methods to improve the application's performance. [4] They are listed and explained below.

#### Hot Spot Detection

The HotSpot JVM is trace-based, so it does not compile method by method, which allows to compile only parts of a method and interpreting the rest. HotSpot uses the approach of selective optimization, which is based on the assumption that the execution mostly resides in a small part of the code. This is also known as 80/20 rule, which means that 80% of the execution time is covered by only 20% of the code. Since compilers have a lot more overhead than interpreters, interpreting is probably better for the major part of the code. The HotSpot JVM starts the application without compiling anything, and analyses the execution during runtime. With this method, it can detect the so-called *hot spots*, which then are JIT compiled to native code and further optimized. This hot spot detection is running as a background thread during the whole execution, which makes it possible to adapt hot spots to the users behavior and to increase efficiency during runtime. This method usually comes with a great improvement of performance, but since it is based on an assumption, that is not always the case. If the methods of the code are more or less equal distributed, there are no hot spots, which results in a bad performance.

#### Method Inlining

After collecting further information during runtime, HotSpot also checks the frequency of method invocations. Method invocations are time-consuming since they require some overhead, like the usage of the stack to be able to jump back to the invoking code section. If a method is often called by another method, its code can be copied inside that method instead of always using the invocation. This reduces the overhead of calling the methods repeatedly, and it does result in bigger code blocks, which eventually open new possibilities for other optimizing methods.

Method inlining causes problems in combination with recursive calls. Obviously, inlining a recursive call would be an never ending loop. The most common way to solve this is to inline only a certain amount of calls of the recursive methods.

#### Dynamic Deoptimization

A big problem of method inlining is that not every method can be inlined. In Java, non-static methods are virtual by default unless they are final, which means that they are overridable and that their behavior is dependent on their context. In addition, Java uses dynamic loading, which means that classes could be loaded during runtime, and thus also override existing methods. [5] This causes problems for the optimization, since it could be that previously inlined code is going to be overridden during runtime. However, HotSpot is able to check if a method is overridden at a certain moment during runtime. This makes it possible to speculatively inline methods which are not final, but it could be that a later loaded class overrides the method, which then requires the inlining to be undone. This is called *dynamic deoptimization*. Without this speculative approach, inlining for Java applications would only be possible for final methods.

As a logical conclusion, the use of the keyword *final* in Java facilitates the optimization for the JVM.

#### Range check elimination

Other than e.g. C, Java requires a strict check of array indices before accessing the array. Furthermore, if the application reads, modifies and then saves a value of the array, this would cause two range checks. During runtime, the JVM can check if it is possible for an index to leave the bounds of the array - for example between a read and a write operation - and eventually remove some of the range checks.

#### Fast reflection

Reflection describes the ability of an object to analyse and modify the structure of the code (including itself) during runtime. Java provides APIs to use reflection on methods and constructors, using *Object* as default type. This makes it possible to make use of instances and their methods without knowing their exact type. To reduce the overhead of reflections, the APIs use bytecode stubs for the often used reflective objects. Using these stubs, the reflections can be processed by the compiler which improves their performance.

#### I/O Optimization

I/O operations have vast time costs. For this reason, optimizing I/O operations gets a greater focus, by producing highly-optimized machine code which is tailored to the hardware it is running on. This results in a remarkable yield of performance, especially for I/O-intensive applications.

#### The Server Compiler

The HotSpot Server Compiler is a tuned-up version of the client compiler. It focuses more on performance optimizations, which causes the startup to be slower, but usually pays back with a better performance during runtime. It does use more aggressive optimization strategies, by using extensive analysis during interpretation. Some of those resulting optimizations also rely on assumptions based on the given analysis, which could force the JVM to deoptimize it later on.

#### C. When to use JIT?

As already stated before, the HotSpot JVM does not always use JIT compilation. Analogous to the optimization technique, it bases its decision whether to compile or to interpret on the runtime analysis. [3] The idea is that methods which are hardly or never used will produce way too much overhead when being compiled. Instead of compiling, JVM will interpret these methods by running a semantic equivalent directly on the machine, which is less time consuming than compiling the code part. The effectiveness of this method is of course directly dependent of the constraints used for the decision. The interpretation / optimization will be performed method by method, so this decision is not based on traces or hot paths. This heuristic decision mostly delivers a better performance than *always JIT* or *always interpret* does. However in certain special cases - for instance if the executed methods deliver no useful pattern for the analysis - the performance will not

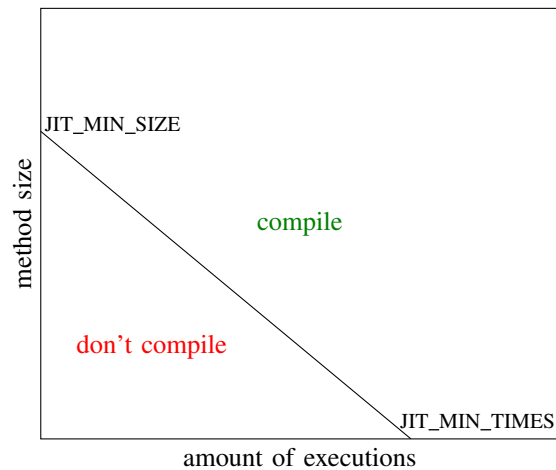


Fig. 5. Plot of the heuristic approach, based on amount of executions and size of the method

improve. For such a decision, one should consider that interpreting a method repeatedly is much more time consuming than compiling a method which will nearly never be used again. So the heuristics should favorize JIT and rather compile too much than too little.

The very basic metrics which could be used are the size of the method and the execution count. The size of the method is relevant because a method consisting of only a few lines will cost a respectively large amount of time for compiling, compared to the resulting time saving. A method which is often executed should also be compiled, since that will lower the time costs on this method. So the conclusion is that the worst method to compile is small and rarely used, while the best method to compile is large and often used. An example for the heuristic decision whether to compile or not can be seen in Fig.5, based on two parameters *JIT\_MIN\_SIZE* and *JIT\_MIN\_TIMES*.

## VIII. CONCLUSION

JIT compilation provides compilers with the ability to be highly independent, and to optimize its code in a very efficient way. But the implementation of JIT compilers is a difficult task. Planned optimization methods could result in a slower execution, by needing too much time to generate the related profiles or optimizing the code. The adaptive optimization methods is based on the execution of the program, which leads to the fact that the optimization techniques vary in their effectivity. Therefore, one has to decide which optimization methods are most probable to be profitable and should be used. Furthermore, one should consider the fact that interpreting is in some cases better than JIT compilation, which again aggravates the construction.

## REFERENCES

- [1] Oracle Corporation, *Oracle JRockit JVM Diagnostics Guide* pp.29-31, April 2009.

- [2] X. Zhang and R. Gupta, *Whole Execution Traces* In Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37). IEEE Computer Society, Washington, DC, USA, 105-116, 2004.
- [3] J. L. Schilling, *The Simplest Heuristics May Be the Best in Java JIT Compilers* SIGPLAN Not. 38, 2 (February 2003), 36-46, 2003.
- [4] Oracle Technology Network, *The Java HotSpot Performance Engine Architecture*, <http://www.oracle.com/technetwork/java/whitepaper-135217.html#3>
- [5] M. Paleczny et al., *The Java HotSpot Server Compiler* In Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01), Vol. 1. USENIX Association, Berkeley, CA, USA, 2001.
- [6] S. Banerjee, *Performance Tuning of Computer Systems* pp.12-13, University of Ottawa, Computer Architecture Research Group.
- [7] *GNU gprof documentation* <https://sourceware.org/binutils/docs/gprof/>
- [8] R. Gupta, *Lecture 6: Code Optimizations: Constant Propagation & Folding* <http://www.cs.ucr.edu/~gupta/teaching/201-09/My6.pdf>, University of California Riverside, 2009.
- [9] R. Gupta, *Lecture 8: Code Optimizations: Partial Dead Code Elimination* <http://www.cs.ucr.edu/~gupta/teaching/201-09/My8.pdf>, University of California Riverside, 2009.
- [10] P. Colea, *Generalizing Loop-Invariant Code Motion In a Real-World Compiler* Imperial College London, Department of Computing, pp.20, June 2009.
- [11] M. Ide, *Study on method-based and trace-based just-in-time compilation for scripting languages* Graduate School of Engineering, Yokohama National University, pp.3-8, September 2015.
- [12] M. Bebenita et al., *Trace Based Compilation in Interpreter-less Execution Environments* In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10). ACM, New York, NY, USA, 59-68, 2010.
- [13] D. Kästner, *Compiler Structure* <http://www.rw.cdl.uni-saarland.de/~kaestner/es0203/lectdk08.pdf>, Universität des Saarlandes, 2002.

# Synchronous vs. Asynchronous Programming

Jan Pascal Maas

**Abstract**—The contrast of synchronous and asynchronous programming has become more important in the last couple of years as especially many web-based applications rely on the latter. This paper shall characterize both approaches with an overview about their concepts and application areas. This will also be supported by the discussion of sample implementations of each concept.

Additionally, this paper will discuss the importance of concurrency for these concepts, especially regarding the synchronization of a system.

## I. INTRODUCTION

When implementing various tasks in software, many constraints can alter the way a programmer works. Two important restrictions involve concurrency and communication between various parts of an implementation. The requirement of concurrency is especially important when the handling of multiple tasks at once is crucial. In particular, this requirement is used by current web-based applications. In general, this ensures that comparatively fast operations do not wait for slow operations [1]. To implement such a system can be a difficult task in comparison with imperative programs [1], [2]. But still, general programming paradigms can be applied as concurrency is a largely explored field [3]–[6]. Additionally, many approaches for concurrent applications exist [3], [6].

If the need of communication is an issue, the complexity of a system grows largely as the possibly concurrent tasks require information exchange between each other [1]. One example for this would be a system for air traffic control. If all information about the environment is known beforehand, the system can run several tasks successively to control the traffic in an easy way. Also, the system can run concurrently as all tasks are independent of each other. If the traffic is not known beforehand, the control happens while executing the system. With this in mind, it is important to have a system which is both, capable of accepting inputs and managing the air traffic, at the same time. In general, real-time, reactive and interactive systems require a high amount of synchronized communication while also being capable of interacting with the environment or the user [7], [8].

In imperative programs the use of shared-memory or message passing may solve most communication problems. In concurrency, those paradigms can also be used. But for a working implementation of communication, the approaches require synchronization. While this is implicit in message-passing models, where a message can only be received if it was sent before, a shared-memory model needs additional mechanisms to ensure synchrony [1].

To ensure synchrony in any of the discussed concurrent environments, two implementation methods can be used: *spinning*, also known as *busy-waiting*, or by *blocking*. While in

busy-wait synchronization a thread runs a loop to reevaluate a specific condition until it becomes true, the blocking (also called *scheduler-based*) allows a processor to use the resources of a blocked thread for different operations. Before blocking, the corresponding thread leaves a note to ensure the resuming of computation if a specific synchronization condition is required [1].

These two synchronization models can be assigned to two programming paradigms. The *synchronous* approach applies the scheduler-based synchronization in combination with simple implementation methods. On the other hand, the *asynchronous* approach mainly relies on the busy-waiting synchronization. This allows systems to interact with the environment while—asynchronously—computing different results of the system [8]. For real-time, reactive and interactive systems, those approaches allow to easily create working implementations. Even though their main goal is to allow synchronization of programs, the approaches of synchronous and asynchronous programming vary in multiple aspects that should be discussed in this report.

The next section will discuss the two paradigms and delimit them from each other. This also build the basis for the detailed discussion of two asynchronous implementation approaches in the languages Node.js and F#. In the following, implementation issues in concurrent programming will be discussed before section V will discuss the implementation of synchrony in the three languages LUSTRE, SIGNAL and ESTEREL which all follow the Synchrony Hypothesis [7]. The report will end up concluding the approaches and an evaluation about the practicability of both programming paradigms in various scenarios.

## II. DELIMITATION OF THE PARADIGMS

In many implementations, the delimitation of synchronous and asynchronous approaches is of no importance as necessary requirements are not fulfilled. These premises contain various architectural aspects like the use of multi-threaded systems. Also, the use of communication and especially its necessity of synchronization is crucial [1]. These three requirements describe if the use of synchronous or asynchronous approaches is necessary. If not, most imperative techniques are enough to solve a problem with an implementation. If at least some of these requirements are part of the issue, any of the approaches can solve it in their respective way. Still, synchronous and asynchronous programming techniques deliver various solutions.

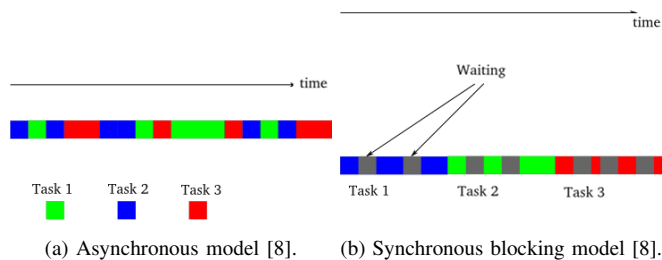


Fig. 1. Comparison of the single-threaded synchronous blocking and asynchronous model [8].

### A. Synchronous Programming

In synchronous programming, the *scheduler-based synchronization* is used [8]. By this, a thread of a program is blocked to ensure the correct computation in the end. Additionally, the thread may continue if the resources which are necessary for correct computation are available [1]. Even though this ensures correctness of a system, this approach delivers some major issues which should be avoided widely.

First of all, the blockage of a thread might block the complete system. As an example, a client which requires data from a given database server may be considered. If one thread acts as a client and issues the request for data synchronously to an other thread that may act as a server, the client-side application waits for the response. This response can take some time as the server has to check its database for corresponding entries. During this time, the client is not capable of sending new requests or change the view as its thread is blocked. This can cause multiple issues starting from violating a deadline over frustrated users to client time outs as the requested data is not available soon enough [8].

Except for long execution times, this paradigm also requires the correct and especially manual use of synchronization mechanisms in a concurrent computing environment. This leads to issues ensuring the correctness of a system. This may be discussed wider in section IV.

### B. Asynchronous Programming

The main difference between synchronous and asynchronous approaches is shown in figure 1. It shows that the synchronous approach used blocking to synchronize different tasks while the asynchronous is interleaving the execution which is also called non-blocking.

In an asynchronous approach, *busy-waiting synchronization* is used to synchronize a program. This means that a specific condition—like the occurrence of an event—is reevaluated over time. If it becomes true, which means that the evaluated condition depends on an external system like a database, some actions which were specified beforehand take place. This allows a system which is implemented asynchronously to keep working without getting blocked. Instead, the main thread of the system is still running while the operation which would block the system is executed by a different thread. This allows

the asynchronous system to react to varying inputs which means that the user is capable of interacting with it [1].

Additionally, an asynchronous program will outperform a synchronous single-threaded one in this case. As shown in figure 1b, a synchronous program would wait for corresponding external information as long as necessary and block the executing task. An asynchronous program on the other hand would work as discussed before and shown in figure 1a [2].

Furthermore, the main process of the asynchronous system stays single threaded. This leads to a process execution scheme like in figure 1a. It shows that processes can be interleaved in a single thread if the current executing process is blocked because it depends on an external system. Hence, a programmer is capable to determine that if a single process is executing, another is not. One more difference is that the programmer is in control of the decision to suspend a task. In the synchronous model on the other hand, this decision is largely dependent on the operating systems (OS) scheduling. Also, an asynchronous task will continue to run until it explicitly transfers control to an other one [2].

It is also important that the asynchronous programming approach allows the programmer to implement a system without thinking about different synchronization operations. This results from the fact that it is necessary to organize each task as a smaller sequence of steps that may be executed intermittently. In this way, the programmer is especially not responsible for ensuring the temporal correctness of a program as this task is handled by the asynchronous model [2]. Still, asynchrony does not forbid the combination with multi-threading and the usage of both in the same system [8].

To realize the asynchronous programming approach, different models can be used. A model which is frequently used strongly depends on the usage of events. This so called *inversion of control* or *Hollywood principle* [9] offers efficiency and scalability combined with the control over switching between application activities. The implementation relies on event notification facilities like subscribers, publishers and event emitters. If an application has interest in certain events, it registers and is able to read the data corresponding to this event type. When an a priori registered event occurs, the application is notified by using the notification facilities. Afterwards, the application has to handle the occurrence in its own environment [2].

Another paradigm to implement asynchronous programming is using concurrency. In such systems, the I/O processing is the most important one. When using concurrency, it is possible to use multiple threads to perform I/O bound tasks. A major issue for this model is that its scalability is limited. This is caused through the context switching overhead of threads which can not be eliminated easily [2], [10].

Even though asynchronous programming simplifies many issues the multi-threaded model is having, it is not completely perfect. Still, its problems are different in comparison with single- or multi-threaded models in general. As many implementations—for some examples see sections III-A and following—rely on the event based paradigm, it is an issue that



not all interprocess-communication can be reduced to an event notification. This means that developers have to fall back to alternatives like writing some other event-capable mechanism [2].

Also, the sheer complexity of asynchrony in certain programming languages is a big issue—especially in terms of events. Mostly, event handling for different events and their corresponding contexts is possible with the use of varying callback functions. If a programming language lacks of anonymous functions and closures, such as C, developers need to specify an individual function for each event and its corresponding context. To ensure every function has access to the data they need is a challenging task and can be very complex. Also, the code can be very unmaintainable or just a little more than impenetrable [2].

### III. ASYNCHRONOUS APPROACHES

As described in the last section, different models to implement asynchrony exist. The two most important ones—event-oriented and by the use of continuations—will be discussed in this section on the basis of two widespread languages. While Node.js is completely built around event-orientation and is extensively using its options to synchronize the system [2], the functional language F# from Microsoft's .NET framework uses continuations instead of multiple OS threads to implement asynchrony and concurrency [9]. The outdated approach of C# that uses concurrent operations to achieve asynchrony should not be considered in deep here as its use of multiple OS threads to perform I/O operations is expensive and does not scale well, especially because of the overhead of context switching [10]. A future version of C# will use the proposed approach of F# for asynchronous behaviour [9].

#### A. Node.js—Event Loop Approach

When talking about asynchronous programming, the use of events is very important. On the basis of JavaScript's event and callback mechanisms, the runtime *Node.js* implements a non-blocking I/O model that is lightweight and efficient [11]. The main concept behind this implementation is—in contrast to traditional web applications which follow a request/response multi-threaded stateless model [12]—a single-threaded event loop model. This allows Node.js to handle multiple concurrent clients in a simple manner. The event loop can be described with the following pseudo code [12]:

```

loop
  if Event Queue receives a JavaScript Function Call then
    ClientRequest = EventQueue.getClientRequest();
    if request requires Blocking I/O or takes more computation time then
      Assign request to Thread T1
    else
      Process and Prepare response
    end if
  end if
end loop

```

The process is always the same. At first, the incoming client request is placed into the event queue. This is important as the event loop checks periodically if pending requests exist. In the next step, one request is taken from the event queue. If the demanded actions for this request do not require the use of blocking I/O operations like accessing a database or the file system, execute the operations and return the computed response to the corresponding client. If blocking I/O or computational intensive operations are required, the demanded action is transferred to a different thread. This thread is taken from an internal pool that is managed by Node.js. If a thread has finished its operation(s), it returns a response to the event loop which will forward it to the corresponding client. Note that the thread pool is limited and might not be capable of serving the complete amount of incoming requests at a specific moment in time. Still, this allows the main event loop to continue scheduling multiple requests until all threads are in use.

This model has some major advantages. First of all, it is easy to handle a huge number of concurrent client's requests particularly easy. Also, this model does not require the creation of multiple OS threads to handle those requests as Node.js is handling these with an internal thread pool. Furthermore, the amount of resources used is less than with multiple OS threads [12].

#### B. F#—Continuations

In F#, asynchrony is achieved mostly over the use of *continuations*. This allows the programmer to decide what a function is supposed to do in multiple situations instead of returning. While the return of a function is only in its control, the use of continuations transfers it to the power of the programmer. For asynchrony in F#, three different cases of continuations should be considered: success, exception and cancellation. Each continuation can be transferred to a function using parameters [9].

Syntactically, the use of asynchrony differs from the standard language in F#. The use of asynchrony is achieved by the use of new expressions given by a new syntactic category *aexpr*:

```
expr := async { aexpr }
```

The full grammar of this syntactic category is shown in listing 1. To use an asynchronous computation, the type `Async<T>` is mandatory. It will produce a value of type T and deliver it to a continuation. When using the expression `let! v = expr in aexpr`, asynchronous computations form a monad and can bind a result from another asynchronous computation. A *monad* is an abstract type that is capable of transferring values with the generic operations `return` and `bind`:

```
Return: 'a -> M<'a>
Bind:   M<'a> -> ('a -> M<'b>) -> M<'b>
```

This also means that asynchronous operations are capable of binding the results of core language expressions using the standard `let v = expr in aexpr`. Note that this also

Listing 1. Full grammar of the async library [9].

```

aexpr :=
| do! expr // execute async
| let! pat = expr in aexpr // execute & bind async
| let pat = expr in aexpr // execute & bind expression
| return! expr // tailcall to async
| return expr // return result of async expression
| aexpr; aexpr // sequential composition
| if expr then aexpr else aexpr // conditional on expression
| match expr with pat -> aexpr // match expression
| while expr do aexpr // asynchronous loop on synchronous
  guard
| for pat in expr do aexpr // asynchronous loop on synchronous
  list
| use val = expr in aexpr // execute & bind & dispose
  expression
| use! val = expr in aexpr // execute & bind & dispose async
| try aexpr with pat -> aexpr // asynchronous exception handling
| try aexpr finally expr // asynchronous compensation
| expr execute // expression for side effects

```

implies that asynchronous calls may have side effects as F# is an impure and strict functional language [9].

Additionally, the library sometimes needs task generators. In the following example, a function is declared but does not run:

```

let sleepThenReturnResult =
  async { printfn "before_sleep"
          do! Async.Sleep 5000
          return 1000
        }

```

This declaration does neither start a task nor has side effects. This requires to run a task generator `Async<_>` to observe side effects at every run. Note that task generators are capable of running synchronously using `Async`. `RunSynchronously sleepThenReturnResult`. This runs the operation `sleepThenReturnResult` in the background and prints “before sleep”. Afterwards, it does a non-blocking sleep for 5 seconds and then delivers the result to the blocking operation. This allows the computation multiple tasks in parallel using a synchronization operation. Additionally, the choice to have asyncs be task generators allows elimination state in contrast to futures or tasks that always need to be started explicitly and can only be run once. Also, an asynchronous computation can be run synchronously until the first point that it yields as a co-routine if it does not produce a result [9].

Especially the last named feature is interesting as it requires to answer where a callback is run. In the .NET framework each running computation implicitly has access to a *synchronization context* that implies that the callback is running “somewhere”. This can be abused to run asynchronous callbacks on the basis of function closures [9].

Most features delivered by the asynchronous library use the capability of the language to handle different contexts. This allows the delivery of operations for non-blocking I/O primitives that schedule the continuation of an asynchronous computation as a callback in response to an event. Also, loops can be achieved by using asynchronous tailcalls in general. Additionally, F# offers direct support of `for` and `while` loops.

The issue of asynchronous resource clean up is also taken care of by using the language feature `use`: The usage of `use !` allows the programmer to directly dispose resources. Note that this does happen whether the corresponding tasks succeed, fail or are cancelled, even though callbacks and asynchronous responses are implied. Finally, cancellation of tasks is also in asynchrony as it happens on the primitive operational level and not on arbitrary machine code. This is achieved by implicit propagation of a *cancellation token* through the execution [9].

In conclusion, the given asynchronous implementation allows the compiler to reduce the syntax of such tasks to “computation expressions”. This means that any `Async<T>` is represented as a function with three continuations for success, exception and cancellation. Also, this implementation avoids the direct use of extra OS threads to ensure high scalability. Furthermore, this implementation is a nicer way of writing event based code.

#### IV. CONCURRENT PROGRAMMING

As discussed earlier, the implementation of a concurrent system is not an easy task. In general, a programmer is handling a single thread which executes different tasks one after an other. When implementing a new task, this property allows the assumption that all earlier tasks have finished without errors with all their output available for use [8].



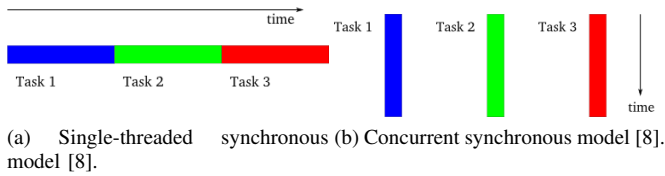


Fig. 2. Comparison of the single- and multi-threaded synchronous model [8].

To implement a single-threaded synchronous algorithm in a concurrent manner, a programmer needs to change his view on a system. Due to the fact that a system depends on hardware and software, the view might change if different software is used on an identical hardware basis. This property leads to the creation of different *models* to define hardware-independent classes of systems [3].

A programmer does only need the most abstract view of a system as this is the view of a programming environment. It defines, how a programmer can address the system. This programming view is not only defined by the hardware of a system, but especially by the operating system, compiler or runtime library. Still, most of the models are based on a *process or thread concept* [3]. This model emphasizes truly concurrent execution of all implemented tasks. Even if the program is executed on a single processor, the view should still be a parallel execution of all tasks as the operating system is taking care of a concurrent execution scheme like interleaving the various tasks [8]. The contrast between the synchronous single-thread and concurrent model is illustrated in figure 2.

Although the model for concurrent programming seems simple, a threaded program can be complex in practice. In particular, a programmer needs to think in the parallel model no matter what hardware should be used in the system. To change the model, programs may work correctly on single-threaded systems while they do work incorrectly on multi-threaded ones [8]. This issue is also supported by the sheer complexity of synchronization which is discussed in section II. While this is no big issue in asynchronous approaches, the use of synchronization requires different rudiments in the corresponding programming languages.

## V. SYNCHRONOUS APPROACHES

### A. LUSTRE

The language *LUSTRE* is data-flow oriented and mainly used for automatic control and signal processing systems. The application area indicates that the language is focussed on temporal correctness—especially according to the input and output behaviour. To structure the code of a program, variables, streams, nodes and assertions can be used. One key feature is the treatment of variables as infinite sequences:

$$(x_0 = e_0, x_1 = e_1, \dots, x_n = e_n, \dots)$$

On variables, any common arithmetic, boolean and conditional operators—called data operators—can be used to combine multiple instances. Note that the definition of a variable is

only possible on the basis of equations which are considered in the mathematical sense [13].

Furthermore, the language consists of four more operators. While the operator `pre(X)` returns the previous sequence of  $X$ , the initialization can happen using an operator “followed by” `->`. To allow more complex programs in which some values of variables do only make sense under some specific condition, two extra operators need to be defined as it is necessary to define variables that are not computed in every cycle or clock step. To do so, the sampling operator `when` can be used. It allows the definition a sequence  $X$  on the basis of expressions  $E$  and boolean values  $B$  as shown in Table I. Still, this implies on the other hand that  $E$  `when`  $B$  does not have the same notion in time as  $E$  and  $B$  as the new sequence is computed on the basis of  $E$  on a kind of “clock” which is represented by  $B$ . This leads to the issue that two variables may describe the same sequence without being equal. This new structure based on the couple formed by a sequence of values and another of boolean expressions is called *stream*. The use of streams enables synchronization of the program, especially when modelling timing constraints [14].

Since the use of `when` alone is not sufficient to enable operations on differently clocked streams, another operator is needed to project instances to the same clock. This can be done by using the `current` operator. This operator takes the last clocked value and for each new cycle it fills the gap between two clock steps. This allows operations over variables of different clocks [14].

As declarative language, *LUSTRE* also supports a function like syntax called *nodes*. When declaring and instantiating multiple nodes, the emerged structure is called a *net*. Finally, another optimization feature are *assertions*. These generalize equations and consist of boolean expressions that should always be true. They indicate to the compiler where code could be optimized. Also, it can help the programmer to synchronize the program as these statements are facts that apply to the whole program [13].

### B. SIGNAL

The synchronization concept of the—also data-oriented—language *SIGNAL* is similar to the concept in *LUSTRE*. The main difference is the direct modelling of variables as *signals* which allows better adaptations to signal processing systems. These basic notions are ordered sequences of its possibly typed values. Additionally, a clock is supplied on each signal which indicates if a value is available. Note that the clock is not absolute but relative and does not describe a real timing relation. Still, it enables to characterize the difference between distinct signals [15].

As the model of *SIGNAL* starts on the direct basis of signal processing systems, it allows the application of common operations for such applications. This includes delaying, undersampling and composing of signals. Additionally, it allows an explicit synchronization using the command `synchro`. In the following example,  $a$  and  $v$  are signals,  $c$  is a boolean signal and the operator `$1` indicates a delay of one step.

TABLE I  
EXAMPLE OF THE OPERATORS WHEN AND CURRENT IN LUSTRE [14].

E	=(	$e_0$	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	...	)
B	=(	$tt$	$ff$	$\#$	$tt$	$ff$	$ff$	...	)
X = E when B	=(	$x_0 = e_0$		$x_1 = e_2$	$x_2 = e_3$			...	)
y = current(X)	=(	$e_0$	$e_0$	$e_2$	$e_3$	$e_3$	$e_3$	...	)

Note that the composition of multiple processes on signals is indicated by (`| ... | ...`):

```
( | a := true when c |
  synchro a, v |
  v := zv + 1 |
  zv := v $1
| ) .
```

Additionally, the language is capable of merging two signals with possibly distinct clocks using `default`. Note that this enables the programmer to synchronize the code and create signals with a higher clock that both of the input signals. This last capability is based on the merging “algorithm”: `a default b` means that the signals `a` and `b` are merged such that the values of `a` are taken if available. If a value from `a` is not available at a certain step of the clock, the corresponding value from `b` is taken if available [15].

### C. ESTEREL

In contrast to SIGNAL and LUSTRE, the language ESTEREL is an imperative language. One important notion in ESTEREL is the *reaction*. It describes the process of computing the output signals based on an input event. Those reactions are completely encapsulated which means that every signal has a fixed status and a current value in each reaction. The *status* of a signal can be present or absent. The *value* depends on this status. If the signal is absent, its value is inherited from the previous reaction and initially  $\perp$ . In a present state, the current value is adopted from the input or output value if the signal is input or either output or local respectively [16].

Its main features for implementing synchrony—except for following the *Synchrony Hypothesis* which means that any output is computed instantly from a given input [7]—are dealing uniformly with input, output or locally declared signals. Specifically, the features are named emit, present and watching. The use of *emit* describes the act of sending output signals to the environment. This implements a communication paradigm based on the sender-receiver pattern [1]. To react on the reception of a signal, the *watching* statement can be used. It implements a watchdog which is waiting for a corresponding signal:

```
do
    I1 -> O1
  watching I2;
  emit O2
```

In this example, the output O2 is emitted on every immediate input I1 until the input I2 is received. Afterwards, the output

O2 is emitted and any further occurrence of I1 is ignored. This means that the watching statement can be used as a timing constraint. Note that if the two inputs occur at the same time, the watchdog is not triggered. The third important statement in ESTEREL is *present*. It detects for the presence of a signal in the current reaction. It works like the watching statement except for the important notion that it does not imply a time limit to the execution [16].

## VI. CONCLUSION

This report discussed the concept of both, synchronous and asynchronous programming. While the first concept is mainly blocking the execution of multiple parts of the program to ensure correctness, the latter is outsourcing the execution of potentially blocking executions to a different thread. This allows the program to stay responsive and enables a better interaction possibility.

These characteristics are also shown in the respective fields of application. While applications based on synchronous programming languages have a huge impact in real-time, automation and signal processing systems, the asynchronous field is larger. It can be used in modern web applications using Node.js, in the common Windows environment using F# or even in reactive systems using asynchronous libraries for corresponding languages. Also, the asynchronous programming approach outperforms the synchronous one in some cases and is at least as fast as the latter. Overall, the asynchronous approach is powerful and in many cases a better choice than the synchronous, even though its justification for existence is given by certain applications.

## REFERENCES

- [1] M. L. Scott, *Programming Language Pragmatics, Third Edition*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.
- [2] S. Tilkov and S. Vinoski, “Node.js: Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.
- [3] T. Rauber and G. Rnger, *Parallele Programmierung*, 3rd ed. Springer-Verlag Berlin Heidelberg, 2012.
- [4] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989.
- [5] S. G. Akl, *Parallel Computation: Models and Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.
- [6] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.
- [7] A. Benveniste and G. Berry, “The synchronous approach to reactive and real-time systems,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep 1991.
- [8] D. Peticolas, “An introduction to asynchronous programming and twisted,” <http://krondo.com/?p=1209>, 2009, accessed: 2015-11-07.

- [9] D. Syme, T. Petricek, and D. Lomov, "The f# asynchronous programming model," in *In Proceedings of Principles and Applications of Declarative Languages, 2011*. ACM SIGPLAN, 2011. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=147194>
- [10] S. Edouard, "Asynchronous vs concurrent – what it means in plain english," <http://blog.stevendouard.com/asynchronoussness-vs-concurrency/>, 2014, accessed: 2015-11-22.
- [11] Node.js Foundation, "Node.js," <https://nodejs.org/>, 2015, accessed: 2015-12-17.
- [12] R. Posa, "Node.js processing model – single threaded model with event loop architecture," <http://www.journaldev.com/7462/node-js-processing-model-single-threaded-model-with-event-loop-architecture>, 2015, accessed: 2015-12-17.
- [13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.
- [14] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 178–188. [Online]. Available: <http://doi.acm.org/10.1145/41625.41641>
- [15] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier, "Signal—a data flow-oriented language for signal processing," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 2, pp. 362–374, Apr 1986.
- [16] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992. [Online]. Available: [http://dx.doi.org/10.1016/0167-6423\(92\)90005-V](http://dx.doi.org/10.1016/0167-6423(92)90005-V)

# Aspects of Object Orientation

Jan Niklas Rösch

Email: jan.roesch@student.uni-luebeck.de

**Abstract**—Object-oriented programming (OOP) consists of a wide variety of aspects. All of them play an important part in defining how an object-oriented system works. This paper gives a brief overview of the most relevant aspects of OOP and their varying implementations and differences. This paper tackles many of the fundamental features and facets of OOP by focusing on Variance, Type systems, Subtyping and Subclassing.

## I. INTRODUCTION

While Simula is widely accepted as the first programming language to use object-oriented ideas, Smalltalk is the most relevant when it comes to defining the logic and the paradigms behind most modern object-oriented languages. Since OOP occurred in the 1970s there have been many different definitions and methods to implement an object-like behaviour.

When it comes to discussing different aspects of object-oriented programming there are many things worth to consider. Most notable about a language that handles different objects is the way it deals with them and how these are related to each other.

The facets described in this paper play a crucial role in defining how this interaction is designed. These aspects like variance, subtyping or the used type system do not stand on their own but contribute to the overall behaviour of the language. It is important to understand each of them in order to make an informed decision when choosing a programming language to work with.

We are going to have a look at variance first. Therefore section II will be split into the different types of variance. Each type will be explained with its weaknesses and strengths. Section III gives an overview of the differences and similarities of class-based and prototype-based programming. On that topic we will discuss implementations of subtyping and subclassing mechanisms used in modern programming languages. Section IV explains the two most commonly used type systems. Structural and nominal typing make up the basis for the general behaviour of objects and their relationship.

At the end we give a conclusion to sum up the importance of these topics in modern object-oriented programming languages.

## II. VARIANCE

Variance is a fundamental aspect of object-oriented programming and is widely discussed. It describes the behaviour of complex structures based on the subtyping of the associated classes.

Given a class called *Animal* and a class called *Cat* which is a subtype of this particular *Animal* class, rules of subtyping imply that a *cat* is also an *animal*. While this is rather obvious,

subtyping does not give any information on whether a more complex structure like a list behaves the same. This is where variance comes into play. It describes for example whether this list of cats can also be treated as a list of animals or not. There are four different types of variance that will be explained in the following sections. All of them are equally useful and although often being mixed up, they "can (and should) be integrated in a type-safe manner in object-oriented languages." [1]

In order to declare a type to use variance we can use specific keywords. In Scala for example generic types do not use variance by default. A *Stack* is simply defined as `class Stack[A]{...}` with *A* being the generic type. In Scala we would add a `+` or `-` in front of the generic type to declare the type of variance we want to use [2]. C# supports the *in/out* annotation.

In order to give a formal definition, we use some terms of the boolean algebra. For that matter we use the term  $A \leq B$  to describe that *A* is a subtype of *B*. The term  $A \geq B$  implies that *A* is a supertype of *B*. Also  $I\langle A \rangle$  stands for any complex structure using type *A*.

With these definitions we can now look into the different types of variance.

### A. Covariance

Covariance is the most common approach in OOP when it comes to subtyping of complex structures. It states that the ordering of types is preserved, meaning that the structure of subtyping will be the same when transferred into other structures like lists or arrays. The following term represents this behaviour.

$$A \leq B \rightarrow I\langle A \rangle \leq I\langle B \rangle$$

This also implies that it is possible to use an object of the derived-class wherever an object of the base-class would be applicable. In our example a *Cat* can be used instead of an *Animal*.

Many programming languages use this approach to define the behaviour of the most common structures. Arrays in Java for example are covariant.

```
String [] str = new String [1];
Object [] obj = str;
```

Fig. 1. Arrays in Java

Since arrays are implemented as covariant and *String* is a derived class from *Object*, this piece of code will work

completely fine. However this leaves room for unwanted behaviour since it is still possible to add any object to the array although it should only contain strings. Consider the following code-snippet:

```
String [] str = new String [1];
Object [] obj = str;

obj [0] = 2;
```

Fig. 2. Broken array covariance

This code will compile without any errors but will cause a runtime-exception as we try to put an integer-value into an array of strings. So this array is safe to read but not safe to write. This behaviour is sometimes also referred to as "broken array covariance". Using a stricter type system could have caught this exception at compile-time but would also have a negative effect on the general subtyping rules of the programming language.

With the introduction of generics, many languages now feature a way of implementing these parametrized structures without needing to rely on covariance.

### B. Contravariance

Contravariance is the opposite of covariance as it reverses the ordering of types represented in the term below.

$$A \leq B \rightarrow I\langle B \rangle \leq I\langle A \rangle$$

Although this seems to be pretty unintuitive for many developers, it does come with some benefits.

As demonstrated in the C#-code below it is possible to have an Action-type, which is used to pass a method without explicitly declaring a custom delegate, accept a method that has a more general type than the one that is expected. [3]

```
public class Animal {}
public class Cat : Animal {}

class Program
{
    static void Feed(Animal animal){}

    static void Test()
    {
        Action<Animal> feedAnimal = Feed;

        Action<Cat> feedCat = Feed;

        feedAnimal = feedCat;
    }
}
```

Fig. 3. Contravariant implementation in C#

Here the first Action expects a Function that takes an Animal as parameter. Therefore we can easily pass the function "Feed" without any problems and without relying on variance at all. However it would not be possible to pass it any other function that takes a more derived type, such as Cat.

Using a contravariant approach solves this issue. With the second Action we can safely use a more general type although it would usually expect a more specialized type. We can also pass it the more general Action since Cat is a subclass of Animal. Now we could have different subclasses of Animal using the same "Feed" Function without the need of altering any of the classes themselves.

### C. Invariance

Invariance is used to avoid any kind of type errors. It expresses that neither of the more complex structures is a subtype of the other, regardless of the type-hierarchy of the underlying types. This behaviour could be expressed as follows:

$$A \leq B \rightarrow I\langle A \rangle \not\leq I\langle B \rangle \wedge I\langle B \rangle \not\leq I\langle A \rangle$$

This can be used whenever neither covariant nor contravariant behaviour can be used safely. To make this clear we will look at the example from the beginning with an array of animals and an array of cats.

We have already explained that it is not safe to use a covariant approach where an array of cats can be stored in an array of animals as we would be able to put a Dog in there. So this could only be used in an read-only array.

Contravariance would suggest to treat an array of animals as an array of cats. However this is not safe either because a reader would expect a cat as outcome but in this array of animals other subtypes could be stored as well.

The following example gives another example on how to use invariance.

```
void MammalReadWrite( IList<Mammal> mammals ) {
    Mammal mammal = mammals [0];
    mammals [0] = new Tiger ();
}
```

Fig. 4. Importance of invariant behaviour

We can not pass this function a list of giraffes because we are going to write a tiger into it. So an IList<T> can not be covariant. We also are not allowed to put a list of animals in there because we are going to read a mammal out of it but our animal does not necessarily has to be of any subtype of mammal. Therefore IList<T> can not be contravariant either. [4]

So in order to make this structure completely type-safe it can be useful to use an invariant constructor.

#### D. Bivariance

Bivariance means that both structures have to be a subtype of each other. Therefore both following terms have to be true.

$$I\langle A \rangle \leq I\langle B \rangle$$

$$I\langle B \rangle \leq I\langle A \rangle$$

This is either impossible or not allowed in most programming languages. Because of this we will not discuss bivariance any further but solely name it for the sake of completeness.

### III. SUBTYPING AND SUBCLASSING

In object-oriented programming everything revolves around objects and how they relate to each other. We will discuss this topic using the two most relevant methods when it comes to subtyping and subclassing in order to understand how objects are treated in general. In the course of this we are going to look into the differences of class-based and prototype-based programming.

It is important to understand that there is a decisive difference between subtyping and subclassing [5]. Subtyping on the one hand, often called interface-inheritance, refers to the act of declaring a type as a subtype of another one. So they share a common interface with all the shared methods and fields. The resulting relationship is called a "is-a"-connection, hence a subtype can be used when a type of the base-type would be required. This statement is known as the Liskov substitution principle and is the elemental aspect of subtyping [6]. Subclassing on the other hand only describes the reuse of code. A subclass uses the code of the base-class but is not necessarily a subtype. Therefore subclassing does not alter the type-hierarchy but is a convenient way for the programmer to reuse existing code.

#### A. Class-based

Class-based languages like Smalltalk and Java use classes as blueprints in order to create objects off of it with the exact same structure. Every class has a constructor, being implicitly invoked or explicitly declared, that allocates memory and initializes all fields in the object when instantiated. The new object will inherit all methods and attributes from the class.

The following code demonstrates how we can access the methods in an object that was created according to a class blueprint. In this case a constructor is explicitly defined in order to initialize the attribute of the new object.

```
class numBox{
    private int number;

    public numBox(int num){
        this.number = num;
    }

    public int getNum(){
        return this.number;
    }
}

numBox num3 = new numBox(3);
print(num3.getNum()); //3
```

Fig. 5. Typical implementation of a class

Classes can not change at runtime and therefore the structure of a class and its instantiated objects can not change either. This makes class-based programming very well structured. Besides being easy for the programmer to understand since he does not need to track different versions of the same type, this comes also with many benefits when it comes to compiler behaviour. It is easier to optimize compiler tasks in comparison to prototype-based languages since all class information are available at compile-time and efficient method checks as well as instance-variable lookups can be implemented much more efficiently.

However class-based programming has one major drawback. In these languages, subclassing can not stand on its own and implies subtyping even if it should not be allowed in the present instance. Consider two classes *rectangle* and *square* which is a subtype of *rectangle*, assuming getter and setter methods exist for both width and height. A square has an equal width and height which are dependent on each other. But if a square is used covariantly where a rectangle is expected, unexpected behaviour may occur since the dimensions of the expected rectangle could be changed independently which would violate the square's invariants.

#### B. Prototype-based

Objects inherit from objects. What could be more object oriented than that? [7]

This famous quote captures the key feature of prototype-based programming. While it renounces the use of classes of any kind, it only relies on objects as the main powerhouse.

The main difference when compared to class-based programming is how objects get instantiated. Since there are no classes present that could act as a template, prototype-based programming relies on two different ways of managing objects.

The first is called cloning. This means that an object which serves as the prototype gets cloned to another object. There can be more specific fields added to the clone in order to make it a more specialized version of the prototype. Consider

the following code in JavaScript-like syntax. There are two constructor-functions present, assuming both do have some implemented properties like values or functions attached. We can set the prototype of the ChildClass to match an object of the ParentClass in order to inherit its properties. Every ChildClass object that would be created would now be considered an instance of both classes.

```
function ParentClass () { ... }
function ChildClass () { ... }

ChildClass.prototype = new ParentClass ();
```

Fig. 6. Using the prototype property to use inheritance

The second way is called *ex-nihilo* instantiation. Most languages support some kind of root object to clone from in the first place. This object comes predefined with some of the most important methods such as *toString()* and can be used to declare the most relevant objects. Using object literals we can define new objects that inherit from the default object [8]. The following code gives an example on how to use object literals to create new objects based on the default prototype.

```
var empty_Object = {};

var flight = {
  airline : "Oceanic",
  number : 815,
  departure : {
    city : "Sydney",
    date : "2004-9-22"
  }
}
```

Fig. 7. Using *ex-nihilo* instantiation to create a new object

Many languages feature links between the prototype and the associated clones, meaning that when the prototypes properties are changed the clones will update as well. However this is not supported or even wanted in some languages. If it is not supported we call it pure prototyping or concatenative prototyping. This can improve performance since no delegation takes place during method-lookups although it is much more expensive when it comes to memory management because each property has to be copied for every clone instead of solely maintaining references. Also it makes it possible to change a prototype without altering the clones and therefore have different versions of the same prototype.

This is the point that makes people argue about prototype-based programming a lot. In contrast to class-based languages, prototype-based programming encourages the change of prototypes at runtime. It is possible for example to add or remove functions to an object or -although not recommended-

to change an object's prototype as a whole. This makes programming much more flexible but also has a huge potential for errors.

#### IV. TYPE SYSTEMS

Type Systems ensure type-safety by defining rules about how they are treated in general and when two types can be seen as equal or compatible. This compatibility aspect is specific to each programming language and varies widely depending on different factors, such as support for subtyping or how the underlying equation theories are implemented.

These typing rules are not exclusive to OOP but are needed in every kind of programming. Still they make up one of the essential parts to object orientation since the typing of objects is essential to it.

Because this is a very huge topic we will focus on two of the most common terms, namely nominal vs structural typing, in order to get a basic understanding of the differences as well as advantages and disadvantages of each of them.

##### A. Structural Typing

In structural typing two elements are compatible when they both have the same structure. By structure we mean the fields and methods with their parameters and return values that are present in the element. Most programming languages that feature structural typing do not take the name of the object into account although the name of its fields and methods are relevant to its compatibility. Also it is independent of the place of declaration.

This automatism leaves room for unwanted behaviour because it would be possible to have two equivalent types in structure but very different in meaning. In a structural-typed environment these would be seen as compatible and the programmer would have to track this error on his own.

The following code-snippet is an example for how two elements are structurally equivalent but have two very different meanings.

```
record DistanceInInches
{
  double dist;
};

record DistanceInCentimeters
{
  double dist;
};
```

Fig. 8. An example of structural equivalence

An unwanted type-compatibility like this made a NASA operation fail by bringing a Mars orbiter too close to the planet causing it to disintegrate. [9]

When it comes to structural subtyping interfaces of the available elements will be automatically created and edited



when needed without the programmer having to maintain these himself. Consider two Elements A and B with A having two methods `foo()` and `bar()` while B has methods `foo()` and `baz()`. They will automatically share an implicit common interface with the function `foo()` which the programmer can use. If an element C is added later with the same function or one of the existing elements has been changed, the interface will update accordingly.

This makes structural typing much more expressive and intrinsic [10] as it allows the programmer to declare types without having to define a complete subtyping hierarchy beforehand.

There are differences on how the subtype relationship is defined, depending on the programming language. Most languages relying on a structural typed system take a very simple approach. A type is a subtype of another if and only if it contains all fields of the base type. The subtype may contain additional fields as well. In the following example there are two different types. The animal type only has one field declared as a boolean to it. The dog type also has the exact same definition and is therefore considered a subtype of animal. It also has an additional field of type `int` which has no effect on the implicit subtyping.

```
type animal = {
  mammal: bool;
}

type dog = {
  mammal: bool;
  age: int;
}
```

Fig. 9. Subtyping in a structural typed environment

### B. Nominal Typing

Sometimes referred to as "nominative typing", it is used in many of the most common programming languages today, such as C++, C# and Java. Often a combination of both systems is possible in these languages. Nominal is a subset of structural typed systems and considered to be much more type-safe. However this safety comes at a cost of reduced flexibility.

In nominally typed systems it is no longer possible to accidentally make an object a subtype of another one only because they share the same structure. Therefore the programmer has to explicitly declare it to be a subtype via inheritance. This can be very beneficial and is likely the most important reason why nominal typing became so popular. In most cases there is no real reason to use nominal typing since it does not really add useful features but there are many where it is not possible to use structural typing and nominal typing is the only fitting alternative. As seen in the previous example having two types accidentally be compatible can cause various problems. These errors would simply not be possible with

nominal typing. Another standard example when it comes to describing nominal typing is that there should be a clear difference between `cowboy.draw()` and `circle.draw()` and we certainly do not want them to have a common interface [10].

In nominally typed systems the programmer has to explicitly declare the subtyping hierarchy on his own. In the following example a `Cat` is a subtype of `Animal` and therefore inherits its fields and methods. Without the `extends Animal` annotation, a cat would be completely distinct from all animals.

```
class Animal{
  void feed(){ }
}

class Cat extends Animal{
  int age = 5;
}
```

Fig. 10. Explicitly declared subtyping hierarchy

## V. CONCLUSION

To conclude this paper it is safe to say that object-oriented programming is a very huge topic that provide many different interesting topics to discuss. This brief overview of some important aspects of OOP is supposed to give a insight into how different implementations or approaches to certain topics can have a critical effect on the strengths and weaknesses of a specific language.

Things like the use of variance, type systems and ways of subtyping all make up the key elements of a language and therefore one should have a closer look at these when choosing a language to work with or when designing a new one. The topics we considered in this paper are not standing alone on their own but come together like pieces of a bigger puzzle and when used correctly they can have a huge impact on the program or software they are used in.

Most modern languages feature these aspects in one way or the other, sometimes supporting a mixture of them in order to create an even more powerful tool.

Since its first appearance, OOP has evolved a lot and many new features were introduced to it. Nowadays OOP is the most used concept in programming and presumably will always be.

## REFERENCES

- [1] G. Castagna, "Covariance and contravariance: Conflict without a cause," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 431–447, 1995.
- [2] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*. Artima Inc, 2008.
- [3] <https://msdn.microsoft.com/en-us/library/dd465122.aspx/>, [Online; accessed 2015].
- [4] <http://stackoverflow.com/questions/4669858/simple-examples-of-co-and-contravariance/>, [Online; accessed 2015].
- [5] W. R. Cook, W. L. Hill, and P. S. Canning, "Inheritance is not subtyping," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, 1990, pp. 125–135.

- [6] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994.
- [7] <http://javascript.crockford.com/prototypal.html/>, [Online; accessed 2015].
- [8] D. Crockford, *JavaScript - the good parts: unearthing the excellence in JavaScript*. O'Reilly, 2008.
- [9] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. LaPiana, P. Rutledge, D. Folta, and R. Sackheim, "Mars climate orbiter mishap investigation board phase i report, 44 pp," NASA, *Washington, DC*, 1999.
- [10] D. Malayeri and J. Aldrich, "Integrating nominal and structural subtyping," in *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, 2008, pp. 260–284.