# Metacasanova: a high-performance meta-compiler for Domain Specific Languages

Francesco Di Giacomo

# Contents

# Chapter 1

# Introduction

> About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.
>
> Edsger Dijkstra

The number of programming languages available on the market has dramatically increased during the last years. The tiobe index [...], a ranking of programming languages based on their popularity, lists 50 programming languages for 2017. This number is only a small glimpse of the real amount, since it does not take into account several languages dedicated to specific applications. This growth has brought a further need for new compilers that are able to translate programs written in those languages into executable code. The goal of this work is to investigate how the development speed of a compiler can be boosted by employing meta-compilers, programs that generalize the task performed by a normal compiler. In particular the goal is creating a meta-compiler that significantly reduces the amount of code needed to define a language and its compilation steps, while maintaining acceptable performance.

This Chapter introduces the issue of expressing the solution of problems in terms of algorithms in Section 1.1. Then we proceed by defining how the semi-formal definition of an algorithm must be translated into code executable by a processor (Section 1.2). In this section we discuss the advantages and disadvantages of using different kinds of programming languages with respect to their affinity with the specific hardware architecture and the scope of the domain they target. In Section 1.3 we explain the reason behind compilers and we explain why building a compiler is a time-consuming task. In Section 1.4 we introduce the idea of meta-compilers as a further step into generalizing the task of compilers. In this section we also explain the requirements, benefits, and the relevance as a scientific topic. Finally in Section 1.5 we formulate the problem statement and the research questions that this work will answer.

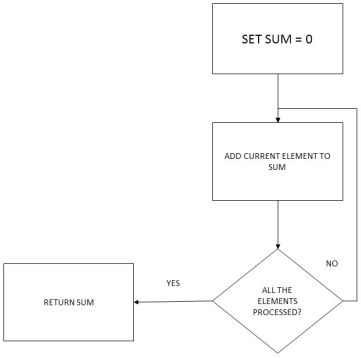


**Figure 1.1:** Flow chart for the sum of a sequence of numbers

## 1.1 Algorithms and problems

Since the ancient age, there has always been the need of describing the sequence of activities needed to perform a specific task [...], to which we refer with the name of *Algorithm*. The most ancient known example of this dates back to the Babylonians, who invented algorithms to perform the factorization and the approximation of the square root [...]. Regardless of the specific details of each algorithm, one needs to use some kind of language to define the sequence of steps to perform. In the past people used natural language to describe such steps but, with the advent of the computer era, the choice of the language has been strictly connected with the possibility of its implementation [...]. Natural languages are not suitable for the implementation, as they are known to be verbose and ambiguous. For this purpose, several kind of formal solutions have been employed, which are described below.

### Flow charts

A flow chart is a diagram where the steps of an algorithm are defined by using boxes of different kinds, connected by arrows to define their ordering in the sequence. The boxes are rectangular-shaped if they define an *activity* (or processing step), while they are diamond-shaped if they define a *decision*. An example of a flow chart describing how to sum the numbers in a sequence is described in Figure 1.1.

### Pseudocode

Pseudocode is a semi-formal language that might contain also statements expressed in natural language and omits system specific code like opening file writers, printing messages on the standard output, or even some data structure declaration and initialization. It is intended mainly for human

reading rather than machine reading. The pseudocode to sum a sequence of numbers is shown in Algorithm 1.1.

**Algorithm 1.1** Pseudocode to perform the sum of a sequence of integer numbers

```
function SumIntegers(l list of integers)
    sum ← 0
    for all x in l do
        sum ← sum + x
    end for
    return sum
end function
```

#### Advantages and disadvantages

Using flow charts or pseudo-code has the advantage of being able to define an algorithm in a way which is very close to the abstractions employed when using natural language: a flow chart combines both the use of natural language and a visual interface to describe an algorithm, pseudo-code allow to employ several abstractions and even define some steps in terms of natural language. The drawback is that, when it comes to the implementation, the definition of the algorithm must be translated by hand into code that the hardware is able to execute. This could be done by implementing the algorithm in a low-level or high-level programming language. This process affects at different levels how the logic of the algorithm is presented, as explained further ahead.

## 1.2 Programming languages

A programming language is a formal language that is used to define instructions that a machine, usually a computer, must perform in order to produce a result through computation [...]. There is a variety of taxonomies used to classify programming languages [...], but all of them are considering four main characteristics [...]: the level of abstraction, or how close to the specific targeted hardware they are, and the domain, which defines the range of applicability of a programming language. In the following sections we give an exhaustive explanation of the aforementioned characteristics.

### 1.2.1 Low-level programming languages

A low-level programming language is a programming language that provides little to no abstraction from the hardware architecture of a processor [...]. This means that it is strongly connected with the instruction set of the targeted machine, the set of instructions a processor is able to execute. These languages are divided into two sub-categories: *first-generation* and *second-generation* languages [...]:

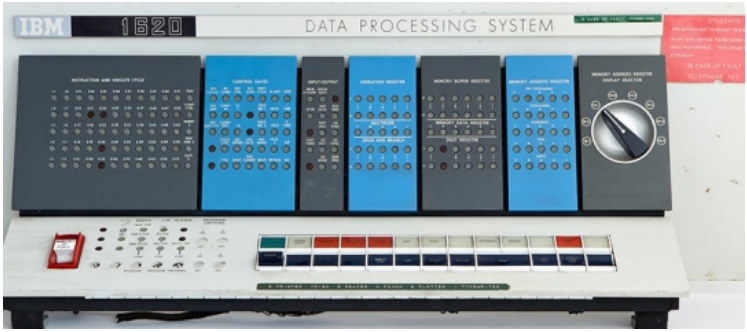


**Figure 1.2:** Front panel of IBM 1620

### First-generation languages

*Machine code* falls into the category of first-generation languages. In this category we find all those languages that do not require code transformations to be executed by the processor. These languages were used mainly during the dawn of computer age and are rarely employed by programmers nowadays [...]. Machine code is made of stream of binary data, that represents the instruction codes an their arguments [...]. Usually this stream of data is treated by programmers in hexadecimal format, which is then remapped into binary code. The programs written in machine code were once loaded into the processor through a front panel, a controller that allowed the display and alteration of the registers and memory (see Figure 1.2). An example of machine code for a program that computes the sum of a sequence of integer numbers can be seen in Listing 1.1.

```
1   00075   c7 45 b8 00 00
2   00 00
3   0007c   eb 09
4   0007e   8b 45 b8
5   00081   83 c0 01
6   00084   89 45 b8
7   00087   83 7d b8 0a
8   0008b   7d 0f
9   0008d   8b 45 b8
10  00090   8b 4d c4
11  00093   03 4c 85 d0
12  00097   89 4d c4
13  0009a   eb e2
```

**Listing 1.1:** Machine code to compute the sum of a sequence of numbers

### Second-generation languages

The languages in this category provides an abstraction layer over the machine code by expressing processor instructions with mnemonic names both for the instruction code and the arguments. For example the arithmetic sum instruction `add` is the mnemonic name for the instruction code `0x00` in `x86` processors. Among these languages we find *Assembly*, that is mapped

with an *Assembler* to machine code. The Assembler can load directly the code or link different *object files* to generate a single executable by using a *linker*. An example of assembly `x86` code corresponding to the machine code in Listing 1.1 can be found in Listing 1.2. You can see that the code in the machine code `00081 83 c0 01` at line 5 has been replaced by its mnemonic representation in Assembly as `add eax, 1`.

```
1  mov DWORD PTR _i$1[ebp], 0
2  jmp SHORT $LN4@main
3  $LN2@main:
4  mov eax, DWORD PTR _i$1[ebp]
5  add eax, 1
6  mov DWORD PTR _i$1[ebp], eax
7  $LN4@main:
8  cmp DWORD PTR _i$1[ebp], 10     ; 0000000aH
9  jge SHORT $LN3@main
10 mov eax, DWORD PTR _i$1[ebp]
11 mov ecx, DWORD PTR _sum$[ebp]
12 add ecx, DWORD PTR _numbers$[ebp+eax*4]
13 mov DWORD PTR _sum$[ebp], ecx
14 jmp SHORT $LN2@main
```

**Listing 1.2:** Assembly x86 code to compute the sum of a sequence of numbers

#### Advantages and disadvantages

Writing a program in low-level programming languages has been known to produce programs that are generally more efficient than their high-level counterparts [...]. However, the high-performance comes at great costs: (*i*) the programmer must be an expert of the underlying architecture and of the specific instruction set of the processor, (*ii*) the program loses portability because the low-level code is tightly bound to the specific hardware architecture it targets, and (*iii*) the logic and readability of the program is hidden among the details of the instruction set itself.

### 1.2.2 High-level programming languages

A high-level programming language is a programming language that offers a high level of abstraction from the specific hardware architecture of the machine [...]. Unlike machine code (and in some way also assembly), high-level languages are not directly executable by the processor and they require some kind of translation process into machine code. The level of abstraction offered by the language defines how high level the language is. Several categories of high-level programming language exist, but the main one are described below.

#### Imperative programming languages

*Imperative programming languages* model the computation as a sequence of statements that alter the state of the program (usually the memory state). A program in such languages consists then of a sequence of *commands*. Notable examples are FORTRAN, C, and PASCAL. An example of the program used in Listing 1.1 and 1.2 written in C can be seen in Listing 1.3. Line 5 to 9 corresponds to the Assembly code in Listing 1.2.

```
1  int main()
2  {
3    int numbers[10] = { 1, 6, 8, -2, 4, 3, 0, 1, 10, -5 };
4    int sum = 0;
5    for (int i = 0; i < 10; i++)
6    {
7      sum += numbers[i];
8    }
9    printf("%d\n", sum);
10   return 0;
11 }
```

**Listing 1.3:** C code to compute the sum of a sequence of numbers

### Declarative programming languages

*Declarative programming languages* are antithetical to those based on imperative programming, as they model computation as an evaluation of expressions and not as a sequence of commands to execute. They are often compared to imperative programming languages by stating that declarative programming defines *what* to compute and not *how* to compute it. This family of languages include *functional programming*, *logic programming*, and *database query languages*. Notable examples are F#, Haskell, Prolog, SQL, and Linq (which is a query language embedded in C#). Listing 1.4 shows the code to perform the sum of a sequence of integer numbers in F#.

```
let sumList l = l |> List.fold (+) 0
```

**Listing 1.4:** F# code to compute the sum of a sequence of numbers

## 1.2.3 General-purpose vs Domain-specific languages

*General-purpose languages* are defined as languages that can be used across different application domains and lack abstractions that specifically target elements of a single domain. Example of these are languages such as C, C++, C#, and Java. Although several applications are still being developed by using general-purpose programming languages, in several contexts it is more convenient to rely on *Domain-specific languages*, because they offer abstractions relative to the problem domain that are unavailable in general-purpose languages. Notable examples about the use of domain-specific languages are listed below.

#### Graphics programming

Rendering a scene in a 3D space is often performed by relying on dedicated hardware. Modern graphics processors rely on shaders to create various effects that are rendered in the 3D scene. Shaders are written in Domain-Specific languages, such as GLSL or HLSL [...], that offer abstractions to compute operations at GPU level that are often used in computer graphics, such as vertices and pixel transformations, matrix multiplications, and interpolation of textures. Listing 1.5 shows the code to implement light reflections in HLSL. At line 4 you can, for example, see the use of matrix multiplication provided as a language abstraction in HLSL.

```
VertexShaderOutput VertexShaderSpecularFunction(VertexShaderInput input,
      float3 Normal : NORMAL)
{
  VertexShaderOutput output;
  float4 worldPosition = mul(input.Position, World);
  float4 viewPosition = mul(worldPosition, View);
  output.Position = mul(viewPosition, Projection);
  float3 normal = normalize(mul(Normal, World));
  output.Normal = normal;
  output.View = normalize(float4(EyePosition,1.0f) - worldPosition);
  return output;
}
```

**Listing 1.5:** HLSL code to compute the light reflection

### Game programming

Computer games are a field where domain-specific languages are widely employed, as they contain complex behaviours that often require special construct to model timing event-based primitives, or executing tasks in parallel. These behaviours cannot be modelled, for performance reasons, by using threads so in several occasions [...] a domain-specific providing these abstractions is implemented. In Listing 1.6 an example of the SQF domain-specific language for the game ArmA2 is shown. This language offers abstractions to wait for a specific amount of time, to wait for a condition, and to spawn scripts that run in parallel to the callee, that you can respectively see at lines 18, 12, and 10.

```
1  "colorCorrections" ppEffectAdjust [1, pi, 0, [0.0, 0.0, 0.0, 0.0],
       [0.05, 0.18, 0.45, 0.5], [0.5, 0.5, 0.5, 0.0]];
2  "colorCorrections" ppEffectCommit 0;
3  "colorCorrections" ppEffectEnable true;
4
5  thanatos switchMove "AmovPpneMstpSrasWrflDnon";
6  [[],(position tower) nearestObject 6540,[["USMC_Soldier",west]],4,true
       ,[]] execVM "patrolBuilding.sqf";
7  playMusic "Intro";
8
9  titleCut ["", "BLACK FADED", 999];
10 [] Spawn
11 {
12   waitUntil{!(isNil "BIS_fnc_init")};
13   [
14     localize "STR_TITLE_LOCATION" ,
15     localize "STR_TITLE_PERSON",
16     str(date select 1) + "." + str(date select 2) + "." + str(date
         select 0)
17   ] spawn BIS_fnc_infoText;
18   sleep 3;
19   "dynamicBlur" ppEffectEnable true;
20   "dynamicBlur" ppEffectAdjust [6];
21   "dynamicBlur" ppEffectCommit 0;
22   "dynamicBlur" ppEffectAdjust [0.0];
23   "dynamicBlur" ppEffectCommit 7;
24   titleCut ["", "BLACK IN", 5];
25 };
```

**Listing 1.6:** ArmA 2 scripting language

### Shell scripting languages

Shell scripting languages, such as the *Unix Shell script*, are used to manipulate files or user input in different ways. They generally offer abstractions to the operating system interface in the form of dedicated commands. Listing 1.7 shows an example of a program written in Unix shell script to convert an image from JPG to PNG format. At line 3 you can see the use of the statement `echo` to display a message in the standard output.

```
1  for jpg; do
2    png="${jpg%.jpg}.png"
3    echo converting "$jpg" ...
4    if convert "$jpg" jpg.to.png ; then
5      mv jpg.to.png "$png"
6    else
7      echo 'jpg2png: error: failed output saved in "jpg.to.png".' >&2
8      exit 1
9    fi
10 done
11 echo all conversions successful
12 exit 0
```

**Listing 1.7:** Unix shell code

### Advantages and disadvantages

High-level programming languages offer a variety of abstractions over the specific hardware the program targets. The obvious advantage of this is that the programmer must not be an expert of the underlying hardware architecture or instruction set. A further advantage is that the available abstractions are closer to the semi-formal description of the underlying algorithm as pseudo-code. This produces two desirable effects: (*i*) the readability of the program is increased as the available abstractions are closer to the natural language than the equivalent machine code, and (*ii*) that being able to mimic the semi-formal version of an algorithm, which is generally how the algorithm is presented and on which its correctness is proven, grants a higher degree of correctness in the specific implementation [...].

The use of a high-level programming language might, in general, not achieve the same high-performance as writing the same program with a low-level programming language, but modern code-generation optimization techniques that can achieve similar performance are known [...]. A further major issue in using high-level programming language is that the machine cannot directly execute the code, thus the use of a compiler that translates the high-level program into machine code is necessary.

The portability of a high-level programming language depends on the architecture of the underlying compiler, thus some languages are portable and the same code can be run on different machines (for example Java), while others might require to be compiled to target a specific architecture (for example C++).

## 1.3 Compilers

A compiler is a program that transforms source code defined in a programming language into another computer language, which usually is object code

but can also be code written into a high-level programming language [...]. Writing a compiler is a necessary step to implementing a high-level programming language. Indeed, a high-level programming languages, unlike low-level ones, are not executable directly by the processor and need to be translated into machine code, as stated in Section 1.2.1 and 1.2.2.

The first complete compiler was developed by IBM for the FORTRAN language and required 18 person-years for its development [...]. This clearly shows that writing a compiler is a hard and time-consuming task.

A compiler is a complex piece of software made of several components that implement a step in the translation process. The translation process performed by a compiler involves the following steps:

1. *syntactical analysis:* In this phase the compiler checks that the program is written according to the grammar rules of the language. In this phase the compiler must be able to recognize the *syntagms* of the language (the "words") and also check if the program is conform to the syntax rules of the language through a grammar specification.

2. *type checking:* In this phase the compiler checks that a *syntactically correct program* performs operations conform to a defined *type systems*. A type system is a set of rules that assign properties called types to the constructs of a computer program [...]. The use of a type system drastically reduces the chance of having bugs in a computer program [...]. This phase can be performed at compile time (*static typing*) or the generated code could contain the code to perform the type checking at runtime (*dynamic typing*).

3. *code generation:* In this phase the compiler takes the *syntactically and type correct program* and performs the translation step. At this point an equivalent program in a target language will be generated. The target language can be object code, another high-level programming language, or even a bytecode that can be interpreted by a virtual machine.

All the previous steps are always the same disregarding of the language the compiler translates from and they are not part of the creative aspect of the language design[...]. Approaches to automating the construction of the syntactical analyser are well known in literature [...], to the point that several lexer/parser generators are available for programmers, for example all those belonging to the `yacc` family such as `yacc` for C/C++, `fsyacc` for F#, `cup` for Java, and `Happy` for Haskell. On the other hand, developers lack a set of tools to automate the implementation of the last two steps, namely the type checking and the code generation.

For this reason, when implementing a compiler, the formal type system definition and the operational semantics, which is tightly connected to the code generation and defines how the constructs of the language behave, must be translated into the abstractions provided by the host language in which the compiler will be implemented. Other than being a time-consuming activity itself, this causes that (*i*) the logic of the type system and operational semantics is lost inside the abstraction of the host-language, and (*ii*) it is difficult to extend the language with new features.

## 1.4 Meta-compilers

In Section 1.3 we described how the steps involved in designing and implementing a compiler do not require creativity and are always the same, disregarding of the language the compiler is built for. The first step, namely the syntactical analysis, can be automated by using one of the several lexer/parser generators available, but the implementation of a type checker and a code generator still relies on a manual implementation. This is where meta-compilers come into play: a meta-compiler is a program that takes the source code of another program written in a specific language and the language definition itself, and generates executable code. The language definition is written in a programming language, referred to as *meta-language*, which should provide the abstractions necessary to define the syntax, type system, and operational semantics of the language, in order to implement all the steps above.

### 1.4.1 Requirements

As stated in Section 1.4, a meta-compiler should provide a meta-language that is able to define the syntax, type system, and operational semantics of a programming language. In Section 1.3 we discussed how methods to automate the implementation of syntactical analyser are already known in scientific literature. For this reason, in this work, we will focus exclusively on automating the implementation of the type system and of the operational semantics. Given this focus, we formulate the following requirements:

- The meta-language should provide abstractions to define the constructs of the language. This includes the possibility of defining control structures, operators with any form of prefix or infix notation, and the priority of the constructs that is used when evaluating their behaviour. Furthermore, it must be possible to define the equivalence of language constructs. For instance, an integer constant might be considered both a value and a basic arithmetic expression.

- The meta-language must be able to mimic as close as possible the formal definition of a programming language. This will bring the following benefits: (*i*) Implementing the language in the meta-compiler will just involve re-writing almost one-to-one the type system or the semantics of the language with little or no change, (*ii*) the correctness and soundness [...] of the language formal definition will be directly reflected in the implementation of the language, and (*iii*) any extension of the language definition can be just added as an additional rule in the type system or the semantics.

- The meta-compiler must be able to embed libraries from external languages, so that they can be used to implement specific behaviours such as networking transmission or specific data structure usage.

### 1.4.2 Benefits

Programming languages usually are released with a minimal (but sufficient to be Turing-complete) set of features, and later extended in functionality

in successive versions. Several times this process is slow and significant improvements or additions are only seen after some years from the last release. For example, Java was released in 1996 and lacked an important feature such as Generics until 2004, when J2SE 5.0 was released. Furthermore, Java and C++ lacked a construct, which is becoming more and more important with the years [...], such as lambda abstractions until 2016, while a similar language like C# 3.0 was released with such capability in 2008. The slow rate of changing of programming languages is due to the fact that every abstraction added to the language must be reflected in all the modules of its compiler: the grammar must be extended to support new syntactical rules, the type checking of the new constructs must be added, and the appropriate code generation must be implemented. Given the complexity of a software such as a compiler, this process requires a huge amount of work, and it is often obstructed by the low flexiblity of the compiler as piece of software, and the need for backward compatibility [...]. Using a meta-compiler would speed up the extension of an existing language because it would require only to change on paper the type system and the operational semantics, and then add the new definitions to their counterpart written in the meta-language. This process is easier because the meta-language should mimic as close as possible their behaviour. Moreover, backward compatibility is automatically granted because an older program will simply use the extended language version to be compiled by the meta-compiler.

To this we add the fact that, in general, for the same reasons, the development of a new programming language is generally faster when using a meta-compiler. This could be beneficial into the development of Domain-specific languages of various kind. Indeed, this kind of languages are often employed in situations where the developers have little or no resources to develop a fully-fledged hard-coded compiler by hand. For instance, it is desirable for game developers to focus on aspect that are strictly tight to the game itself, for example the development of an efficient graphics engine or to improve the game logic. At the same time they would need a domain-specific language to express some behaviours typical of games, thing that could be achieved by using a meta-compiler rather than on a hand-made implementation.

### 1.4.3 Scientific relevance

Meta-compilers have been researched since the 1980's [...] and some solutions have been proposed [...]. In general meta-compilers perform poorly with respect to hard-coded compilers because they add the additional layer of abstraction of the meta-language. Moreover, a specific implementation of a compiler opens to the possibility of implementing language-specific optimizations during the code generation phase. In general we find in scientific literature a substantial effort in developing techniques to optimize the code generation for compilers [...] but not many attempts in producing optimized meta-compilers (one notable exception being [put reference to RML here]). We argue that it could be interesting to present a novel approach into optimize the code generation of a meta-compiler, which might open new horizons to the research on code generation optimization also for normal compilers. Furthermore, the growth in need for domain-specific languages [...] requires the capability of producing compilers in a short amount of time, to which a

significant contribution could be given by presenting a solution based on a meta-compiler. Finally, producing a domain-specific for a field like game development, where high performance is paramount, through a meta-compiler could prove that they can be used to produce languages with decent performance.

## 1.5 Problem statement

In Section 1.2 we showed the advantages of using high-level programming languages when implementing an algorithm. Among such languages, it is sometimes desirable to employ domain-specific languages that offer abstractions relative to a specific application domain (Section 1.2.3). In Section 1.3 we described the need of a compiler for such languages, and that developing one is a time-consuming activity despite the process being, in great part, non-creative. In Section 1.4 we introduced the role of meta-compilers to speed up the process of developing a compiler and we listed the requirements and the benefits that one should have. In Section 1.4.3 we explained why we believe that meta-compilers are a relevant scientific topic if coupled with the problem of of developing domain-specific languages in response to the their increasing need. We can now formulate our problem statement:

**Problem statement:** *Is it possible to build a domain-specific language by using a meta-compiler and to reduce the complexity and length of the code needed to generate runnable code for a program written in that language while having acceptable performance?*

The first parameter we need to evaluate in order to answer this question is the size of the code reduction needed to implement the domain-specific language. At this purpose, the following research question arises:

**Research question 1:** *To what extend can a meta-compiler reduce the amount of code required to create a compiler for a given programming language?*

The second parameter we need to evaluate is the eventual performance loss caused by introducing the abstraction layer provided by the meta-compiler. This leads to the following research question:

**Research question 2:** *How much is the performance loss introduced by the meta-compiler with respect to an implementation written in a language compiled with a traditional compiler? Is this loss acceptable?*

The third parameter we need to evaluate is the gain in terms of code size reduction by using a meta-compiler with respect to a hand-made implementation of a compiler for the same language, which is evaluated through the following research question:

**Research question 3:** *What is the advantage of using a meta-compiler in term of code reduction with rispect to a hand-made implementation?*

## 1.6 Thesis structure

# Chapter 2

# Background

## 2.1 Compilers

- General architecture overview.
- Lexing/Parsing
- Type checking
- Code generation

Describe extensively the structure of all the modules of a compiler.

## 2.2 Meta-compilers

- General overview
- META-languages overview.
- RML overview.
- Possibly other meta-compilers (?)

Describe what it is existing in literature.

# Chapter 3

# Meta-casanova

## 3.1 Meta-casanova language

- Informal description of the grammar and rule evaluation.
- Example: Small example (like Peano numbers)

Take the description from the INTETAIN paper and extend it with more examples and informal details.

## 3.2 Meta-casanova compiler architecture

- Formal syntax.
- Formal rule evaluation.
- Code generation.

Take the syntax from ACM concept paper and the formal evaluation of rules. Explain how the Yacc parser is built and the parser utility support and operator precedence handling. Explain how the code generation part is handled.

## 3.3 Example language in the meta-compiler

- Overview of C–
- Memory representation.
- Scoping.

Explain the case study of C–. Explain how the memory representation is built and how the scoping is handled. Discuss in detail the implementation in Meta-Casanova.

## 3.4 Casanova 2

- Language structure and definition.
- Example of program in Casanova 2.

- Advantages of Casanova 2 as a DSL.
- Difficulty in code generation and type checking due to the state machine generation and interface with .NET.

Explain how Casanova 2 language is defined. Give a small example of a program in Casanova 2 (maybe the patrol sample).

## 3.5 Casanova 2.5 in Meta-Casanova

- World and Entity representation
- Casanova rule evaluation
- Interruption as Continuation Passing Style
- Evaluation

Entities as records implemented with Maps and Casanova rules. Explanation of the tick function (extend the paper). Interruption of control structures with Continuation Passing Style in Metacasanova with example. Using custom .NET libraries in Metacasanova. Performance decays due to the extensive use of Maps for memory access and wrappers around primitive data structures of .NET.

# Chapter 4

# Meta-casanova optimization

The performance decay in Meta-Casanova is caused by the use at runtime of a dictionary to represent records and accesses to record fields. This problem can be eliminated because (*i*) we know that the record cannot grow at execution time, i.e. the fields are always the same during the entire execution of the program, and (*ii*) the field structure does not change at runtime, i.e. their names and types are always the same during the execution. For these reasons we can exploit type functions and module generations to inline at compile time the field accesses.

We can represent a record recursively as a field followed by the rest of the record definition. The recursion will terminate by defining an empty record field. First of all we need to define a way to represent the type of the record. This can be done by using a module that contains a type function returning the type of the record

```
Module "Record" : Record {
  TypeFunc "RecordType"  : *
}
```

This code defines a module called `Record` that contains a type function `RecordType` that returns a kind (because the record type can vary, and so we cannot constraint the return type to be only one type). Now we can define the structure of an empty record, i.e. a record without fields. This is done by defining a type function which returns a `Record`. Calling this type function will generate a module for the empty record.

```
TypeFunc EmptyRecord => Record

-----------------------
EmptyRecord => Record {

  -------------------
  RecordType => unit

  -----------
  cons -> ()
}
```

The function `cons` is used to construct (i.e.) to place data inside the record field. In this case we store unit because the record field is empty.

Now we can define a field for a record. A record field is a type function that generates a record module by taking the name of the field, its type, and the type of the rest of the record: the record type returns the type of a pair containing the type of the current field and the type of the remaining record.

```
TypeFunc RecordField => string => * => Record => Record

-----------------------------------
RecordField name type r => Record {
  Func "cons" -> type -> r.RecordType : RecordType

  --------------------------------------
  RecordType => (type * r.RecordType)

  -------------------------
  cons x xs -> (x,xs)
}
```

The function `cons` in this case constructs the record field returning a pair with the value of the current field and the rest of the fields. We now have to define how to get and set the value of the fields. The getter is defined as a module that has a type function `GetType` that returns the type of the field to get, and a function `get` that returns the value of the field.

```
Module "Getter" => string => Record : Getter name r {
  TypeFunc "GetType" : *
  Func "get" -> r.RecordType -> GetType
}
```

At this point we use a type function with a premise to generate two different versions of the `Getter`: one if the current field is the one we are looking for and one if the current field is not the one we want to get. The first version simply generates a `Getter` module where the `GetType` function returns the type of the field and the function `get` extracts the value in it.

```
name = lt
--------------------------------
GetField lt (Recordfield name type r) => Getter name r {
  GetType => type
  get (x,xs) -> x
}
```

When the field we are looking for is not the one we are looking at, we have to keep searching in the rest of the record. The function `GetType` calls a new `Getter` that is able to lookup in the rest of the record and calls the `GetType function` of this new getter. The `get` function uses the `get` in the new `Getter` to extract the value of the field.

```
name <> lt
-----------------------------------
GetField lt (RecordField name type r) => Getter name r {
  TypeFunc "Getfield1" : Getter

  -------------------------------
  GetField1 => GetField lt r

  -------------------------------
  GetType => GetField1.GetType

  ---------------------------------
```

```
  get (x,xs) -> GetField1.get xs
}
```

The `Setter` is defined analogously to the `Getter`, except that the function `set` takes as extra argument the value to set.

```
Module "Setter" => string => Record : Setter lt r => {
  TypeFunc "SetType" : *
  Func "set" -> (r.RecordType) -> SetType : (r.RecordType)
}
```

The `Setter` comes as well in two version, one when the field we are looking at is the one we want to modify, and one when the field is not the one we want to change. In the first case the function `SetType` simply returns the type of the field and `set` sets the field with the proper value.

```
TypeFunc "SetField" => string => Record : Record

name = lt
---------------------------------
SetField lt (RecordField name type r) => Setter name r {

  -----------------
  SetType => type

  -------------------
  set (x,xs) v -> (v,xs)
}
```

In the second case we define a new `SetField` which is able to generate the `Setter` for the rest of the record.

```
name <> lt
-----------------------------------------------
SetField lt (RecordField name type r) => Setter name r {
  TypeFunc "SetField1" : Setter

  -------------------------
  SetField1 => SetField lt r

  ---------------------------
  SetType => SetField1.SetType

  -----------------------------------
  set (x,xs) v -> SetField1.set xs v
}
```

## 4.1 Improve the performance of Metacasanova

Here explain the reasons behind performance decay and how this could be avoided. Show that in entities the fields are always in the same order and the structure is never changed, so we could inline directly the getter and setter for the field without using a Map.

## 4.2 Type functions

- Overview of type classes in Haskell and modules in Caml.
- Higher kinded polymorphism

Explain the background about type functions.

## 4.3 Type functions in Meta-casanova

- Explain how to extend Metacasanova with type functions.
- Explain the idea behind the type function inlining.
- Extend the semantics of rule evaluation to include module generation and type function evaluation.
- Records with type functions.

## 4.4 Type function interpeter

Here describe how the inlining process is implemented in the meta-compiler with the type function interpreter

## 4.5 C– optimization

Show how to optimize the current C– implementation by using Type Functions to populate the symbol table.

## 4.6 Casanova 2.5 optimization

- Entity definition with Type Functions.
- Entity traversal with Type Functions (?)
- Rule update definition with Type Functions.
- Evaluation.

Show how to use type functions to define an entity as a Record and how to inline the getter and setter of fields in rules.

# Chapter 5

# Networking primitives in Casanova 2

## 5.1 General idea

- Overview of networking architecture with partially synchronized local game states.
- P2P architecture with *master* and *slave* execution.

There is no "real" game state: every client sees an approximation of the real game state containing only data that matters for networking synchronization. The rest is approximated locally. A client controls directly a portion of the game state (master), the rest is requested remotely to other clients (slave).

## 5.2 Syntax and semantics

- Connection.
- Master and slave rules.
- Semantics of slave rules.

Propagating creation of entities during the connection. Execution of rules depending on the locality of the portion of game state: if the entity is master then we execute the master block, otherwise the slave block.

## 5.3 Send and receive primitives in Casanova 2

- Sending and receiving basic data types.
- Entity synchronization at connection.
- Sending and receiving updates on entities.
- Sending and updating lists.

Basic data types send and receive (should be expanded with custom-defined types). Maps for entities and entity references. Explain how remote

copies of entities are handled. Problems with references in a distributed environment. Handling entities updates with a dictionary to overcome this problem. Creating and updating lists in a distributed environment with incremental updates.

## 5.4 Meta-compiler implementation of networking operators

# Chapter 6

# Final evaluation

## 6.1 Performance evaluation of unoptimized Metacasanova

Evaluation from the INTETAIN paper. Comparison with Python.

## 6.2 Performance evaluation of optimized Metacasanova

## 6.3 Networking evaluation

# Chapter 7

# Discussion and conclusion

Here it would be wise to discuss about the fact that you still need to define a program for a language implemented in Metacasanova in term of syntax of Metacasanova itself. Remark that it is trivial to solve this problem by writing a parser, for example in Yacc, that maps the proper syntax of the language into the syntax of Metacasanova, but we do not implement it because it has no scientific value.