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.

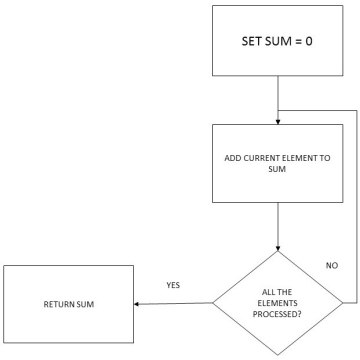


Figure 1.1: Flow chart for the sum of a sequence of numbers

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 [...]:

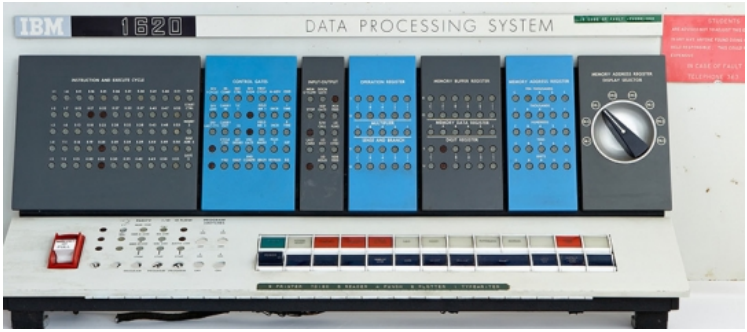


Figure 1.2: Front panel of IBM 1620

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.

```
00075  c7 45 b8 00 00
00 00
0007c  eb 09
0007e  8b 45 b8
00081  83 c0 01
00084  89 45 b8
00087  83 7d b8 0a
0008b  7d 0f
0008d  8b 45 b8
00090  8b 4d c4
00093  03 4c 85 d0
00097  89 4d c4
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.

```
1  VertexShaderOutput VertexShaderSpecularFunction(VertexShaderInput input,
       float3 Normal : NORMAL)
2  {
3    VertexShaderOutput output;
4    float4 worldPosition = mul(input.Position, World);
5    float4 viewPosition = mul(worldPosition, View);
6    output.Position = mul(viewPosition, Projection);
7    float3 normal = normalize(mul(Normal, World));
8    output.Normal = normal;
9    output.View = normalize(float4(EyePosition,1.0f) - worldPosition);
10   return output;
11 }
```

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

> Trying to outsmart a compiler defeats much of the purpose of using one.
>
> Kernighan and Plauger - *The Elements of Programming Style.*

2.1 Architectural overview of a compiler

Compilers are software that read as input a program written in a programming language, called *source language*, and translate it into an equivalent program expressed with another programming language, called *target language*. Usually the target language is machine code, but this is not mandatory. A special kind of compilers are interpreted, that directly execute the program written in the source language rather than translating it into a target language. Some languages, like Java, use a hybrid approach, that is they compile the program into an intermediate language that is later interpreted by a *virtual machine*. Another approach involves the translation into a target high-level language [...].

Although the architecture of a compiler may slightly vary depending on the specific implementation, the translation process usually consists of the following steps:

1. **Lexical analysis:** this phase is performed by a module called *lexer* that is able to process the text and identify the syntactical elements of the language, called *tokens*.

2. **Syntactical analysis:** this phase is performed by a module called *parser*, that checks whether the program written in the source language is compliant to the formal syntax of the language. The parser is tightly coupled with the lexer, as it needs to identify the tokens of the language to correctly process the syntax rules. The parser outputs a

representation of the program, called *Abstract Syntax Tree*, for later use.

3. **Type checking:** this phase is performed by the *type checker* that uses the rules defined by a *type system* to assign a property to the elements of the language called *type*. The types are used to determine whether the abstractions of the language, in a program that is syntactically correct, are used in a meaningful way.

4. **Code generation:** the code generation phase requires to choose one or more target languages to emit. In the latter case, the code generator must have a modular structure to allow to interchange the output language. For this reason this step is usually preceded by an *intermediate code generation* step, that converts the source program into an intermediate representation close to the target language. This phase can later be followed by different kinds of code optimization phases.

In what follows we extensively describe each module that was summarized above.

2.2 Lexer

As stated above, the lexer task is to recognize the *words* or *tokens* of the source language. In order to perform this task the token structure must be expressed in a formal way. Below we present such formalization and we describe the algorithm that actually recognizes the token.

Let us consider a finite alphabet Σ, a *language* is a set of strings, intended as sequences of characters in Σ.

Definition 2.1. A string in a language L in the alphabet Σ is a tuple of characters $\mathbf{a} \in \Sigma^n$.

A notable difference between languages in this context and human-spoken languages is that, in the former, we do not associate a meaning to the words but we are only interested to define which words are part of the language and which are not. Regular expressions are a convenient formalization to define the structure of sets of strings:

Definition 2.2. The following are the possible ways to define regular expressions:

- *Empty:* The regular expression ϵ is a language containing only the empty string.
- *Symbol:* $\forall a \in \Sigma$, $\mathbf{a}$ is a string containing the character a.
- *Alternation:* Given two regular expressions M and N, a string in the language of $M|N$, called alternation, is the sets of strings in the language of M or N.
- *Concatenation:* Given two regular expressions M and N, a string in the language of $M \cdot N$ is the language of strings $\alpha \cdot \beta$ such as $\alpha \in M$ and $\alpha \in N$.
- *Repetition:* Given a regular expression M, its Kleene Closure M^* is formed by the concatenation of zero or more strings in the language M.

The regular expressions defined in Definition 2.2 can be combined to define tokens in a language.

Regular expressions can be processed by using a finite state automaton. Informally a finite state automaton is made of a finite set of states, an alphabet Σ of which it is able to process the symbols, and a set of symbol-labelled edges that connect two states and define how to transition from one state to another. Automata can be divided into two categorise: *non-deterministic finite state automata (NFA)* and *deterministic finite state automata (DFA)*. Formally we have the following definitions:

Definition 2.3. A non-deterministic finite state automaton (NFA) is made of:

- A finite set of states S.
- An alphabet Σ of input symbols.
- A state $s_0 \in S$ that is the starting state of the automaton.
- A set of states $F \subset S$ called final or accepting states.
- A set of transitions $\mathcal{T} \subseteq S \times (\Sigma \cup \{\epsilon\}) \times S$.

Definition 2.4. A deterministic finite state automaton (DFA) is a NFA where the transition is a function, i.e.

$$\tau : S \times \Sigma \to S$$
$$\tau(s_i, c) = s_j$$

and $\nexists\, \tau(s, c_i), \tau(s, c_j) \mid c_i = c_j \; \forall i, j$.

Informally, in NFA's there might be two transitions from the same state that can process the same symbol, while in DFA's for the same state there exists one and only one transition able to process a symbol and no transition processes the empty string. Regular expressions can be converted in NFA by using translation rules. The formalization of the algorithm can be found in [25], here we just show an informal overview for brevity.

2.2.1 Finite state automata for regular expressions

In this section we present an informal overview of the translation rules for regular expressions into NFA's, and an algorithm to convert an NFA into a DFA.

Conversion for Symbols A regular expression containing just one symbol $a \in \Sigma$ can be converted by creating a transition $\tau(s_i, a) = s_j$.

Conversion for concatenation The conversion for concatenation is recursive: the base case of the recursion is the symbol conversion. The conversion of a concatenation of n symbols $a_1 a_2, ..., a_n$ is obtained by adding a transition from the last state of the conversion for the first $n-1$ symbols into a new state through a transition processing the n-th symbol, $\tau(s_{n-1}, a_n) = s_n$.

Conversion for alternation The alternation $M|N$ is obtained by creating an automata with a ϵ-transition into a new state, that we call s_ϵ. From s_ϵ we recursively generate the automata for both M and N. Both automata can finally reach the same state through an ϵ-transition.

Conversion for Kleene closure The Kleene Closure M^* is obtained by initially creating an ϵ-transition into a state s_ϵ. s_ϵ can recursively transition to the automaton for M, which in turn transitions through an ϵ-transition to s_ϵ.

Conversion for M$^+$ The regular expression M^+ contains the concatenation of one or more strings in M. This can be translated by translating $M \cdot M^*$.

Conversion for M? The regular expression $M?$ is a shortcut for $M|\epsilon$, thus it can be translated by using the conversion rule for the alternation.

2.2.2 Conversion of a NFA into a DFA

As stated in Section 2.2, a NFA might have, for the same state, a set of transitions that process the same symbol (including the empty string since ϵ-transitions are allowed). This means that a NFA must be able to guess which transition to follow when trying to process a token. This is not efficient to implement in a computer, thus it is better to use a DFA where there can be only one way of processing a symbol for a given state. An algorithm to automate such conversion exists and is presented in [7] but there exists an algorithm to directly convert regular expressions into DFA's, as shown in [6]. Below we present the algorithm to convert NFA's into DFA's.

The informal idea behind the algorithm is the following: since a DFA cannot contain ϵ-transitions or transitions from one state into another containing the same symbols, we have to construct an automaton that skips the ϵ-transitions and pre-calculates the calculation of the sets of states in advance. In order to do so, we need to be able to compute the *closure* of a set of states. Informally the closure of a set of states S is the sates that can be reached by one of the states of S through an ϵ-transition. The formal definition is given below:

Definition 2.5. The closure $\mathcal{C}(S)$ of a set of states S is defined as

- $\mathcal{C}(S) = S \cup \left(\bigcup_{s \in T} \tau(s, \epsilon) \right)$
- if $\exists\, \mathcal{C}'(S) \mid \mathcal{C}(S) \subseteq \mathcal{C}'(S) \Rightarrow \mathcal{C}'(S) = \mathcal{C}(S)$.

Algorithm 2.1 computes the closure of a set of states. Note that the algorithm termination is granted because we are considering finite-state automata.

At this point we can build the set of all possible states reachable by consuming a specific character. We call this set *edge* of a set of states d.

Algorithm 2.1 Closure of S

$T \leftarrow S$
repeat
  $T' \leftarrow T$
  $T \leftarrow \cup \left(\bigcup_{s \in T'} \tau(s, \epsilon)\right)$
until $T = T'$

Definition 2.6. Let d be a set of states, then the *edge* of d is defined as

$$\mathcal{E}(d, c) = \mathcal{C}\left(\bigcup_{s \in d} \tau(s, c)\right)$$

Now we can use the *closure* and *edge* to build the DFA from a NFA.

Algorithm 2.2 NFA into DFA conversion

$states[0] \leftarrow \emptyset$
$states[1] \leftarrow \mathcal{C}(s_1)$
$p \leftarrow 1$
$j \leftarrow 0$
while $j \leq p$ **do**
  for all $c \in \Sigma$ **do**
    $e \leftarrow \mathcal{E}(states[j], c)$
    if $\exists\, i \leq p \mid e = states[i]$ **then**
      $trans[j, c] \leftarrow i$
    else
      $p \leftarrow p + 1$
      $states[p] \leftarrow e$
      $trans[j, c] \leftarrow p$
    end if
  end for
  $j \leftarrow j + 1$
end while

Algorithm 2.2 performs the conversion into a DFA but we need to adjust it in order to mark the final states of the automaton. A state d is final in the DFA if it is final if any of the states in $state[d]$ is final. In addition to marking final states, we must also keep track of what token is produced in that final state.

2.3 Parser

Regular expressions are a concise declarative way to define the lexical structure of the terms of a language, but they are insufficient to describe its syntax, i.e. how to combine tokens together to make "sentences". A compiler uses the parser module to check the syntactical structure of a program. As we will see more in depth below, the parser is tightly coupled with the lexer, which is used by it to recognize tokens. In order to present the structure of the parser, it is first necessary to introduce *context-free grammars*.

As before we consider a language as a set of tuples of characters taken from a finite alphabet Σ. Informally, a context-free grammar is a set of productions of the form $symbol \rightarrow symbol_1\ symbol_2\ ...symbol_n$, where the left argument can be replaced by the sequence of symbols contained in the right argument. Some productions are *terminal*, meaning that they cannot be replaced any longer, while the others are *non-terminal*. Terminal symbols can only appear on the right side, while non-terminals can appear on both sides. Formally a context free grammar is defined as follows

Definition 2.7. A *context-free grammar* is made of the following elements:

- A set of non-terminal symbols N.
- A finite set of terminal symbols Σ, called *alphabet*.
- A non-terminal symbol $S \in N$ called *starting symbol*.
- A set of productions P in the form $N \rightarrow (N \cup \Sigma)^*$.

Note that Definition 2.7 allows *context-free* grammars to process also regular expression, thus context-free grammars are more expressive then regular expressions. In what follows we assume that the terminal symbols are treated as tokens with regular expressions that can be processed by a lexer, but in general a context-free grammar does not require a lexer DFA to process terminal symbols.

In order to check if a sentence is valid in the grammar defined for a language, we perform a process called *derivation*: starting from the symbol S of the grammar, we recursively replace non-terminal symbols with the right side of their production. The derivation can be done in different ways: we can start expanding the leftmost non-terminal in the production or the rightmost one. The result of the derivation usually generates a data structure called *parse tree* or *abstract syntax tree*, which connects a non-terminal symbol to the symbols obtained through the derivation; the leaves of the tree are terminal symbols.

2.3.1 LR(k) parsers

Simple grammars can be parsed by using *left-to-right parse, leftmost-derivation, k-tokens lookahead*, meaning that the parser processes a symbol by performing a derivation starting from the leftmost symbol of the production, and looking at the first k tokens of a string of the language. The weakness of this technique is that the parser must predict which production to use only knowing the first k tokens of the right side of the production. For instance, consider the two expression

$$(15 * 3 + 4) - 6$$
$$(15 * 3 + 4)$$

and the grammar

$$
\begin{aligned}
&S \to E\ eof\\
&E \to E + T\\
&E \to E - T\\
&E \to T * F\\
&E \to T/F\\
&E \to T\\
&T \to F\\
&F \to id\\
&F \to num\\
&F \to (E)
\end{aligned}
$$

In the first case the parser should use the production $E \to E - T$ while in the second it should use the production $E \to T$. This grammar cannot be parsed by a LL(k) parser because it is not possible to decide which of the two productions must be used just by looking at the first k leftmost tokens. Indeed expressions of that form could have arbitrary length and the lookahead is, in general, insufficient. In general LL(k) grammars are context-free, but not all context-free grammars are LL(k), so such a parser is unable to parse all context-free grammars.

A more powerful parser is the *left-to-right parse, rightmost-derivation, k-tokens lookahead* or LR(k). This parse maintains a *stack* and an *input* (which is the sentence to parse). The first k tokens of the input are the *lookahead*. The parser uses the stack and the lookahead to perform two different actions:

- *Shift*: The parser moves the first input token to the top of the stack.
- *Reduce*: The parser chooses a grammar production $N_i \to s_1\ s_2\ ...s_j$ and pop $s_j, s_{j-1}, ..., s_1$ from the top of the stack. It then pushes N_i at the top of the stack.

The parser uses a DFA to know when to apply a shift action or a reduce action. The DFA is insufficient to process the input, as DFA's are not capable of processing context-free grammars, but it is applied to the stack. The DFA contains edges labelled by the symbols that can appear in the stack, while states contain one of the following actions:

- s_n: shift the symbol and go to state n.
- g_n: go to state n.
- r_k: reduce using the production k in the grammar.
- a: accept, i.e. shift the end-of-file symbol.
- $error$: invalid state, meaning that the sentence is invalid in the grammar.

The automaton is usually represented with a tabular structure, which is called *parsing table*. The element $p_{i,s}$ in the table represents the transition from state i when the symbol at the top of the stack is s.

In order to generate the parsing table (or equivalently the DFA for the parser) we need two support functions, one to generate the possible states the automaton can reach by using grammar productions, and one to generate the actions to advance past the current state. We introduce an additional notation to represent the situation where the parser has reached a certain position while deriving a production.

Definition 2.8. An *item* is any production in the form $N \to \alpha.X\beta$, meaning that the parser is at the position indicated by the dot where X is a grammar symbol.

At this point we are able to define the *Closure* function, that adds more items to a set of items when the dot is before a non-terminal symbol, which is shown in Algorithm 2.3. Note that, for brevity, we present the version to generate a LR(0) parser, for a LR(1) parser a minor adjustment must be made.

Algorithm 2.3 Closure function for a LR(0) parser

```
function CLOSURE(I)
    repeat
        for all N → α.Xβ ∈ I do
            for all X → γ do
                I ← I ∪ {X → .γ}
            end for
        end for
    until I′ ≠ I
    return I
end function
```

The algorithm starts with an initial set of items I and adds all grammar productions that contain X as left argument as items with the dot at the beginning of their right argument, meaning that the symbols of the production must still be completely parsed.

Now we need a function that, given a set of items, is able to advance the state of the parser past the symbol X. This is shown in Algorithm 2.4.

Algorithm 2.4 Goto function for a LR(0) parser

```
function GOTO(I, X)
    J ← ∅
    for all N → α.Xβ ∈ I do
        J ← J ∪ {N → αX.β}
    end for
    return CLOSURE(J)
end function
```

The algorithm starts with a set of items and a symbol X and creates a new set of items where the parser position has been moved past the symbol X. It then compute the closure of this new set of items a returns it.

We can now proceed to define the algorithm to generate the LR(0) parser, which is shown in Algorithm 2.5. The initial state is made of all the productions where the left side is the starting symbol, which is equivalent to compute the closure of $S' \to .S\ eof$. It then proceeds to expand the set of states and the set of actions to perform. Note that we never compute GOTO(I, eof) but we simply generate an *accept* action. Now, for all actions in E where X is a terminal, we generate a shift action at position (I, X), for all actions where X is non-terminal we put a goto action at position (I, X),

and finally for a state containing an item $N_k \to \gamma.$ (the parser is at the end of the production) we generate a r_k action at (I, Y) for every token Y.

In general parsing tables can be very large, for this reason it is usually wise to implement a variant of LR(k) parsers called LALR(k) parsers, where all states that contain the same actions but different lookaheads are merged into one, thus reducing the size of the parsing table. LR(1) and LALR(1) parsers are very common, since most of the programming languages can be defined by a LR(1) grammar. For instance, the popular family of parser generators `Yacc` produces LALR(1) parsers.

Algorithm 2.5 LR(0) parser generation

$T \leftarrow$ CLOSURE($\{S' \to .S\ eof\}$)
$E \leftarrow \emptyset$
repeat
  $T' \leftarrow T$
  $E' \leftarrow E$
  for all $I \in T$ **do**
    for all $N \to \alpha.X\beta\ \in I$ **do**
      $J \leftarrow$ GOTO(I, X)
      $T \leftarrow T \cup \{J\}$
      $E \leftarrow E \cup \{I \xrightarrow{X} J\}$
    end for
  end for
until $E' = E$ **and** $T' = T$

2.3.2 Parser generators

2.3.3 Monadic parsers

Monadic parsing is an alternative to traditional parsers, such as LR(k) and LALR(k) presented above. Monadic parsers have inferior performance with respect to LR(k) and LALR(k) [21] parsers but they are extensible, i.e. they do not rely on a limited set of combinators to describe the grammar of language as for parser generators. Monadic parsers were extensively explained in [21, 31], here we present a variation that can deal also with error handling. Before explaining how to implement a monadic parser, we introduce the concept of Monad:

Definition 2.9. A *Monad* is a tern made of the following elements:

- A type constructor M.
- A unary operation $Return\ ::\ a \to M\ a$.
- A binary operation $Bind\ ::\ M\ a \to (a \to M\ b) \to M\ b$. The bind can also be written by using the symbol $>>=$.

where both operations satisfy the following properties:

- $a >>= return \equiv a$.
- $(a >>= f) >>= g \equiv a >>= (\lambda x.fx >>= g)$.

We now proceed to define a parser monad by defining (*i*) the type constructor for the parser, (*ii*) the unary operator, (*iii*) the binary operator, and (*iv*) parser combinators as an example of the extensibility of the parser monad. Note that below we provide an implementation in F#, which does not have type classes as Haskell, so the parser monad does not use any type argument and directly defines the operators for this specific instance of monad.

Parser type constructor and monadic operations A parser is defined in literature as a function that takes as input a text and returns a list of pairs made of the parsing result and the rest of the text to process. The parsing result is usually the syntax tree generated by the parser. The result is a list because the same syntactical structure might be processed in different ways. By convention, an empty list denotes a parser failure. Here we propose a variation of this traditional implementation in order to provide a better error report.

In this alternative implementation, the parser is a function that takes as input the text to process, a *parsing context* that might hold auxiliary information necessary for the parsing, the current position of the parser in the text, and returns either a tuple containing the parsing result, the text left to process, an updated context, and the updated position, or an error in case of a parser failure.

```
type Parser<'a, 'ctxt> = { Parse : List<char> -> 'ctxt -> Position ->
     Either<'a * List<char> * 'ctxt * Position, Error>}

static member Make(p:List<char> -> 'ctxt -> Position -> Either<'a * List
     <char> * 'ctxt * Position, Error>) : Parser<'a,'ctxt> = { Parse = p
      }
```

The *return* operation should take as input a generic value of type `'a` and return a `Parser<'a,'ctxt>`. The return simply creates the parser function for the given input:

```
member this.Return(x:'a) : Parser<'a,'ctxt> =
  (fun buf ctxt pos -> First(x, buf, ctxt, pos)) |> Parser.Make
```

According to the Definition 2.9, the bind operator must take as input a `Parser<'a>`, a function `'a -> Parser<'b>` and return `Parser<'b>`. The bind generates a function that runs the input parser on the text. The result of the input parser can, according to its definition, contain a parsing result or an error in case of failure. The function generated by the bind must be able to handle these two situations: in case of a correct result the function creates a new parser using the parsing result and runs it on the remaining portion of the text, while in case of an error it simply outputs the error. In this way, when parsing fails, the error will be propagated ahead.

```
member this.Bind(p:Parser<'a,'ctxt>, k:'a->Parser<'b,'ctxt>) : Parser<'b
     ,'ctxt> =
(fun buf ctxt pos ->
  let all_res = p.Parse buf ctxt pos
  match all_res with
  | First p1res ->
      let res, restBuf, ctxt', pos' = p1res
      (k res).Parse restBuf ctxt' pos'
  | Second err -> Second err ) |> Parser.Make
```

Parser combinators With the parser monad implemented above, we can implement several parser combinators that can be used to define the grammar of a language. Here we show only a small glimpse of the possible combinators that can be implemented.

The first parser combinator that we present is the *choice*. The choice takes as input two parsers and runs the first. If the first parser succeeds than its result is returned, otherwise the second is run. If it succeeds its result is return, otherwise the whole parser outputs an error. This combinator is useful, for instance, when there might be two possible choices for a token in a statement. For instance, in either Java or C# is possible to exchange the order of the access modifier and the static modifier in the method declaration, thus both `public static` or `static public` are valid combinations. This combinator would try to parse the declaration in the first way, and if it fails it will try also the second option. Of course if the syntax of both combinations is wrong the parser will fail completely. The code for the combinator is shown below:

```
static member (++) (p1:Parser<'a,'ctxt>, p2:Parser<'a,'ctxt>) : Parser<'
    a,'ctxt> =
  (fun buf ctxt p ->
  match p1.Parse buf ctxt p with
  | Second err1 ->
      match p2.Parse buf ctxt p with
      | Second err2 -> Second err2
      | p2res -> p2res
  | p1res -> p1res) |> Parser.Make
```

A useful variation of this combinator, is the one that executes two parsers with different generic types and returns a `Either` data type, containing either the result of the first or the second.

```
static   member (+) (p1:Parser<'a,'ctxt>, p2:Parser<'b,'ctxt>) : Parser<
    Either<'a,'b>,'ctxt> =
  (fun buf ctxt p ->
   match p1.Parse buf ctxt p with
   | Second err1 ->
       match p2.Parse buf ctxt p with
       | Second err2 -> Second(err2)
       | First p2res ->
           let res,restBuf,ctxt',pos = p2res
           First(Second res, restBuf, ctxt', pos)
   | First p1res ->
       let res, restBuf, ctxt', pos = p1res
       First((First res), restBuf, ctxt', pos)) |> Parser.Make
```

Other combinators are possible, but for brevity we have only shown two. It should appear clear how this approach is completely extensible with no limitations. Any combinator would take as input two parsers and define the type of the resulting parser. The implementation will contain the logic to combine two parsers together. For example, another parser combinator is the application of 0 or more times of the same parser.

To complete this discussion, we now show how to parse a specific character and a keyword. The parser for a character takes as input the text to process and the character to match. If the input text is empty of course the parser immediately fails because no character will ever be matched. Otherwise if the first character of the text matches the one provided then we

return the matched character as result and the rest of the text to process, otherwise we output an error. The function also takes care of updating the position of the parser accordingly and to skip line breaks.

```
let character(c:char) : Parser<char, 'ctxt> =
  (fun buf ctxt (pos:Position) ->
   match buf : List<char> with
   | x::cs when x = c ->
       let pos' =
       if x = '\n' then
         pos.NextLine
       else
         pos.NextCol
         First( c, cs, ctxt, pos')
   | _ ->
      Second (Error(pos, sprintf "Expected character %A" c))) |> Parser.
           Make
```

The word parser takes as input the text to process and the word to match. It then applies the character parser to the word until it has all been processed. In the code below the syntax `let! x = y` is a syntactical sugar for `y >>= fun x -> ...` in the fashion of Haskell `do` notation.

```
let rec word (w:List<char>) : Parser<List<char>, 'ctxt> =
  p{
    match w with
    | x::xs ->
        let! c = character x
        let! cs = word xs
        return c::cs
    | [] ->
        return []
  }
```

2.4 Type systems and type checking

2.5 Operational semantics

2.6 Meta-compilers

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

Describe what it is existing in literature.

Chapter 3

Metacasanova

3.1 Repetitive steps in compilers development

In Section **??** we briefly stated that the process of developing a compiler includes several steps that are repetitive, i.e. their behaviour is always the same regardless of the language for which the compiler is built. In this section we show in what way this process is repetitive and what is the common pattern

3.1.1 Type checking

Type systems are generally expressed in the form of logical rules [13], made of a set of premises, that must be verified in order to assign to the language construct the type defined in the conclusion. For example the following rule defines the typing of an `if-then-else` statement in a functional programming language:[1]

$$\frac{\Gamma \vdash c : \mathit{bool} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{if } c \text{ then } t \text{ else } e : \tau}$$

In this rule Γ is the environment. The type rule first evaluates the premises, which means that if the condition of the `if-then-else` has type `bool` and both `then` and `else` blocks have the same type, then the whole `if-then-else` has the type of either blocks.

Typing a construct of the language requires to evaluate its corresponding typing rule. In order to do so, the behaviour of each typing rule must be implemented in the host language in which the compiler is defined. Independently of the chosen language, the behaviour will always be the following : (*i*) evaluate a premise, (*ii*) if the evaluation of the premise fails then the construct fails the type check and an error is returned, (*iii*) repeat step 1 and 2 until all the premises have been evaluated, and (*iv*) assign the type to the construct that is defined in the rule conclusion.

[1] Note that the type rule of `if-then-else` in an imperative programming language is different.

3.1.2 Semantics

Semantics define how the language abstractions behave and can be expressed in different ways, for example with a term-rewriting system [23] or with the operational semantics [17]. For the scope of this work, we choose to rely on the operational semantics. The definition of the operational semantics of a language abstraction is, again, in the form of a logical rule where the conclusion (which is the final behaviour of the construct) is achieved if the evaluation of the premises lead to the desired results. For instance, the operational semantics of a while loop could be the following:

$$\frac{\langle c \rangle \Rightarrow \texttt{true}}{\langle \text{while } c \text{ do } L \ ; \ k \rangle \Rightarrow \langle L \ ; \text{ while } c \text{ do } L \ ; \ k \rangle} \qquad \frac{\langle c \rangle \Rightarrow \texttt{false}}{\langle \text{while } c \text{ do } L \ ; \ k \rangle \Rightarrow \langle k \rangle}$$

Again, the behaviour of the semantics rule must be encoded in the host language in which the compiler is being developed, but the pattern it follows is always the same. This step, depending on the implementation choice, might also require to translate this behaviour into an *intermediate language* representation that is more suitable for the subsequent code generation phase.

3.1.3 Discussion

The examples above show how the behaviour of the type checking and semantics rules must be hard-coded in the language chosen for the compiler implementation, regardless of the fact that their pattern is constantly repeated in every rule. This pattern can be captured in a meta-language that is able to process the type system and operational semantics definition of the language and produce the code to execute the behaviour of the rules. In this work we describe the meta-language for *Metacasanova*, a meta-compiler that is able to read a program written in terms of type system/operational semantics rules defining a programming language, a program written in that language, and output executable code that mimics the behaviour of the semantics. Such a language relieves the programmer from writing boiler-plate code when implementing a compiler for a (Domain-Specific) language. For this reason we formulate the following research question:

Research question 1: *To what extent Metacasanova eases the development speed of a compiler for a Domain-Specific Language, in terms of code length compared to the hard-coded implementation, and how much does the abstraction layer of the Metacompiler affect the performance of the generated code?*

Another problem that arises when using meta-compilers is the performance decay given by the introduction of their additional abstraction layer. One of the reasons for this performance decay (see Section 4.2.5) is that the meta-language (and thus the meta-type system) is unaware of the type system and the memory model of the language implemented in the meta-compiler. For this reason, checking the types and accessing the memory requires to dynamically look up a symbol table defined with the abstractions provided by the meta-language. The need for performance is for Meta-

casanova important because it is being used to extend the DSL for games *Casanova* [4, 3]. Thus, we formulate a second research question:

Research question 2: *In what way can we embed the type system of the implemented language in Metacasanova in order to get rid of the dynamic lookups at runtime and what is the performance gain of this optimization?*

We try to answer these two research questions by using a two-steps methodology: (*i*) we present an architecture for Metacasanova aimed to automate the process of code generation, and then (*ii*) we propose a language extension to embed the implemented language type system in the meta-type system of Metacasanova.

3.1.4 Related work

RML [26] is a meta-compiler based on operational semantics that is similar to Metacasanova. Its syntax is very close to that of ML and it generates C code. A notable effort was done to optimize the tail calls in the generated code for the rules, but the problem arisen by Research Question 2 is not addressed.

Stratego [11] is a meta-compiler based on a transformation system. A transformation language consists of a series of constructor calls to construct the terms of the grammar and functions that specify how to evaluate the terms. Stratego is not a typed language, so it does not ensure that the terms and transformation functions are used consistently.

A language extension for Haskell involving *template meta-programming* exists [28]. Although a valuable and elegant approach, using Haskell language extensions is not suitable for domain-specific languages for games due to the wide use of monads a lambda abstractions, which greatly affect the performance, and the lazy nature of Haskell that affects the memory usage. In Section **??** we underline how this project was born to ease the extension of a domain-specific language for game development, thus this was not a suitable choice for our initial goals.

Syntax Macro meta-programming [12] is an approach that operates during the parsing phase. Macros are used to produce an abstract syntax tree that is replaced when the macro is invoked. One notable example of this kind of
meta-programming can be found in the language Lisp. Macros guarantee syntactic safety [32] but not semantics safety, since no meta-type system is available for macros.

3.1.5 Requirements of Metacasanova

In order to relieve programmers of manually defining the behaviour described in Section **??** in the back-end of the compiler, we propose the following features for Metacasanova:

- It must be possible to define custom operators (or functions) and data containers. This is needed to define the syntactic structures of the language we are defining.

- It must be typed: each syntactic structure can be associated to a specific type in order to be able to detect meaningless terms (such as adding a string to an integer) and notify the error to the user.
- It must be possible to have polymorphic syntactical structures. This is useful to define equivalent "roles" in the language for the same syntactical structure; for instance we can say that an integer literal is both a *Value* and an *Arithmetic expression*.
- It must natively support the evaluation of semantics rules, as those shown above.

We can see that these specifications are compatible with the definition of meta-compiler, as the software takes as input a language definition written in the meta-language, a program for that language, and outputs runnable code that mimics the code that a hard-coded compiler would output.

3.1.6 General overview

A Metacasanova program is made of a set of `Data` and `Function` definitions, and a sequence of rules. A data definition specifies the constructor name of the data type (used to construct the data type), its field types, and the type name of the data. Optionally it is possible to specify a priority for the constructor of the data type. For instance this is the definition of the sum of two arithmetic expression

```
Data Expr -> "+" -> Expr : Expr
```

Note that Metacasanova allows you to specify any kind of notation for data types in the language syntax, depending on the order of definition of the argument types and the constructor name. In the previous example we used an infix notation. The equivalent prefix and postfix notations would be:

```
Data "+" -> Expr -> Expr : Expr
Data Expr -> Expr -> "+" : Expr
```

A function definition is similar to a data definition but it also has a return type. For instance the following is the evaluation function definition for the arithmetic expression above:

```
Func "eval" -> Expr : Value
```

In Metacasanova it is also possible to define polymorphic data in the following way:

```
Value is Expr
```

In this way we are saying that an atomic value is also an expression and we can pass both a composite expression and an atomic value to the evaluation function defined above.

Metacasanova also allows to embed C# code [2] into the language by using double angular brackets. This code can be used to embed .NET types when

[2] See Section ?? for the motivation.

defining data or functions, or to run C# code in the rules. For example in the following snippets we define a floating point data which encapsulates a floating point number of .NET to be used for arithmetic computations:

```
Data "$f" -> <<float>> : Value
```

A rule in Metacasanova, as explained above, may contain a sequence of function calls and clauses. In the following snippet we have the rule to evaluate the sum of two floating point numbers:

```
eval a => $f c
eval b => $f d
<<c + d>> => res
------------------------
eval (a + b) => $f res
```

Note that if one of the two expressions does not return a floating point value, then the entire rule evaluation fails. Also note that we can embed C# code to perform the actual arithmetic operation. Metacasanova selects a rule by means of pattern matching (in order of declaration of rules) on the function arguments. This means that both of the following rules will be valid candidates to evaluate the sum of two expressions:

```
...
---------------
eval expr => res

...
----------------
eval (a + b) => res
```

Finally the language supports expression bindings with the following syntax:

```
x := $f 5
```

3.1.7 Formalization

In what follows we assume that the pattern matching of the function arguments in a rule succeeds, otherwise a rule will fail to return a result. The informal semantics of the rule evaluation in Metacasanova is the following:

R1 A rule with no clauses or function calls always returns a result.

R2 A rule returns a result if all the clauses evaluate to `true` and all the function calls in the premise return a result.

R3 A rule fails if at least one clause evaluates to `false` or one of the function calls fails (returning no results).

We will express the semantics, as usual, in the form of logical rules, where the conclusion is obtained when all the premises are true. In what follows we consider a set of rules defined in the Metacasanova language R. Each rule has a set of function calls F and a set of clauses (boolean expressions) C. We use the notation f^r to express the application of the function f through the rule r. We will define the semantics by using the notation $\langle expr \rangle$ to

mark the evaluation of an expression, for example $\langle f^r \rangle$ means evaluating the application of f through r. The following is the formal semantics of the rule evaluation in Metacasanova, based on the informal behaviour defined above:

$$\text{R1: } \frac{\begin{array}{c} C = \emptyset \\ F = \emptyset \end{array}}{\langle f^r \rangle \Rightarrow \{x\}}$$

$$\text{R2: } \frac{\begin{array}{c} \forall c_i \in C \,, \langle c_i \rangle \Rightarrow true \\ \forall f_j \in F \,, \exists r_k \in R \mid \langle f_j^{r_k} \rangle \Rightarrow \{x_{r^k}\} \end{array}}{\langle f^r \rangle \Rightarrow \{x_r\}}$$

$$\text{R3(A): } \frac{\exists c_i \in C \mid \langle c_i \rangle \Rightarrow false}{\langle f^r \rangle \Rightarrow \emptyset}$$

$$\text{R3(B) } \frac{\forall r_k \in R \,, \exists f_j \in F \mid \langle f_j^{r_k} \rangle \Rightarrow \emptyset}{\langle f^r \rangle \Rightarrow \emptyset}$$

R1 says that, when both C and F are empty (we do not have any clauses or function calls), the rule in Metacasanova returns a result. R2 says that, if all the clauses in C evaluates to true and, for all the function calls in F we can find a rule that returns a result (all the function applications return a result for at least one rule of the program), then the current rule returns a result. R3(a) and R3(b) specify when a rule fails to return a result: this happens when at least one of the clauses in C evaluates to false, or when one of the function applications does not return a result for any of the rules defined in the program.

In the following section we describe how the code generation process works, namely how the `Data` types of Metacasanova are mapped in the target language, and how the rule evaluation is implemented.

In Section **??** we defined the syntax and semantics of Metacasanova. In this section we explain how the abstractions of the language are compiled into the generated code. We chose C# as target language because the development of Metacasanova started with the idea of expanding the DSL for game development Casanova with further functionalities. Casanova hard-coded compiler generated C# code as well because it is compatible with game engines such as Unity3D and Monogame. At the same time, C# grants decent performance without having to manually manage the memory such as for lower-level languages like C/C++. Code generation in different target languages is possible but still an ongoing project (see Section **??**).

3.2 Parsing

3.3 Type checking

3.4 Code generation

3.4.1 Data structures code generation

The type of each data structure is generated as an interface in C#. Each data structure defined in Metacasanova is mapped to a `class` in C# that implements such interface. The class contains as many fields as the number of arguments the data structure contains. Each field is given an automatic name `argC` where `C` is the index of the argument in the data structure definition. The data structure symbols used in the definition might be preprocessed and replaced in order to avoid illegal characters in the C# class definition. The class contains an additional field that stores the original name of the data structure before the replacement is performed, used for its "pretty print". For example the data structure

```
Data "$i" -> int : Value
```

will be generated as

```
public interface Value {  }

public class __opDollari : Value
{
public string __name = "$i";
public int __arg0;

public override string ToString()
{
return "(" + __name + " " + __arg0 + ")";
}
}
```

3.4.2 Code generation for rules

Each rule contains a set of premises that in general call different functions to produce a result, and a conclusion that contains the function evaluated by the current rule and the result it produces. The code generation for the rules follows the steps below:

1. Generate a data structure for each function defined in the metaprogram.

2. For each function f extract all the rules whose conclusion contains f.

3. Create a `switch` statement with a case for each rule that is able to execute the function (the function is in its conclusion).

4. In the case block of each rule, define the local variables defined in the rule.

5. Apply pattern matching to the arguments of the function contained in the conclusion of the rule. If it fails, jump immediately to the next case (rule).

6. Store the values passed to the function call into the appropriate local variables.

7. Run each premise by instantiating the class for the function used by it and copying the values into the input arguments.

8. Check if the premise outputs a result and, in the case of an explicit data structure argument, check the pattern matching. If the premise result is empty or the pattern matching fails for all the possible executions of the premise then jump to the next case.

9. Generate the result for the current rule execution.

In what follows, we use as an example the code generation for the following rule (which computes the sum of two integer expressions in a programming language):

```
eval a -> $i c
eval b -> $i d
<< c + d >> -> e
----------------
eval (a + b) -> $i e
```

From now on we will refer to an argument as *explicit data argument* when its structure appears explicitly in the conclusion or in one of the premises, as in the case of `a + b` in the example above.

Data structure for the function

As first step the meta-compiler generates a class for each function defined in the meta-program. This class contains one field for each argument the function accepts. It also contains a field to store the possible result of its evaluation. This field is a `struct` generated by the meta-compiler defined as follows:

```
public struct __MetaCnvResult<T> { public T Value; public bool HasValue;
      }
```

The result contains a boolean to mark if the rule actually returned a result or failed, and a value which contains the result in case of success.

For example, the function

```
Func eval -> Expr : Value
```

will be generated as

```
public class eval
{
public Expr __arg0;
public __MetaCnvResult<Value> __res;
...
}
```

Rule execution

The class defines a method `Run` that performs the actual code execution. The meta-compiler retrieves all the rules whose conclusion contains a call to the current function, which define all the possible ways the function can be evaluated with. It then creates a `switch` structure where each `case` represents each rule that might execute that function. The result of the rule is also initialized here (the `struct` will contain a default value and the boolean flag will be set to `false`). Each `case` defines a set of local variables, that are the variables used within the scope of that rule.

Local variables definitions and pattern matching of the conclusion

At the beginning of each `case`, the meta-compiler defines the local variables initialized with their respective default values. It also generates then the code necessary for the pattern-matching of the conclusion arguments. Since variables always pass the pattern-matching, the code is generated only for arguments explicitly defining a data structure (see the examples about arithmetic operators in Section **??**) and literals. If the pattern matching fails then the execution jumps to the next `case` (rule). For instance, the code for the following conclusion

```
...
-------------
eval (a + b) -> $i e
```

is generated as follows

```
case 0:
{
Expr a = default(Expr);
Expr b = default(Expr);
int c = default(int);
int d = default(int);
int e = default(int);
if (!(__arg0 is __opPlus)) goto case 1;
...
}
```

Note that an explicit data argument, such in the example above, might contain other nested explicit data arguments, so the pattern-matching is recursively performed on the data structure arguments themselves.

Copying the input values into the local variables

When each function is called by a premise, the local values are stored into the class fields of the function defined in Section 3.4.2. These values must be copied to the local variables defined in the `case` block representing the rule. Particular care must be taken when one argument is an explicit data. In that case, we must copy, one by one, the content of the data into the local variables bound in the pattern matching. For example, in the rule above, we must separately copy the content of the first and second parameter of the explicit data argument into the local variables `a` and `b`. The generated code for this step, applied to the example above, will be:

```
__opPlus __tmp0 = (__opPlus)__arg0;
a = __tmp0.__arg0;
b = __tmp0.__arg1;
```

Note that the type conversion from the polymorphic type `Expr` into `opPlus` is now safe because we have already checked during the pattern matching that we actually have `opPlus`.

Generation of premises

Before evaluating each premise, we must instantiate the class for the function that they are invoking. The input arguments of the function call must be copied into the fields of the instantiated object. If one of the arguments is an explicit data argument, then it must be instantiated and its arguments should be initialized, and then the whole data argument must be assigned to the respective function field. After this step, it is possible to invoke the `Run` method of the function to start its execution. The first premise of the example above then becomes (the generation of the second is analogous):

```
eval a -> $i c
```

```
eval __tmp1 = new eval();
__tmp1.__arg0 = a;
__tmp1.Run();
```

Checking the premise result

After the execution of the function called by a premise, we must check if a rule was able to correctly evaluate it. In order to do so, we must check that the result field of the function object contains a value, and if not the rule fails and we jump to the next case (rule), which is performed in the following way:

```
if (!(__tmp1.__res.HasValue)) goto case 1;
```

If the premise was successfully evaluated by one rule, then we must check the structure of the result, which leads to the following three situations:

1. The result is bound to a variable.

2. The result is constrained to be a literal.

3. The result is an explicit data argument.

In the first case, as already explained above, the pattern matching always succeeds, so no check is needed. In the second case, it is enough to check the value of the literal. In the last case, all the arguments of the data argument must be checked to see if they match the expected result. In general this process is recursive, as the arguments could be themselves other explicit data arguments. If the result passes the check, then the result is copied into the local variables, in a fashion similar to the one performed for the function premise. For instance, for the premise

```
eval a -> $i c
```

the meta-compiler generates the following code to check the result

```
if (!(__tmp1.__res.Value is __opDollari)) goto case 1;
__MetaCnvResult<Value> __tmp2 = __tmp1.__res;
__opDollari __tmp3 = (__opDollari)__tmp2.Value;
c = __tmp3.__arg0;
```

Generation of the result

When all premises correctly output the expected result, the rule can output the final result. In order to do that, the generated code must copy the right part of the conclusion (the result) into the `res` variable of the function class. If the right part of the conclusion is, again, an explicit data argument, then the data object must first be instantiated and then copied into the result. For example the result of the rule above is generated as follows:

```
res = c + d;
__opDollari __tmp7 = new __opDollari();
__tmp7.__arg0 = res;
__res.HasValue = true;
__res.Value = __tmp7;
break;
```

After this step, the rule evaluation successfully returns a result.

This implementation choice is due to the fact that we plan to support partial function applications, thus, when a function is partially applied, there is the need to store the values of the arguments that were partially given. This could still be implemented with static methods and lambdas in C#, but not all programming languages natively support lambda abstractions, so we chose to have a set-up that allows us to change the target language without dramatically altering the logic of code generation.

Chapter 4

Language design in Metacasanova

4.1 The C-- language

4.2 Casanova 2.5 in Metacasanova

In this section we will briefly introduce the Casanova language, a domain specific language for games. We then show a re-implementation, which we call Casanova 2, of the Casanova 2 language hard-coded compiler as an example of use of Metacasanova.

4.2.1 The Casanova language

Casanova 2.5 is a language oriented to video game development which is based on Casanova 2 [5]. A program in Casanova is a tree of *entities*, where the root is marked in a special way and called *world*. Each entity is similar to a *class* in an object-oriented programming language: it has a constructor and some fields. The fields do not have access modifiers because they are not directly modifiable from the code except with a specific statement. Each entity also contains a list of *rules*, that are methods that are ticked in order with a specific refresh rate called `dt`. Each rule takes as input four elements: `dt`, `this`, which is a reference to the current entity, `world` that is a reference to the world entity, and a subset of entity fields called *domain*. A rule can only modify the fields contained in the domain. The rules can be paused for a certain amount of seconds or until a condition is met by using the `wait` statement. It is possible to modify the values of the fields in the domain by using the `yield` statement which takes as input a tuple of values to assign to the fields. When the `yield` statement is executed the rule is paused until the next frame. Also the body of control structures (`if-then-else`, `while`, `for`) is interruptible. In the following section we show the implementation of Casanova 2.5 in Metacasanova.

4.2.2 Casanova 2.5

The memory in Casanova 2.5 is represented using three maps, where the key is the variable/field name, and the value is the value stored in the variable/field. The first dictionary represents the global memory (the fields of the `world` entity or *Game State*), the second dictionary represents the current entity fields, and the third the variable bindings local to each rule.

The core of the entity update is the `tick` function. This function evaluates in order each rule in the entity by calling the `evalRule` function. This function executes the body of the rule and returns a result depending on the set of statements that has been evaluated. This result is used by `tick` to update the memory and rebuild the rule body to be evaluated at the next frame. The result of `tick` is a `State` containing the rules updated so far, and the updated entity and global fields. Since a rule must be restarted after the whole body has been evaluated, we need to store a list containing the original rules, which will be restored when evaluation returns `Done` (see below). At each step the function recursively calls itself by passing the remaining part of original rules (the rules which body was not altered by the evaluation of the statements) and modified rules (which body has been altered by the evaluation of the statements) to be evaluated. The function stops when all the rules have been evaluated, and this happens when both the original and the modified rule lists are empty.

Interruption is achieved by using *Continuation passing style*: the execution of a sequence of statements is seen as a sequence of steps that returns the result of the execution and the remaining code to be executed. Every time a statement is executed we rebuild a new rule whose body contains the continuation which will be evaluated next.

The possible results returned by the `tick` function are the following: (*i*) `Suspend` contains a `wait` statement with the updated timer, the continuation, and a data structure called `Context` which contains the updated local variables, the entity fields, and the global fields. The function rebuilds a rule which body is the sequence of statements contained by the `Suspend` data structure. (*ii*) `Resume` is returned when the timer must resume after the last waited frame. In order not to skip a frame we must still re-evaluate the rule at the next frame and not immediately. In this case the argument of `Resume` is only the remaining statements to be executed. (*iii*) `Yield` stops evaluation for one frame. We use the continuation to rebuild the rule body. Memory is updated by `evalRule`. (*iv*) `Done` stops the evaluation for one frame and rebuilds the original rule body by taking it from the original rules list.

For brevity we write only the code for `Suspend`. A full implementation can be found at [18]. You can see a schematic representation of the tick function in Figure 4.1.

```
evalRule (rule dom body k locals delta) fields globals => Suspend (s;
     cont) (Context newLocals newFields newGlobals)
r := rule dom s cont newLocals dt
tick originals rs newFields newGlobals dt => State updatedRules
     updatedFields updatedGlobals
st := State (r::updatedRules) updatedFields updatedGlobals
------------------------------------------------------
tick (original::originals) ((rule dom body k locals delta)::rs) fields
     globals dt => st
```

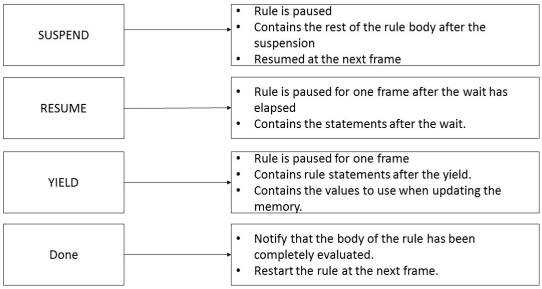


Figure 4.1: Casanova 2.5 rule evaluation

The function `evalRule` calls `evalStatement` to evaluate the first statement in the body of the rule passed as argument. The result of the evaluation of the statement is processed in the following way: (*i*) if the result is `Done`, `Suspend` or `Resume` then it is just returned to the caller function. We omit the code for this case, since it is trivial; (*ii*) if the result is `Atomic` it means that the evaluated statement was uninterruptible and the remaining statements of the rule must be re-evaluated immediately; (*iii*) if the result is `Yield` then the fields in the domain are updated recursively in order and then the updated memory is encapsulated in the `Yield` data structure and passed to the caller function.

```
evalStatement b k ctxt dt => Atomic z c
evalRule (rule dom z nop c dt) => res
--------------------------------
evalRule (rule dom b k ctxt dt)  => res
```

```
evalStatement b k (Context locals fields globals) dt => Yield ks values
     context
updateFields dom values context  => updatedContext
-----------------------------------------------------------
evalRule (rule dom b k locals dt) fields globals => Yield ks values
     updatedContex
```

Note that, in case of a rule containing only atomic statements, we will eventually return `Done` after having recursively called `evalStatement` for all the statements, and the rule will be paused for one frame.
The `evalStatement` function is used both to evaluate a single statement and a sequence of statements. When evaluating a sequence of statements, the first one is extracted. A continuation is built with the following statement and passed to a recursive call to `evalStatement` which evaluates the extracted statement. If the existing continuation is non-empty, then it is added before the current continuation. If both the continuation and the body are empty (situation represented by the `nop` operator) then it means the rule evaluation has been completed and we return `Done`.

```
a != nop
---------------------                              ------------------------
addStmt a b => a;b                                 addStmt nop nop => nop

addStmt b k => cont
evalStatement a cont ctxt dt => res
-------------------------------
     -----------------------------------
evalStatement (a;b) k ctxt dt => res               evalStatement nop nop
     ctxt dt => Done ctxt
```

We will now present, for brevity, only the evaluation of the `wait` and `yield` statements. Both the evaluation of the control structures and the variable bindings always return `Atomic` because they do not, by definition, pause the execution of the rule.

The `wait` statement has two different evaluations, based on the rules defined in Section **??**: (*i*) the timer has elapsed: in this case we return `Resume` which contains the code to execute after the `wait` statement, or (*ii*) the timer has not elapsed: in this case we return `Suspend` which contains the `wait` statement with the updated timer followed by the continuation.

```
<<t <= dt>> == false
-----------------------------------
evalStatement (wait t) k ctxt dt => Suspend wait <<t - dt>>;k ctxt

<<t <= dt>> == true
-----------------------------------
evalStatement (wait t) k ctxt dt => Resume k ctxt
```

The `yield` statement takes as argument a list of expressions whose values are used to update the corresponding fields in the rule domain. The evaluation rule recursively evaluates the expressions and stores them into a list passed as argument of the `Yield` result. Those arguments are used later by `evalRule` to update the corresponding fields.

```
eval expr ctxt => v
evalYield exprs ctxt => vs
-------------------------------------------
     ----------------------------
evalYield (expr :: exprs) ctxt => v :: vs          evalYield nil ctxt
     => nil
```

In this section we provide an implementation of a patrol script for an entity in a game. The sample is made up of an entity, representing a guard, and a couple of checkpoints. The guard continuously moves between the two checkpoints. We choose this sample because this is a typical behaviour implemented in several games, where the user is able to set up a patrol route for a unit. We show the comparison between the sample implemented in Casanova 2.5 and an equivalent implementation in Python with respect to the running time. We then show a comparison between the hard-coded compiler of Casanova 2.0 and the implementation of Casanova 2.5 in Metacasanova with respect to the code length.

4.2.3 Chosen languages

We compared the running time of the sample in metacompiled Casanova with an equivalent implementation in Python. This language was chosen

based on its use in game development: Python has been used extensively in several games such as Civlization IV [16] or World in Conflict [24] because of the native support for coroutines. We deliberately ignore C++ and C# implementations, although they are widely used in the industry, because we knew in advance [5] that the current version of the code generated by the meta-compiler would not match the high performance of these languages: the main goal of this work is to reduce the effort of writing a compiler for a DSL for games while having acceptable performance.

4.2.4 Performance

The performance results are shown in Table 4.1. We see that the generated code has performance on the same order as Python. This is mainly due to the fact that the memory, in the metacompiled implementation of Casanova, is managed through a map, and because of the virtuality of the implemented operators. Each time Casanova accesses a field in an entity this must be looked up into the map. To this we add the complexity of dynamic lookups when we must deal with polymorphic results into the rules.

From Table 4.2 we see that the implementation of Casanova 2.0 language in Metacasanova is almost 5 times shorter in terms of lines of code than the previous Casanova implementation in F#. We believe it is worthy noticing that structures with complex behaviours, such as *wait* or *when*, require hundreds of lines of codes with a standard approach (the code lines to define the behaviour of the structure plus the support code to correctly generate the state machine), while in the meta-compiler we just need tens of lines of codes to implement the same behaviour. Moreover we want to point out that the previous Casanova compiler was written in a functional programming language: these languages tend to be more synthetic than imperative languages, so the difference with the same compiler implemented in languages such as C/C++ might be even greater.

The readability with respect to the hard-coded compiler code is also improved: we managed to implement the behaviour of synchronization and timing primitives almost imitating one to one the formal semantics of the language definition (see the semantics rules in Section **??** and their implementation in Section **??**). In the hard-coded compiler implementation for Casanova 2.0 the semantics are lost in the code for generating finite state machines.

Casanova 2.5		
Entity #	Average update time (ms)	Frame rate
100	0.00349	286.53
250	0.00911	109.77
500	0.01716	58.275
750	0.02597	38.506
1000	0.03527	28.353
Python		
Entity #	Average update time (ms)	Frame rate
100	0.00132	756.37
250	0.00342	292.05
500	0.00678	147.54
750	0.01087	91.988
1000	0.01408	71.002

Table 4.1: Patrol sample evaluation

Casanova 2.5 with Metacasanova	
Module	Code lines
Data structures and function definitions	40
Query Evaluation	16
While loop	4
For loop	5
If-then-else	4
When	4
Wait	6
Yield	10
Additional rules for Casanova program evaluation	40
Additional rules for basic expression evaluation	201
Total: 300	
Casanova 2.0 compiler	
Module	Code lines
While loop	10
For-loop and query evaluation	44
If-Then-Else	15
When	11
Wait	24
Yield	29
Additional structures for rule evaluation	63
Structures for state machine generations	754
Code generation	530
Total: 1480	

Table 4.2: meta-compiler vs standard compiler

4.2.5 Discussion

Metacasanova has been evaluated in [15] by re-building the DSL for game development Casanova [4, 3]. Even though the size of the code required to implement the language has been drastically reduced (almost 1/5 shorter), performance dropped dramatically. We identified a main problem causing the performance decay that, if solved, will improve the performance of the generated code.

In order to encode a symbol table in the meta-compiler in the current implementation (used for example to store the variables defined in the local scope of a control structure or to model a class/record data structure), we are left with two options: (*i*) define a custom data structure made of a list of pairs, containing the field/variable name as a string and its value, in the following way

```
Data "table" -> List[Tuple[string, Value]] : SymbolTable
```

or (*ii*) use a dictionary data structure coming from .NET, such as `ImmutableDictionary`, which was the implementation choice for Casanova. In both cases, the behaviour of the language implemented in Metacasanova will be that of a dynamic language, because whenever the value of a variable or class field must be read, the evaluation rule must look up the symbol table at run time to retrieve the value, whose complexity will be $O(n)$ with the list implementation and $O(\log n)$ with the dictionary implementation. This issue is caused by the fact that, in the current state of Metacasanova, the meta-type system is unaware of the type system of the language that is being implemented in the meta-compiler. This is not a problem limited to Metacasanova but to all meta-compilers having a meta-type system that does not allow embedding of the host language type system. In the next section we propose an extension to Metacasanova to overcome this problem by embedding the type system of the implemented language in the meta-type system

of Metacasanova and inlining the code to access the appropriate variable at compile time.

Chapter 5

Metacasanova optimization

In Section **??** and Section **??** we presented the semantics of Metacasanova and we showed how the meta-compiler generates the code necessary to represent the elements of the language and the evaluation of the rules expressed in terms of operational semantics. In Section 4.2.5 we highlighted the problem of performance degradation, due to the additional abstraction layer of the meta-compiler, and identified a possible cause in how the language manages the memory representation. For now, the memory can only be expressed with a dynamic symbol table that must be looked up at run-time in order to retrieve the value of a variable or of a class/record field. In this section we propose an extension to Metacasanova with parametric *Modules* and *Functors* that will allow to inline the access to record fields at compile time and to embed an arbitrary type system into the meta-type system of Metacasanova. Note that in this scope, we use the term functor with the same meaning used in the scope of the language `CamL`, i.e. a function that takes some types as input and returns a type. In order to provide additional clarity to the explanation, we introduce, in the next section, an example that we use as reference across the whole section.

5.1 Case study

Assume that we want to represent a physical body with a `Position` and a `Velocity` in a 2D space. This can be defined as a data structure containing two fields for its physical properties (the example below is written in F#).

```
type PhysicalBody = {
  Position        : Vector2
  Velocity        : Vector2
}
```

In the current state of the Metacompiler, a language that wants to support such a data structure, as stated in Section **??**, should define it either with a list of pairs $(field, value)$ or with a dictionary from .NET.

```
Data "Record" -> List[Tuple[string, Value]] : Record
```

Accessing the values of the fields requires to iterate through this list (or dictionary) and find the field we want to read, with two evaluation rules such as

```
field = name
-------------
getField ((field,value) :: fields) name
  -> value


field <> name
getField fields name -> v
-------------
getField ((field,value) :: fields) name -> v
```

This could be done immediately by inlining the *getter* (or *setter*) for that field directly in the program.

In what follows we add a system of modules and functors to Metacasanova, we explain how the meta-compiler generates the code for them, and we show how to use them to improve the performance of the example above.

5.2 Using Modules and Functors in Metacasanova

A module definition in Metacasanova is parametric with respect to types, in the sense that a module definition might contain some type parameters, and can be instantiated by passing the specific types to use. A module can contain the definition of data structures, functions, or functors.

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

The symbol `*` reads *kind* and means that the functor might return any type. Indeed the type of a record (or class) in a programming language can be "customized" and depends on its specific definition, thus it is not possible to know it beforehand.

We the define two modules for the *getter* and *setter* of a field of a record. In this example, we use type parameters in the module definitions.

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

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

These two modules respectively define a functor to retrieve the type of the record field, and a function to get or set its value. Note that in the function definitions `get` and `set` we are calling the functor of the `Record` module to generate the appropriate type for the signature. This is allowed, since the result of a functor is indeed a type.

A record meta-type (i.e. its representation at meta-language level) is recursively defined as a sequence of pairs $(field, type)$, whose termination is given by `EmptyField`. We thus define the following functors:

```
Functor "EmptyRecord" : Record
Functor "RecordField" => string => * => Record : Record
```

The first functor defines the end point of a record, which is simply a record without fields. The second functor defines a field as the pair mentioned above followed by other field definitions.

Moreover, we must define two functors that are able to dynamically build the *getter* and *setter* for the field.

```
Functor "GetField" => string => Record : Getter
Functor "SetField" => string => Record : Setter
```

The behaviour of functor is expressed, as for normal functions, through a rule in the meta-program. A rule that evaluates a functor returns an instantiation of a module. Note that, inside a module instantiation, it is possible to define and implement functions other than those in the module definition, i.e. the module instantiation must implement *at least* all the functors and functions of the definition. For instance, the following is the type rule instantiating the module for `EmptyRecord`:

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

  Func "cons" : unit

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

  -------------------
  cons -> ()

}
```

The function `cons` defines a constructor for the record, which, in the case of an empty record, returns nothing. The module instantiation for a record field evaluates as well `RecordType`, and has a different definition and evaluation of the function `cons` (because it is constructed in a different way):

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

  ---------------------------------------
  RecordType => Tuple[type,r.RecordType]

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

Note that the return type of `cons` is to be intended as calling `RecordType` of the current module, so as it were
`this.RecordType`. The getter of a field must be able to lookup the record data structure in search of the field and generate a function to get the value from it. For this reason, the functor instantiates two separate modules, depending on the name of the field that we are currently examining.

```
//Rule 1
name = fieldName
thisRecord := RecordField name type r
-----------------
```

```
GetField fieldName (RecordField name type r) => Getter name thisRecord {
  GetType => type

  ---------------
  get (x,xs) -> x}

//Rule 2
name <> fieldName
thisRecord := RecordField name type r
------------------
GetField fieldName (RecordField name type r) => Getter name type
     thisRecord{
  Functor "GetAnotherField" : Getter

  ---------------
  GetAnotherField => GetField fieldName r

  GetAnotherField => g
  ---------------
  GetType => g.GetType

  GetAnotherField => getter
  getter.get xs -> v
  -------------------
  get (x,xs) -> v }
```

Listing 5.1: Module instantiations for getters

Analogously, the setter of a field instantiates two separate modules whether the current field is the one we want to set or not.

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

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

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

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

  -------------------------
  SetAnotherField => SetField lt r

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

  SetAnotherField => setter
  setter.set xs v -> xs'
  ---------------------------------
  set (x,xs) v -> (x,xs') }
```

Listing 5.2: Module instantiations for setters

5.3 Functor result inlining

If a premise or a conclusion contains a call to a functor, this call is evaluated at compile time, rather than at runtime. Metacasanova has been extended

with an interpreter which is able to evaluate the result of the functor calls. The behaviour of the interpreter follows the same logic explained when presenting the code generation steps in Section ??, thus here we do not present the details for brevity. When a rule outputs the instantiation of the module, the generated code will contain only rules of the modules which conclusion contains a function (i.e. functions that output values, not functors). In this way the generated code will contain a different version of those functions depending on the instantiation parameters of the module.

We now show how to use the implementation of the records given in Section 5.2 for the physical body presented as a case study. The definition of the record type for the physical body is done through a functor

```
Functor "PhysicalBodyType" : Record

EmptyRecord => empty
RecordField "Velocity" Vector2 empty => velocity
RecordField "Position" Vector2 velocity => body
--------------------------
PhysicalBodyType => body
```

This rule is evaluated at compile time by the interpreter that generates one module for each field of the `PhysicalBody`, containing the constructor. For example, for the field `Velocity` the interpreter will generate[1]

```
Func "cons" -> Vector2 -> unit : Tuple[Vector2,unit]

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

This because the functor will call the evaluation rule for `RecordField` with the argument `(Recordfield "Velocity" Vector2 (EmptyRecord))`. This rule generates the function `cons` by evaluating the result of the functors `EmptyRecord.RecordType` and `RecordField.RecordType`, which respectively produce `unit` and `Tuple[Vector2,unit]`.

Instantiating a physical body will just require to build a function that returns the type of the physical body, which is obtained by calling the functor `PhysicalBodyType`.

```
Func "PhysicalBody" : PhysicalBodyType.RecordType

-----------------------
PhysicalBody -> PhysicalBodyType.cons((Vector2.Zero,(Vector2.Zero,())))
```

Defining the setter and getter of a field, requires to use the functor `GetField` to generate the appropriate getter function. After the module has been correctly generated, we can use the getter for the field. For example, in order to get the position field, we use the following function.

```
Func "getPos" -> PhysicalBodyType : Vector2

GetField "Position" PhysicalBodyType => getter
getter.get PhysicalBody -> p
-------------------------------
getPos -> p
```

[1] Note that here we give a high-level representation of the generated rules that are actually directly generated as C# code.

The result of the premise `GetField` will be evaluated at compile time through the code in Listing 5.1 and will instantiate a module containing the following function definition and rule.

```
Func "get" -> Tuple[Vector2,Tuple[Vector2,unit]] : Vector2

-------------------------
get (x,xs) -> x
```

Note that the second premise of `getPos` will immediately call the `get` generated in this step. The case of `setPos` is analogous except the setter takes an additional argument.

Reading `Velocity` analogously uses a functor call to generate a getter:

```
Func "getVel" -> PhysicalBodyType : Vector2

GetField "Velocity" PhysicalBodyType => getter
getter.get PhysicalBody -> p
------------------------------
getVel -> p
```

This time the functor will generate two different functions in two separate modules. The first time the record is processed, `Rule 2` in Listing 5.1 will be activated (because the first field in the Record is `Position`). This rule will instantiate an additional module when evaluating the functor call in its premise, which in turn is able to get the `Velocity` field. The rule for `get` in the first module will contain in its premise a call to `get` of the second module.

```
//Code for module1
Func "get" -> Tuple[Vector2,Tuple[Vector2,unit]] : Vector2

module2.get xs -> v
-------------------------
get (x,xs) -> v

//Code for module2 generated by evaluating the functor in the premise of
      Rule 2
Func "get" -> Tuple[Vector2,unit] : Vector2

------------------
get (x,xs) -> x
```

We want to point out that this optimization has been presented on the specific case of records, but can be generalized for any situations where you would use a symbol table. Indeed any symbol table can be expressed with the representation above as a sequence of pair where the first item is the value of the current variable, and the second item is the continuation of the symbol table.

5.4 Functor interpeter

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

5.5 C-- optimization

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

5.6 Casanova 2.5 optimization

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

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

5.7 Evaluation

An extensive evaluation of Casanova implemented in Metacasanova, which we omit for brevity, can be found in [15]. The implementation of Casanova operational semantics in Metacasanova is almost 5 times shorter than the corresponding F# implementation in the hard-coded compiler. In addition to Casanova, we have implemented a subset of the C language called C--. This language supports `if-then-else`, `while-loop`, and `for` statements, as well as local scoping of variables. The total length of the language definition in Metacasanova is 353 lines of code. The corresponding C# code to implement the operational semantics of the language is 3123 lines, thus the code reduction with Metacasanova is roughly 8.84 times. For comparison, in Table 5.2 it is possible to see the code length to implement three different statements, both in Metacasanova and C#. We tested C-- against Python by computing the average running time to compute the factorial of a number. C-- results to be 50 times slower than Python. This result is worse than what we obtained with Casanova, because in order to emulate the interruptible rule mechanism of Casanova in Python you must rely on coroutines that are slower than a program containing simple statements. Moreover, we tested the performance improvement of the optimization using Functors to represent records against the standard one using dynamic symbol tables. The test was run using records with a number of fields ranging from 1 to 10 and updating from 10000 to 1000000 instances of such records. In Table 5.1, we can see that the optimization using Functors leads to a performance increase on average of about 11 times, with peaks of 30 times. The gain increases with the number of fields, thus Functors are particularly effective for records with high number of fields. Figure 5.1 shows a chart of the overall performance of the two techniques (the data points are taken from Table 5.1). The horizontal axis contains the amount of fields per record, while the vertical axis contains the number of records that are being updated. We can see that the performance of the dynamic table degrades considerably when increasing the number of fields, and that the higher the amount of records is, the steeper the curve is. On the other hand, the performance of the implementation with Functors is almost constant, regardless of the amount of fields or records that are being updated. Moreover, note that the

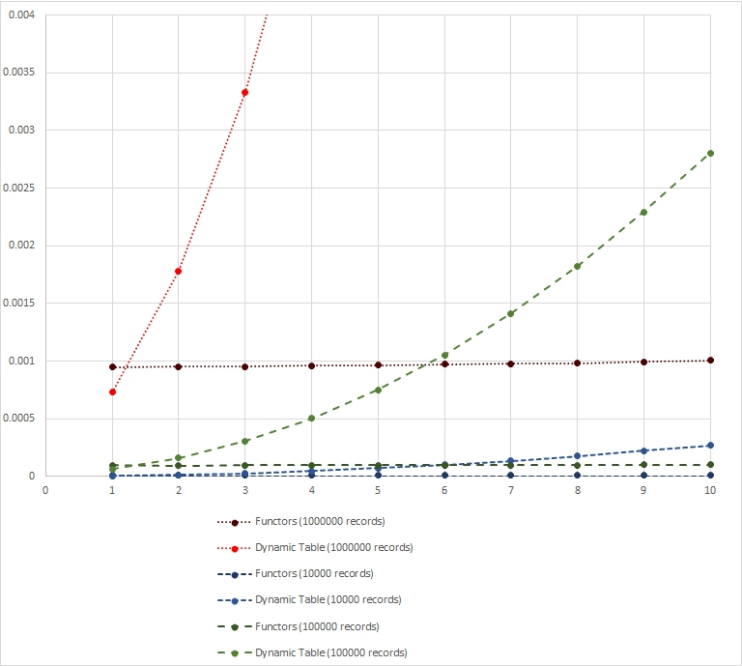


Figure 5.1: Execution time of the different memory models

performance of the dynamic table is improved by the fact that we are using a dictionary implemented in .NET, which can access the entries in $O(\log n)$. If the symbol table were represented as a meta-data structure in the language the performance would be even worse, since it would have to be encoded as a list of pairs with the field name and its value, and its manipulation would be affected by the evaluation rules that should implement this behaviour. Furthermore, the dynamic lookup should be done also to ensure that the types of the record fields are used consistently (for example to prevent that a record is constructed with incompatible values for its fields), while using the functors in Metacasanova embeds the type system of the language in the meta-type system, whose type safety is checked at compile-time rather than at runtime, and this contributes to further increase the performance.

Table 5.1: Running time with the functor optimization and the dynamic table with 10000, 100000, and 1000000 records.

FIELDS	Functors (ms)	Dynamic Table (ms)	Gain
1	1.00E-05	5.00E-06	0.50
2	9.00E-06	1.30E-05	1.44
3	9.00E-06	2.70E-05	3.00
4	9.00E-06	4.50E-05	5.00
5	9.00E-06	7.00E-05	7.78
6	9.00E-06	9.90E-05	11.00
7	9.00E-06	1.33E-04	14.78
8	9.00E-06	1.75E-04	19.44
9	9.00E-06	2.20E-04	24.44
10	9.00E-06	2.70E-04	30.00
		Average gain	11.74

FIELDS	Functors (ms)	Dynamic Table (ms)	Gain
1	9.60E-05	6.30E-05	0.66
2	9.40E-05	1.59E-04	1.69
3	9.50E-05	3.04E-04	3.20
4	9.60E-05	5.03E-04	5.24
5	9.60E-05	7.52E-04	7.83
6	9.60E-05	1.05E-03	10.95
7	9.70E-05	1.41E-03	14.57
8	9.80E-05	1.82E-03	18.59
9	9.90E-05	2.29E-03	23.17
10	1.00E-04	2.81E-03	28.05
		Average gain	11.39

FIELDS	Functors (ms)	Dynamic Table (ms)	Gain
1	9.47E-04	7.29E-04	0.77
2	9.51E-04	1.78E-03	1.87
3	9.50E-04	3.33E-03	3.51
4	9.60E-04	5.43E-03	5.66
5	9.65E-04	8.03E-03	8.32
6	9.71E-04	1.11E-02	11.44
7	9.75E-04	1.47E-02	15.12
8	9.82E-04	1.89E-02	19.28
9	9.92E-04	2.37E-02	23.86
10	1.00E-03	2.87E-02	28.62
		Average gain	11.84

Table 5.2: Code length implementation of C-- and run-time performance

Statement	Metacasanova	C#
`if-then-else`	4	103
`while`	7	73
`For`	11	81

C--	Python
1.26ms	$2.36 \cdot 10^{-2}$ms

Chapter 6

Networking primitives in Casanova 2

In this section we introduce the basic concepts of the implementation of multiplayer game development for Casanova 2. This implementation aims to relieve the programmer of the complexity of hard-coding the network implementation for an online game, while preserving encapsulation in code. We show that code analysis is required to generate the appropriate network primitives to send and receive data. Finally, we present a simple multiplayer game to show a concrete example.

6.1 Introduction

Adding multi-player support to games is a highly desirable feature. By letting players interact with each other, new forms of gameplay, cooperation, and competition emerge without requiring any additional design of game mechanics [20]. This allows a game to remain fresh and playable, even after the single player content has been exhausted. For example, consider any modern AAA (AAA refers to games with the highest development budgets[33]) game such as *Halo 4*. After months since its initial release, most players have exhausted the single player, narrative-driven campaign. Nevertheless the game remains heavily in use thanks to multiplayer modes, which in effect extended the life of the game significantly. This phenomenon is even more evident in games such as *World of Warcraft* or *EVE*, where multiplayer is the only modality of play and there is no single-player experience.

Challenges Multi-player support in games is a very expensive piece of software to build. Multiplayer games are under strong pressure to have very good *performance* [14]. Performance is both expressed in terms of CPU time and in bandwidth used. Also, games need to be very *robust* with respect to transmission delays, packets lost, or even clients disconnected. To make matters worse, players often behave erratically. It is widespread practice among players to leave a competitive game as soon as their defeat is apparent (a phenomenon so common to even have its own name: “rage

quitting" [22]), or to try to abuse the game and its technical flaws to gain advantages or to disrupt the experience of others.

Networking code reuse is quite low across titles and projects. This comes from the fact that the requirements of every game vary significantly: from turn-based games that only need to synchronize the game world every few seconds, and where latency is not a big issue, to first-person-shooter games where prediction mechanisms are needed to ensure the smooth movement of synchronized entities, to real-time strategy games where thousands of units on the screen all need to be synchronized across game instances [29]. In short, previous effort is substantially inaccessible for new titles.

Encapsulation suffers from this ad-hoc nature of the implementation of the networking layer in multiplayer games. Indeed managing the information about game updates over a network requires each game entity to interface the game logic code with network connection and socket objects, data transmission method calls such as send and receive, and support data structures to manage traffic and track the status of common protocols. This happens because each game entity must provide the following functionality in order to work in a multiplayer game:

- Update the logic in the fashion of a singleplayer counterpart.
- Choose what data is necessary to send over the network and create the message containing this information.
- Choose what data can be lost and what data must always be received by the other clients.
- Periodically check if incoming messages contain information that needs to be read and to perform specific updates.

Combining these requirements together within the same entity breaks encapsulation because now the logic of the entity and lots of spurious details only relevant to the networking implementation are mixed together, resulting in a highly noisy program. Maintenance then becomes very hard, as every change in the game logic must also be reflected in the networking implementation.

Existing approaches Networking in games is usually built with either very low-level or very high-level mechanisms. Very low-level mechanisms are based on manually sending streams of bytes and serializing only the essential bits of the game world, usually incrementally, on unreliable channels (UDP). This coding process is highly expensive because building by hand such a low-level protocol is difficult to get right, and debugging subtle protocol mismatches, transmission errors, etc. will take lots of development resources. Low-level mechanisms must also be very robust, making the task even harder.

High-level protocols such as RDP, reflection-based serialization, frameworks (such as Pastry, netty.io), etc. can also be used. These methods greatly simplify networking code, but are rarely used in complex games and scenarios. The requirements of performance mean that many high-level protocols or mechanisms are insufficient, either because they are too slow computationally (especially when they rely on reflection or events) or because they transmit too much data across the network.

6.2 Motivation

To avoid the problems of both existing approaches, we propose a middle ground. We observe that networking fundamental abstractions upon which the actual code and protocols are built do not vary substantially between games, even though the code that needs to be written to implement them does. The similarity comes from the fact that the ways to serialize, synchronize, and predict the behaviour of entities are relatively standard and described according to a limited series of general ideas. The difference, on the other hand, comes from the fact that low-level protocols need to be adapted to the specific structure of the game world and the data structures that make it up. Until now, common primitives have not been syntactically and semantically captured inside existing domain-specific languages for game development [10]. Using the right level of abstraction, these general patterns of networking can be captured, while leaving full customization power in the hand of the developer (to apply such primitives to any kind of game).

6.3 Related work

In the following we discuss some existing networking tools used in game development and we highlight some issues that arise from their use.

The Real time framework (RTF) RTF [19] is a middleware built for C++ to relieve the programmer from dealing with data compression. It is more flexible than solutions based on game engines or hand-made implementations, since it automates the process of data transmission. Moreover, it supports distributed server management. Unfortunately, this solution has several flaws:

- All entities must inherit from the class `Local` and the semantics of the position is pre-determined, often clashing with rendering or physics.
- Platform independence requires that the programmer uses RTF primitive types.
- Data transmission automation requires that all game entities inherit the class `Serializable`.
- Being a middleware, RTF is not aware of what games are going to use it for (every game comes with different data structures). Thus, the developer is tasked to include in his code also logic to update the RTF layer, in order to keep the game updated over the network.

Network scripting language (NSL) NSL [27] provides a language extension based on a send-receive mechanism. Moreover it provides a built-in client side prediction (a feature missing in existing highly concurrent and distributed languages such as Stackless Python [30] and Erlang [8]), which is periodically corrected by the server.

Unreal Engine/Unity Engine Unreal Engine [2] and Unity Engine [1] are commercial game engines supporting networking. Both Unity and Unreal Engine use a client-server approach. In Unreal Engine, the server contains the "true" game state, and the clients contain a "dirty" copy, which is validated periodically. It is possible to define entities (actors in Unreal Engine jargon) that are replicated on the clients. Whenever a replicated actor changes on the server, this change is also reflected on the clients. Additional customization can be achieved through Remote procedure calls (RPCs) of three kinds.

- The function is called on the server and executed on the client. This is used for game elements that do not affect gameplay, such as creating a particle effect when a weapon is fired.

- The function is called on the client and executed on the server. This is useful for events that affect the other clients and should be validated by the server.

- The function is executed in multi-cast, meaning that the server calls the function and that it is executed on both the server and all the clients.

The Unity Engine uses a similar approach based on networking components, synchronized at every frame, and RPC's to define custom synchronization events.

Unfortunately, customization comes at the cost of the level of detail that developers must face. Using RPC's require a deep knowledge of the engine and writing lots of code, as discussed in Section **??**.

In this section we introduce a small example that addresses the requirements of designing a multiplayer game. We then present an architecture that aims to fulfil these requirements.

6.4 The master/slave network architecture

We chose to implement the networking layer in Casanova 2 by using a peer-to-peer architecture for the following reasons:

- Server-client architectures are more reliable but suitable only for specific genres of games (mostly Shooter games), while other genres, such as Real-time strategy games or Online Role Playing Games, use P2P architectures.

- We do not have to write a separate logic for an authoritative game server, which has to validate the actions of clients.

Casanova will provide a generic tracking server, which is run separately from the main program. The tracking server is a thin service that connects players participating in a single game, and helps with forwarding the network traffic through NATs (Network Address Translation).

Each client maintains a local copy of the `world` entity and has direct control over a single portion of it. Instances belonging to such as portion are seen as *master* by this player, who is always allowed to directly change

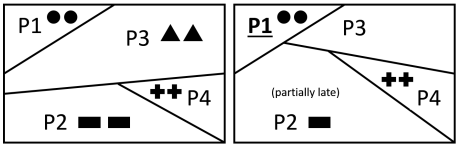
the state of the master instances without having to validate this state change by synchronizing with other players through the network.

Each client also maintains a portion of the world that is not directly under his control. Instances belonging to such as portion are seen as *slave* by this player, who is only allowed to *predict* the local state of the instances and, whenever he receives an update from their masters, must correct this prediction according to the data contained in the received messages. The slave part of the world is thus maintained passively by the client: the only active part is predicting the evolution of the entity state and correcting it whenever he receives an update by its master.

For this purpose, we extend the syntax of Casanova rules by allowing them to be marked with the modifiers `master` and `slave`. These rules are executed respectively on master and slave entities. Note that it is still possible not to mark a rule with these modifiers, which means that the rule is always executed independently of the fact that the entity is either master or slave on that particular client. We also allow to mark a rule as `connecting` and `connected`. These rules are triggered only once respectively when a new client connects and when the clients detect a new connection.

Casanova also provides primitives to send (reliably or unreliably) and receive data. A schematic representation of this architecture can be seen in Figure 6.2.

Figure 6.1: Representation of the game world in a networking scenario



(a) Unknown correct game state when P3 joins the game.

(b) Networking game state seen from the point of view of P1. P2 is partially synchronized, P4 is fully synchronized, and P3 is a new client that is late and is still sending its data

Note the aim of this architecture is to provide language-level primitives to describe the networking logic. This means that the compiler will be able to generate code compatible with the low-level network libraries that provide transmission functions over the network channel without having to change Casanova code in the program. In our implementation, we chose the .NET library `Lidgren`, which is widely used also in commercial game engines such as Unity3D and MonoGame, but nothing prevents the compiler to be expanded in order to target other similar libraries for other languages, such as jgroups [9].

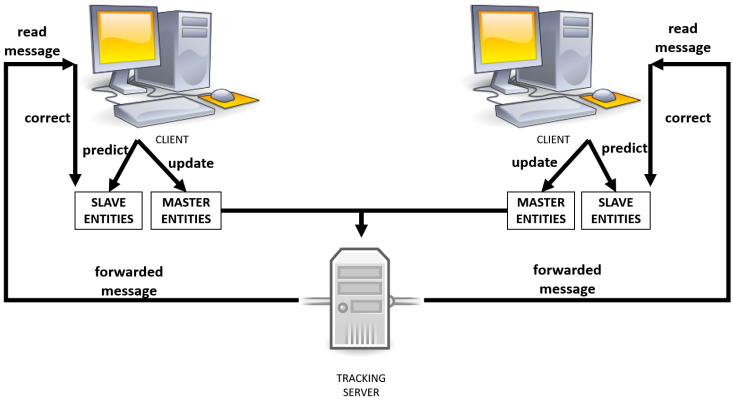


Figure 6.2: master/slave architecture

6.5 Case study

Let us consider a simple shooter game where each player controls a space ship. Players can move forward, backward, and rotate the ship to change direction. Moreover, they can use the ship lasers to shoot other players. If a laser hits an enemy ship, we increase the player's score. Designing such a game requires to address the following issues, depicted by the schematic representation in Figure 6.1:

1. Each player must maintain a local version of the game state (world). In order to avoid to flood the network with messages, all the copies are not fully synchronized at each frame, thus they are slightly different and each client knows the latest version of only part of the copy.

2. A player `connecting` to an existing game must be able to receive the latest update of the game state and send the new ship he will control to existing players in the game.

3. A player already `connected` to the game must detect a new connection and send his master portion of the game state.

4. Each player must be able to control only one ship at a time. This means that the part of the game logic that processes the input and modifies the spatial data of the ship (position and rotation) should only be executed on the ship controlled by the player and not on the local copies of other players' ships. This means that each player sees as `master` only one ship instance.

5. Each player must send the updated state of the ship he controls to the other players after executing the local update. To achieve better performance over the network, the data is not sent at every update, but with a lower frequency.

6. Each player must receive the updated state of `slave` ships controlled by other players. In this phase, we must take into account that, as explained above, not every update is sent, so the player should "predict" what will happen during the game frames in which he does not receive an update.

6.6 Implementation

Each of the scenarios described above requires specific language extensions. These extensions identify connection, ownership (master/slave), and various send and receive primitives. In this section, we introduce each primitive by using a multiplayer game example [1]. We now give an implementation of the shooter game presented above, using the extended version of Casanova 2 with network primitives.

The `world` contains a list of ships controlled by each player.

```
world Shooter = {
Ships  : [Ship]
...
}
```

Each `Ship` contains a position, a rotation, a collection of shot projectiles, and the score.

```
entity Ship = {
Position   : Vector2
Rotation   : float32
Projectiles : [Projectile]
Score      : int
...
}
```

Each `Projectile` contains its position and velocity.

```
entity Projectile = {
Position : Vector2
Velocity : Vector2
...
}
```

Connection

When a player connects, we must consider two different situations: (*i*) a player is already in the game and must send the current game state to the connecting players, and (*ii*) the player who is connecting needs to send the ship he will instantiate and control (its initial state). Both the players in the game and the connecting one must receive the game states that are sent. For this purpose we introduce two additional modifiers, `connecting` and `connected`, that can be added to rule declarations to mark their role in the multiplayer logic.

[1] The game source code and executable can be found at `https://github.com/vs-team/casanova-mk2/wiki/Networking-extension`

Connecting A rule marked with `connecting` is executed once when a player joins the game for the first time. In our example, the player should send his initial state (the created ship) to the other players. We use the primitive `send_reliable` because we must be sure that eventually all players will be notified of the ship creation.

```
world Shooter = {
...
rule connecting Ships =
yield send_reliable Ships
}
```

Connected A rule marked with `connected` is run whenever a new player joins the game by all existing players. When this occurs, each player sends its ship. The system will take care to send only the ship controlled locally by the player itself for each player. The rule will use the `send_reliable` primitive for the same reason explained in the previous point.

```
world Shooter = {
...
rule connected Ships =
yield send_reliable Ships
}
```

Note that even if the code is the same, the semantics of the two rules are different. The first one is executed by the player joining the game, who locally instantiates its `Ship` and must send its list of `Ships` (containing only the local instance) to the other players. The second one is executed by all existing players who must share with the joining player the list of existing ships.

Master updates

As explained above, each client manages a series of local game objects (called *master objects*) that are under its direct control. The other clients read passively any update done on those instances and update their remote copy (*slave objects*) accordingly. We mark rules affecting the behaviour of master objects as `master`. In our example, the following situations are run as master: (*i*) synchronizing the ships among players, (*ii*) updating the ship and projectiles spatial data, and (*iii*) creating and destroying projectiles.

1. Each player is tasked to maintain the list of Ships in the world. This requires to receive the updated list from other players and to store the new value in a master rule. Indeed the world is a special case of an entity that is shared among players, and not directly owned by somebody. Each ship contained in that list and received from other players will be treated appropriately as slaves, while the only one owned by the current player will be under his direct control. In this rule we use `let!`, which is an operator that waits until the argument expression returns a result and then binds it to the variable. The symbol `@` stands for list concatenation. The rule uses `receive_many`, which receives and collects the list of sent ships by the other players.

```
world Shooter = {
...
rule master Ships =
let! ships = receive_many()
yield Ships @ ships
}
```

2. The master version of the ship update reads the input of the player and moves (or rotates) the ship if the appropriate key is pressed. Note that this part must be executed only on a master object, because we want to allow each player to control only the ship he owns and instantiates at the beginning of the game. Below we show just the rule to move forward; the other movement and rotation rules are analogous. We use an *unreliable send* because it is acceptable to lose an update of the position during a certain frame: shortly after, there will be a new update.

```
entity Ship = {
...
rule master Position =
wait world.Input.IsKeyDown(Keys.W)
let vp = new Vector2(Math.Cos(Rotation),
Math.Sin(Rotation)) * 300.0f
let p = Position + vp * dt
yield send p
}
```

   We do the same for projectiles, except the projectile position is continuously updated and synchronized over the network without having to wait that a key is pressed.

3. Creating a new projectile happens when the player shoots. A ship keeps track of the projectiles it has shot so far, and adds a new one to the list of the existing projectiles. The updated list is sent to all players with the new instance of the projectile (which is added as a new head of the list with the operator `::`). Here it is better to precise the semantics of the `yield` in conjunction with the use of networking primitives. A `yield` requires that the written value is type-compatible with the domain of the rule. Thus, when used with a `send` primitive, we must pass as argument a list. The system will ensure, for performance reasons, that the generated code only sends the new items added to the list. This semantics is defined as such for two main reasons: (*i*) when sending the new projectiles we must also update the list in local (and given the immutability of Casanova we must replace the existing one), and (*ii*) because in this way the programmer can focus on the logic of the game as if it were a single-player game without worrying of network-specific details. Note that the last `wait` forces the player to release the key before shooting again (semi-automatic fire). Removing that check would spawn multiple projectiles consecutively, which is not a wanted behaviour.

```
entity Ship = {
...
rule master Projectiles =
```

```
wait world.Input.IsKeyDown(Keys.Space)
let vp = new Vector2(Math.Cos(Rotation),
Math.Sin(Rotation)) * 500.0f
let projs = new Projectile(Position, vp) :: Projectiles
yield send_reliable projs
wait not world.Input.IsKeyDown(Keys.Space)
}
```

Filtering the colliding projectiles and updating the score is run as a master rule. The rule computes the set difference between the ship projectiles and the colliding projectiles and updates the list of projectiles, sending them through the network as well. Even in this case, the network layer sends only the information about the projectiles to remove. Note that the score is managed by each player locally, as it does not require to be synchronized (we do not print the other players' scores. Doing so would indeed require to also send the score).

```
entity Ship = {
...
rule master Projectiles, Score =
let collidingProjs =
[for p in Projectiles do
let ships =
[for s in Ships do
where
s <> this and
Vector2.Distance(p.Position,s.Position) < 100.0f
select s]
where ships.Count > 0
select p]
let newProjectiles = Projectiles - collidingProjs
yield send_reliable newProjectiles,
Score + collidingProjs.Count
}
```

Managing remote instances

The game objects that were not instantiated by a client, but received from another client, are *slave objects* and must be synchronized differently than master objects. For this purpose, a rule can be marked as `slave`. In our example, we use slave rules in the following situations: (*i*) synchronizing other players' ships and projectiles spatial data, and (*ii*) projectiles instantiated by other players.

1. Every remote projectile and ship is synchronized locally by a rule, which tries to `receive` a message containing updated spatial data. Below we provide the code to update the position of the ship; the synchronization of other spatial data is analogous.

```
entity Ship = {
...
rule slave Position = yield receive()
}
```

2. When a projectile is instantiated remotely, we have to receive it and add it to the list of projectiles. We use `receive_many` because the new projectiles are added to a list. This case also supports the situation where a ship could shoot multiple projectiles at the same time.

```
entity Ship = {
...
rule slave Projectiles =
let! projs = receive_many()
yield projs @ Projectiles
}
```

In this scenario is important to discuss the atomicity of these transmissions: in the context of network games, reliability is often sacrificed for better network performance, so most of the data transmissions are unreliable (like in the case of the ship position). This means that we have no guarantee that the message will be received. Several issues can arise from this situation: for example, if a player fails to receive the position of the ship, then it might miss a collision with a projectile. This is a well-known issue in several shooter games and out-of-sync errors might happen during a multiplayer game. However, ensuring that all the data transmissions are reliable might affect network performance to the point that the game would become unplayable because of the network overload.

Casanova 2 allows the programmer to decide whether the transmission should be reliable or not and experiment with the effect of a reliable transmission versus an unreliable one that does not overload the network. For example, the updated list of projectiles, after a collision, is always sent in a reliable way. This is acceptable because collisions are not so frequent. This is not true for the ship position, since movements are very frequent and mostly happen at every frame, thus it is something that should not be sent reliably at every frame.

Furthermore, we want to focus the attention on the implicit relationship between this networking architecture and the encapsulation: as shown for instance in the examples where the ship shoots a projectile, we ensure encapsulation by keeping a semantics that filters completely the details about networking. The programmer only worries about the logic of adding a new projectile, while the details of the network transmission are hidden. A handmade implementation is usually prone to break this separation of concerns because the transmission logic is tightly coupled within the game logic itself.

6.7 Networking in Metacasanova

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.

Bibliography

[1] Unity Game Engine. `http://unity3d.com`.

[2] Unreal Technology. `https://www.unrealengine.com/`.

[3] Mohamed Abbadi. *Casanova 2, A domain specific language for general game development.* PhD thesis, Università Ca' Foscari, Tilburg University, 2017.

[4] Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi, Pieter Spronck, Giulia Costantini, and Giuseppe Maggiore. Casanova: a simple, high-performance language for game development. In *Joint International Conference on Serious Games*, pages 123–134. Springer, 2015.

[5] Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi, Pieter Spronck, Giulia Costantini, and Giuseppe Maggiore. Casanova: A simple, high-performance language for game development. In Stefan Göbel, Minhua Ma, Jannicke Baalsrud Hauge, Manuel Fradinho Oliveira, Josef Wiemeyer, and Viktor Wendel, editors, *Serious Games*, volume 9090 of *Lecture Notes in Computer Science*, pages 123–134. Springer International Publishing, 2015.

[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[7] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[8] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG.* Prentice Hall, 1993.

[9] B. Ban. JGroups - A Toolkit for Reliable Multicast Communication. `http://www.jgroups.org/index.html`, 2002.

[10] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (DSLs) for network protocols. In *International Workshop on Next Generation Network Architecture (NGNA 2009)*, 2009.

[11] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1):52–70, 2008.

[12] WR Campbell. A compiler definition facility based on the syntactic macro. *The Computer Journal*, 21(1):35–41, 1978.

[13] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[14] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[15] Francesco Di Giacomo, Mohamed Abbadi, Agostino Cortesi, Pieter Spronck, and Giuseppe" Maggiore. *Intelligent Technologies for Interactive Entertainment: 8th International Conference, INTETAIN 2016, Utrecht, The Netherlands, June 28–30, 2016, Revised Selected Papers*, chapter Building Game Scripting DSL's with the Metacasanova Metacompiler, pages 231–242. Springer International Publishing, Cham, 2017.

[16] Firaxis Games. Civlization iv scripting api reference. `http://wiki.massgate.net/Our_Python_files_and_Event_Structure`, October 2008.

[17] Plotkin G.D. A structural approach to operational semantics. Technical report, Computer science department, Aarhus University, 1981.

[18] Francesco Di Giacomo. Casanova 2.5 source code. `https://github.com/vs-team/metacompiler/tree/master/Sources/Content/Content/CNV3`, 2016.

[19] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch. RTF: a real-time framework for developing scalable multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 81–86. ACM, 2007.

[20] C. Granberg. *David Perry on game design: a brainstorming toolbox*. Cengage Learning, 2014.

[21] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(4):437–444, 1998.

[22] E. Kaiser and W. Feng. PlayerRating: a reputation system for multiplayer online games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, page 8. IEEE Press, 2009.

[23] Jan Willem Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.

[24] Massive Entertainment. World in conflict script reference. `http://civ4bug.sourceforge.net/PythonAPI/`, September 2007.

[25] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE transactions on Electronic Computers*, (1):39–47, 1960.

[26] Mikael Pettersson. A compiler for natural semantics. In *Compiler Construction*, pages 177–191. Springer, 1996.

[27] G. Russell, A.F. Donaldson, and P. Sheppard. Tackling online game development problems with a novel network scripting language. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 85–90. ACM, 2008.

[28] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[29] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20(2):87–97, 2002.

[30] C. Tismer. Stackless Python. `http://www.stackless.com/`.

[31] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[32] Daniel Weise and Roger Crew. Programmable syntax macros. In *ACM SIGPLAN Notices*, volume 28, pages 156–165. ACM, 1993.

[33] M.J.P. Wolf. *The video game explosion: a history from PONG to Playstation and beyond*. ABC-CLIO, 2008.