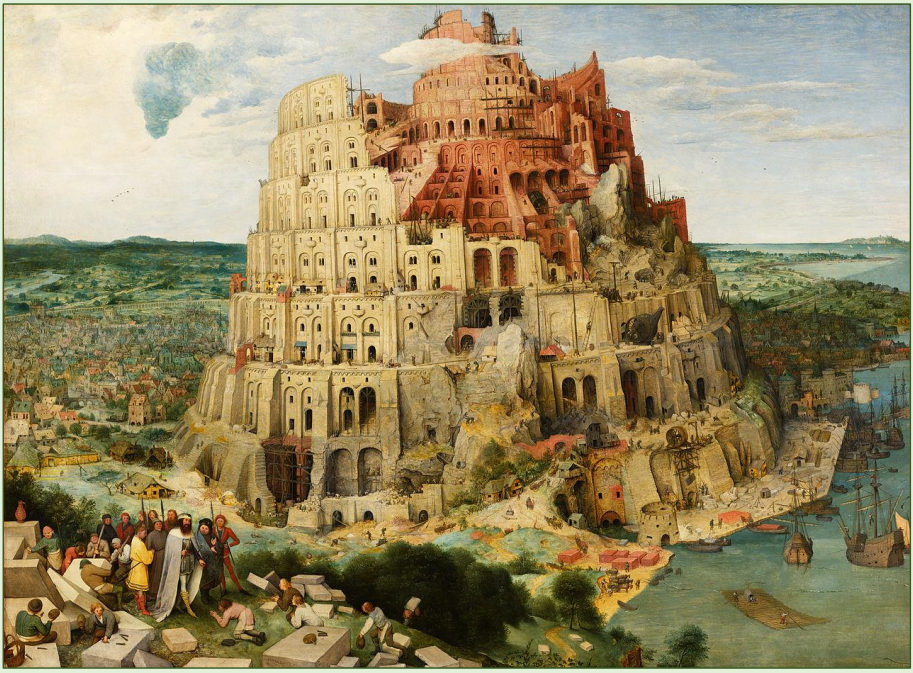
Metacasanova

A High-performance Meta-compiler for Domain-specific Languages



Francesco Di Giacomo

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 of this research 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.

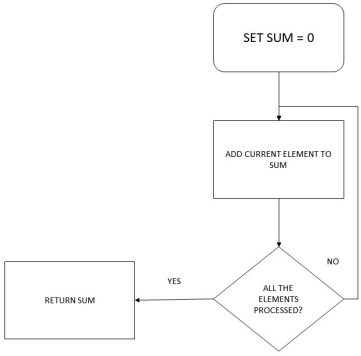


Figure 1.1: Flow chart for the sum of a sequence of numbers

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 reason, 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*. A rectangle with rounded corners denotes the initial step. 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 allows to employ several abstractions and even define some steps in terms of natural language. The drawback of these two formal representations 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.

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 two 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 [...]:

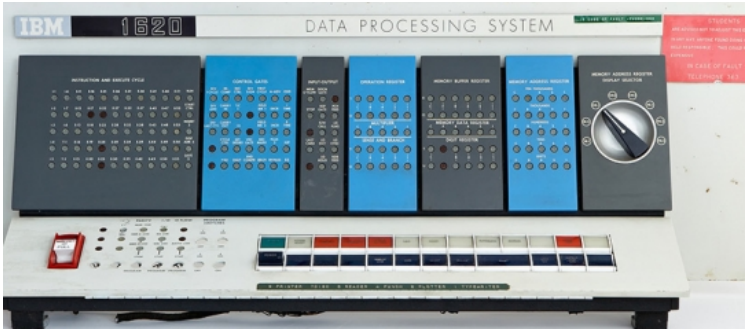


Figure 1.2: Front panel of IBM 1620

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, (*iii*) the logic and readability of the program is hidden among the details of the instruction set itself, and (*iv*) developing a program in assembly requires a considerable effort in terms of time and debugging [...]: assembly lacks any abstraction from the concrete hardware architecture, such as a type system, that partially ensures the correctness of the program or high-level constructs that allow to manipulate the execution of the program.

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 }
```

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. Declarative programming languages are called as such when they are side-effects free or referentially transparent. The definition of referential transparency varies [54], but it is usually explained with the substitution principle, which states that a language is referentially transparent if any expression can be replaced by its value without altering the behaviour of the program [47]. For instance, the following sentences in natural language are both true

```
Cicero = Tullius

''Cicero'' contains six letters
```

but they are not referentially transparent, since replacing the last name with the middle name falsifies the second sentence.

A similar situation in programming languages is met when considering variable assignments: the statement

```
x = x + 5
```

is not referentially transparent. Let us assume this statement appears twice in a program and that at the beginning x = 0. Clearly the expression `x + 5` results in the value 5 the first time, but the second time the same statement is executed the expression has value 10. Thus replacing all the occurrences of `x + 5` with 5 is wrong, which is why imperative languages are not referentially transparent. A more rigorous definition of referential transparency can be found in [58].

Declarative programming languages 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# with a recursive

function. Higher-order functions, such as `fold`,allow even to capture the same recursive pattern into a single function as shown in Listing 1.5. Both implementations are referentially transparent.

```
let rec sumList l =
  match l with
  | [] -> 0
  | x :: xs -> x + (sumList xs)
```

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

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

Listing 1.5: F# code to compute the sum of a sequence of numbers using higher-order functions

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 of 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 1.6 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 1.6: 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 constructs to model timing event-based primitives, or to execute tasks in parallel. These behaviours cannot be modelled, for performance reasons, by using threads. Therefore, in the past, domain-specific languages which provide these abstractions have been implemented. [...]. In Listing 1.7 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 1.7: 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 1.8 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.

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

Listing 1.8: 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 does not need to 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 languages 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 in 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 conforms 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 regardless 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, regardless 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; indeed if a meta-program allows to mimic directly the type system and semantics of the language their correctness is transferred also in the implementation, while this might not be trivial when translating them in the abstractions of a high-level programming language; (*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. This process tends to be slow and often significant improvements or additions are only seen years after the first 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 change 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 compilers, 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 to the development of a high variety of domain-specific languages. Indeed, such 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 aspects that are strictly tied 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, things 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 implementations have been proposed [...]. In general meta-compilers perform poorly compared to hard-coded compilers because they add the additional layer of abstraction of the meta-language. Moreover, a specific implementation of a compiler opens up 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: *To what extent does the use of a meta-compiler ease the development of a domain-specific language?*

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 and is this loss acceptable?*

In case of a performance loss, we need to identify the cause of this performance loss and if an improvement is possible. This leads to the following research question:

Research question 3: *What is the cause of the performance degradation when employing a meta-compiler and how can this be improved?*

1.6 Thesis structure

This thesis describes the architecture of Metacasanova, a meta-compiler whose meta-language is based on operational semantics, and a possible optimization for such meta-compiler. It also shows its the capabilities by implementing a small imperative language and re-implementing the existing domain-specific language for games *Casanova 2*, extending it with abstractions to express network operations for multiplayer games.

In Chapter 2 we provide background information in order to understand the choices made for this work. The chapter presents the state of the art in designing and implementing compilers and existing research on meta-compilers.

In Chapter 3 we present the architecture of Metacasanova by extensively describing the implementation of all its modules.

In Chapter 4 we show how to use Metacasanova to implement two languages: a small imperative language, and *Casanova 2*, a language for game development. At the end of the chapter we provide an evaluation of the performance of the two languages and their implementation length with respect to existing compilers, thus answering to Research Question 1 and 2.

In Chapter 5 we discuss the performance loss of the implementation of the presented languages and we propose an extension of Metacasanova that aims to improve the performance of the generated code, thus answering Research Question 3.

In Chapter 6 we propose an extension of Casanova 2 for multiplayer game development by providing both a hard-coded compiler solution and a meta-compiler one. We then compare the two approaches performance-wise and code-length-wise.

In Chapter 7 we discuss the result and answer the research questions.

Chapter 2

Background

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

This chapter provides background information on compiler construction and the existing knowledge on meta-compiles. The goal of this chapter is dual: (*i*) it provides the reader with sufficient information to understand the implementation choices done when developing Metacasanova, and (*ii*) outlines the complexity of the process of designing and implementing a compiler, thus giving further motivation to this research work.

In Section 2.1 we outline the general architecture of a compiler by giving a short descriptions of all its components and how they work. In Section 2.2 we give a detailed explanation about *regular expressions* necessary to define the "words" of a language, and the *lexer*, showing how to implement one. In Section 2.3 we introduce the notion of *context-free grammars* and we show how to implement a parser able to process the grammatical rules of such grammar. In this chapter we present a parser generator for the language F# that has been used for the implementation of Metacasanova, and then show an alternative to standard parsers in functional programming languages. We then explain how a type system and semantics of a language is expressed, and finally we introduce the concept of metaprogramming and we show examples using metaprogramming in the abstractions provided by a general purpose language (C++), and with existing dedicated metacompilers.

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 manda-

tory. A special kind of compilers are interpreters, 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 [9]:

- *Empty:* The regular expression ϵ is a language containing only the empty string.
- *Symbol:* $\forall a \in \Sigma$, **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 [42], 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 $\mathbf{M}^+$ 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 [8]. Another possible approach is an algorithm to directly convert regular expressions into DFA's, as shown in [7]. 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 Closure of S

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

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.

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 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" [8, 9]. For example, trying to define an arithmetic expression with chained sums would lead to the following (recursive) regular expression:

```
expr = "(" expr "+" expr ")" | digits
```

Now we would need to replace the regular expression with itself, thus obtaining

```
expr = "(" "(" expr "+" expr ")" | digits "+" "(" expr "+" expr ")" |
     digits ")" | digits
```

Algorithm 2.2 NFA into DFA conversion

```
states[0] ← ∅
states[1] ← C(s_1)
p ← 1
j ← 0
while j ≤ p do
    for all c ∈ Σ do
        e ← E(states[j], c)
        if ∃ i ≤ p | e = states[i] then
            trans[j, c] ← i
        else
            p ← p + 1
            states[p] ← e
            trans[j, c] ← p
        end if
    end for
    j ← j + 1
end while
```

It is easy to see that this substitution would never end, as the regular expression keeps growing at each replacement.

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 expressions, thus context-free grammars are more expressive than 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* (alse called LL(k) parsers), 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

$$S \to E\ eof$$
$$E \to E + T$$
$$E \to E - T$$
$$E \to T * F$$
$$E \to T/F$$
$$E \to T$$
$$T \to F$$
$$F \to id$$
$$F \to num$$
$$F \to (E)$$

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)
  $J \leftarrow \emptyset$
  for all $N \to \alpha.X\beta \ \in I$ **do**
    $J \leftarrow J \cup \{N \to \alpha X.\beta\}$
  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

The process of creating a parser can be automated by using a *Parser Generator*. A Parser generator is a programming language that accepts the definition of the grammar of a language and generates a parser (and a lexer) for it. As programming languages generally have a LALR(1) grammar [9], most of parser generators produce a LALR(1) parser. Since in this research work we used F# as a development language, in this section we present the F# lexer and parser generators, belonging to the Yacc generator family, known ans *FsLex* and *FsYacc*.

Definition of a lexer in FsLex

FsLex allows to define the tokens with the regular expression syntax. Each FsLex program begins with a header, where the programmer can specify auxiliary modules and functions to use in the lexer. After the header, it is possible to specify relevant regular expressions that are used by the lexer to analyse the tokens with the standard let-binding syntax of F# [43]. The right argument of this binding is a regular expression, which can be composed with the combinators for regular expressions seen in Section 2.2.1. For example the following regular expression can define the syntax for variable names in a programming language:

```
let simpleId = ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '_' '0'-'9']+
```

Regular expression bindings can be used as alias in the lexer definition. A lexer definition is identified by the keyword `rule` for the binding. The right side of a lexer definition contains a call to the function `parse`, which tries to execute one of the rules specified below to parse a token. Each lexer rule generates a result which is a token data structure. Token data structures are specified at parser level (see below). For instance, the following is a lexer able to recognize comparison operators:

```
rule comparisonOperators = parse
| "=" { Parser.EQUAL }
| ">" { Parser.GT }
| ">=" { Parser.GEQ }
| "<" { Parser.LT }
| "<=" { Parser.LEQ }
| "<>" { Parser.NEQ }
```

Note that, in order to provide useful information about the lexing phase, we might need to access, for instance, the position of the lexer (for error reporting), or to get the string read by the lexer (for example to generate literals when reading numbers or strings). This information is provided by the *lexer buffer*, which is a data structure generated automatically by the parser generator. For example, if the token needs to store its row and column position in the file for error reporting, the definition above can be changed in this way (note the use of the header to define the function `range`):

```
{
  module Lexer

  let range (lexbuf : LexBuffer<_>) = lexbuf.EndPos.Line + 1, lexbuf.
      EndPos.Column
}
```

```
rule comparisonOperators = parse
| ">" { Parser.GT (range lexbuf) }
| ">=" { Parser.GEQ (range lexbuf) }
| "<" { Parser.LT (range lexbuf) }
| "<=" { Parser.LEQ (range lexbuf) }
| "<>" { Parser.NEQ (range lexbuf) }
```

Another useful feature of FsLex is the capability of defining recursive lexers. Let us consider the case of skipping multi-line comments: usually such comments are delimited by a start and end symbol, and the comments spread across multiple lines. For example in `C++/Java/C#` a multi-line comment is delimited by the symbols `/* */`. The lexer must detect the left delimiter of the multi-line comment, and then keep skipping all the symbols until it detects the right delimiter. This means that the lexer must call itself multiple times, using different lexing rules: one to detect the left delimiter, one to handle new lines or characters inside the comment, and one to detect the right delimiter. Furthermore, after handling the comment, the lexer must go back to processing the program normally. The following code shows how to implement such lexer:

```
{
  module Lexer

  let newline (lexbuf : LexBuffer<_>) = lexbuf.EndPos <- lexbuf.EndPos.
      NextLine
  let range (lexbuf : LexBuffer<_>) = lexbuf.EndPos.Line + 1, lexbuf.
      EndPos.Column
}
let newline = ('\n' | '\r' '\n')

rule comment = parse
| "*/"    { programTokens lexbuf }
| newline            { newline lexbuf; comment lexbuf }
| _  { comment lexbuf }

and programTokens = parse
| "/*" { comment lexbuf }
...
//other token definitions
```

Note that `programTokens` calls `comment` when it detects the left delimiter of a multi-line comment. `comment` keeps calling itself until the right delimiter is detected, where it jumps back to `programTokens`.

Definition of a parser in FsYacc

FsYacc allows to define the grammar of a language in terms of productions of a context free grammar. As for the lexer, the parser definition starts with a header where the programmer can specify custom code and modules to use. The grammar defines terminal symbols as tokens, identified by the keyword `%token`. A token specified the name to be used in the grammar productions, and a series of type parameters that are used to store data in a token. For example, the following tokens might be used in a parser for arithmetic expressions:

```
%token PLUS MINUS MUL DIV LPAR RPAR
%token <double> NUMBER
```

Whenever a terminal symbol is encountered during the parsing phase, the parser calls the lexer to generate the data structure for the token. The lexer tries to match the string provided by the parser by using one of its rule and, if it succeeds, it returns the appropriate token data structure. In this part of the grammar we must also specify the starting symbol. This symbol is defined through the keyword `%start`. Since usually we want to generate an abstract syntax tree for the grammar (which must be manually defined), we can specify a return type generated by the parser with the keyword `%type`. For an arithmetic expression this would be, for instance

```
%start start
%type <Expr> start
```

In this section it is also possible to define the operators associativity and precedence, through the keywords `%left`, `%right`, and `%nonassoc`. Terms defined in the same associativity line have the same precedence, and the precedence is ordered according to the line number, so if a term associativity is defined below another, it has higher precedence.

After the terminal symbol definitions, the grammar must specify productions. A production is defined in the following way:

```
productionName:
| rule_1 { action_1 }
| rule_2 { action_2 }
...
| rule_n { action_n }
```

Each action defines the code that the parser executes when that rule is matched. Usually this part is used to build the nodes of the syntax tree, but there is no restriction in what the action can perform, as long as it is valid F# code. It is possible to access the result of evaluating a term in the production by using an index preceded by the symbol `%`, where `%1` refers to the first term in the right hand-side of the production. For example this code might be used to parse an arithmetic expression:

```
%{
  open AST
%}

%token PLUS MINUS MUL DIV LPAR RPAR EOF
%token <float> NUMBER

%left PLUS MINUS
%left MUL DIV

%start start
%type <Expr> start

start : Expression EOF { %1 }

Expression:
| NUMBER { Number %1 }
| Expression PLUS Expression { Plus(%1,%3) }
| Expression MINUS Expression { Minus(%1,%3) }
| Expression MUL Expression { Mul(%1,%3) }
| Expression DIV Expression { Div(%1,%3) }
| LPAR Expression RPAR { Nested(%2) }
```

```
module AST

type Expression =
| Number of float
| Plus of Expression * Expression
| Minus of Expression * Expression
| Mul of Expression * Expression
| Div of Expression * Expression
| Nested of Expression
```

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) [33] parsers but they are extensible, i.e. they do not rely on a limited set of combinators to describe the grammar of a language as for parser generators. Monadic parsers were extensively explained in [33, 62], 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 triplet made of the following elements:

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

where both operations satisfy the following properties:

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

In other words, a monad is a functional design pattern that consists of a data container that provides two operations: one that takes an element whose type is compatible with the element of the container, and returns an instantiation of the container itself, and the other that defines a transformation between two data containers. Monads are a concept borrowed by functional programming language that comes from the much wider concept from category theory [11, 14, 15, 52], whose usage is shown for example in [48, 51, 61, 62]. 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 both Java and 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 zero 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

Being able to verify the correctness of a program is a crucial aspect of programming. When dealing with low-level languages, it is generally difficult to verify and grant the correctness of a program since a language such as assembly does not provide abstractions for the purpose. Modern high-level programming languages, on the other hand, generally provide a way to type their constructs. A type system is a syntactic method that assigns a property called *type* to the constructs of a programming language, in order to prove that a program does not have certain unwanted behaviours [53]. Type systems are generally expressed in the form of inference rules [18, 53], made of a set of premises, that must be verified in order to assign to the language construct the type defined in the conclusion. An inference rule is a logical rule in the form:

$$\frac{\begin{array}{l} premise_1 \\ premise_2 \\ \vdots \\ premise_n \end{array}}{conclusion}$$

where all the premises must be true in order to evaluate the conclusion. Usually the type rules make use of a *typing environment*, which is an association between language constructs and types. For example the following rule defines the typing of an `if-then-else` and a `while-do` statement in an imperative language.

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

$$\frac{\Gamma \vdash c : bool \quad \Gamma \vdash w}{\Gamma \vdash \text{while } c \text{ do } w}$$

In these rules 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 the evaluation of the `then` and `else` block succeeds, then the whole `if-then-else` is correctly typed. Analogously, for the `while-do`, if the condition has type `bool` and the evaluation of the while block is correctly typed, then the whole `while-do` is correctly typed. Note that control structures code blocks are usually not given a type, rather they are considered correct if all their statements are correctly typed. An equivalent way of expressing this is using a special type called *unit* for constructs that do not return a value. This expedient is widely used in hybrid functional programming languages such as F# or CamL. The equivalent rules for the construct above would be:

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

$$\frac{\Gamma \vdash c : bool \quad \Gamma \vdash w : unit}{\Gamma \vdash \text{while } c \text{ do } w : \text{ unit}}$$

Typing a construct of the language requires to evaluate its corresponding typing rule. Unlike for parsers, there exist no tools capable of automatically generating a type checker given the type rules definition, thus the behaviour of each type 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.

During this process the compiler generates a data structure called *symbol table*, which contains information about the type checking process and maintains the type environment.

At the end of the type check, the program is correct with respect to types. At this point, depending on the chosen target language, the compiler might discard or keep the information about the typing process. Usually when targeting a high-level programming language, the information about the types is kept because they are necessary during the code generation process in order to, for instance, generate the proper variable declaration statements. On the other hand, when targeting a low-level untyped programming language, such as assembly, the type information can be discarded.

2.5 Semantics and code generation

Semantics define how the language abstractions behave and can be expressed in different ways, for example with a term-rewriting system [38], reduction semantics [27] or with the operational semantics [29]. Below we provide a description of these three possible representations for the semantics:

Term-rewriting semantics Term-rewriting semantics define a set of rewriting rules that take as input a construct of the language and define how to rewrite it into another form. The rewriting process usually ends when a rewrite rule is replaced by itself. For example the `if-then-else` statements and `while-do` statements can be rewritten with these rules (the `;` symbol denotes a sequence of statements):

```
if true then t else e ; k → t;k
if false then t else e ; k → e;k
while (c = true) do w ; k → w ; while (c) do w ; k
while (c = false) do w; k → k
```

Reduction semantics Reduction semantics use a *reduction context*, which is a program or a fragment of program with a *hole* (denoted by the symbol □) as placeholder to mark where the next computational step is taking place. For example the `if-then-else` and `while-do` statements semantics can be represented in the following way[1]:

```
if □ then s1 else s2 ; k
if true then t else e  → t
if false then t else e → e
while □ do w
while true do w → w ; while □ do w
while false do w → skip
```

Operational semantics Operational semantics define the behaviour of language constructs in terms of logical rules similar to those used for type systems. For instance, the `if-then-else` and `while-do` semantics are expressed as

$$\frac{\langle c \rangle \Rightarrow \texttt{true}}{\langle \text{if } c \text{ then } T \text{ else } E \ ; \ k \rangle \Rightarrow T \ ; \ k}$$

$$\frac{\langle c \rangle \Rightarrow \texttt{false}}{\langle \text{if } c \text{ then } T \text{ else } E \ ; \ k \rangle \Rightarrow E \ ; \ k}$$

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

[1] `skip` is a statement that simply skips to the next statement in a sequence of statements

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

Regardless of the formal representation chosen for the semantics, this must be encoded in the abstractions of the target language during the code generation phase. When choosing a high-level target language encoding the operational semantics of a similar high-level language might be trivial, but generating for instance the code for a functional programming language into an imperative language might prove difficult. For this reason, the code generation step might be preceded by an intermediate code generation step. The intermediate language is usually a simple programming language close to the target language. Notable examples of this are the *three-address code* [8], and the intermediate language used in the *Glasgow Haskell Compiler* [32] and *Utrecht Haskell Compiler* [25].

2.6 Metaprogramming and metacompilers

This section aims to provide the reader with sufficient information to understand the concept of metaprogramming. In this section we explain what metaprogramming is and present existing metacompilation approaches existing in scientific literature. We start by defining what metaprogramming is and we present techniques of metaprogramming in existing programming languages. We then proceed by presenting how different existing metacompilers work.

Metaprogramming is the process of writing computer programs with the ability to treat programs as their data [21]. Metaprogramming takes as input a program written in a meta-language to define a programming language called *object language*, a program written in the object language, and outputs executable code able to run the program. Metaprogramming can be achieved in two different ways: (*i*) by using opportune language abstractions provided by a general-purpose programming language, or (*ii*) using a dedicated metacompiler. In what follows we provide examples in both areas.

2.6.1 Template metaprogramming

Template metaprogramming uses class templates to operate on numbers and types as data. In this section we provide examples in C++ templates, but other languages allow template metaprogramming, with notable examples being Lisp macros and Haskell templates [56]. The template language uses template recursion as loop construct and template specialization as decisional construct.

To better understand how this works, we will implement the factorial function with templates. It is well known that, by definition, the factorial of 0 is 1, while the factorial of a number n is n multiplied by the factorial of $n - 1$. Our template meta-program will thus contain two templates: one for the base case of the recursion and one for the recursive step. The base case of the recursion uses template specialization to stop the computation and immediately return 1:

```
template<>
struct Factorial<0>
{
```

```
  enum { RET = 1 };
};
```

This template contains an enumeration type whose only value is 1. The recursive step will take as input a generic template parameter and recursively call the template definition.

```
template<int n>
struct Factorial
{
  enum { RET = Factorial<n - 1>::RET * n }
};
```

When the `Factorial` template is instantiated with a value different from 0, the non-specialized version is used by the C++ compiler. The enumeration case `RET` then gets the value of `RET` for the same template instantiated for `n - 1` and multiplied by `n`. The generation of templates and their enumeration cases will stop when the template instantiation will be invoked with `Factorial<0>`, which will use the specialized version. Note the use of the scope resolution operator to access the value of the enumeration case. This template can be used instantiating the template with an integer constant, for instance:

```
int main()
{
  cout << Factorial<5>::RET << endl;
}
```

Note that template instantiation is performed at compile-time, so the result of the factorial is actually inlined by the compiler every time the template is instantiated.

A more interesting example is about how to define recursive data structures. Let us consider the implementation of lists in a functional programming language:

```
type List<'a> =
| Empty
| Cons of 'a * List<'a>
```

where the list `[3,4,5,6]` can be built as `Cons(3,Cons (4,Cons(5,Cons(6,Empty))))`. This list representation can be defined with template metaprogramming by defining a template specialization for the empty list, and a non-specialized template for a non-empty list.

```
struct NIL
{
  typedef NIL Head;
  typedef NIL Tail;
};

template<class T, class Tail_ = NIL>
struct Cons
{
  typedef T Head;
  typedef Tail_ Tail;
};
```

Note that the assignment in the template definition specifies an optional template parameter, in the same way as optional method arguments.

Now let us try to define a function to calculate the length of an arbitrary list. This function will have as a base case the empty list, for which it returns 0, otherwise it returns 1 plus the length of the tail. Again this can be implemented with a specialized template and a non-specialized template.

```
template<class List>
struct Length
{
  static const unsigned int RET = Length<List::Tail>::RET + 1;
};

template<>
struct Length<NIL>
{
  static const unsigned int RET = 0;
};
```

The first template recursively class the `Length` template with the type of the tail of the list, extracts the RET field and adds 1. The second template is a specialization created with the `NIL` type and immediately sets the field RET to 0.

In order to test this function we must create a template for a data type (which is a meta-data) that we want to store in the list. In this example we show how to test the function for a list of integers. First of all we must create a template for an integer:

```
template<int n>
struct Int
{
  static const int Value = n;
};
```

This is necessary in order to be able to store the values of the list elements. At this point, the list can be created by calling `Cons` and passing the `Int` data type as argument. Length can then be used with the type of the list that has been created and then we can access its `RET` field.

```
typedef Cons<Int<3>, Cons<Int<4>, Cons<Int<5>, Cons<Int<6>>>>> testList;
cout << Length<testList>::RET << endl;
```

Template metaprogramming complexity can grow exponentially, for example when we want to get the values of the list elements, to the point that a simple function as `nth` requires several templates. We omit the details here, but the reader can find additional information in Appendix A.

2.6.2 Metacompilers

Metacompilers are a special class of compilers used to implement other compilers. A metacompiler takes as input the definition of the syntax, semantics, and possibly the type system of the object language, a program written in the object language, and outputs executable code for it. Metacompilers are written either in a general purpose programming language or in their own meta-language through the process of self-hosting. Self-hosting compilation requires to write a prototypical version of the compiler in another language or an interpreter for it and then use it to compiler the implementation of a subsequent version. In this section we present three existing meta-compilers: (i) META-II for historical reasons to show that research on

meta-compilation had actually been made in early 1970's but the capability of early meta-compilers were limited, (*ii*) RML that is based on natural semantics and, for some aspects, similar to Metacasanova, the meta-compiler that we describe in this work, and finally Stratego, a meta-compiler that is based on term-rewriting semantics, to show an alternative approach to meta-compilers based on natural semantics.

META-II

META-II is one of the earliest metacompilers and, for this reason, quite limited in its capabilities. META-II allows to express the syntax of the object language and actions for the code generation. A meta-program in META-II is made of grammatical symbols, meta-variables, and equations that define the terms of the grammar. A symbol is written as a string surrounded by quotes and beginning with a period, a meta-variable is a string starting with a alphabetical character and followed by an arbitrary amount of alphanumerical characters, and an equation is a sequence of consecutive symbols or ids to indicate concatenation. Alternation is defined with the symbol `/`, which can be used together with the keyword `.EMPTY` to define alternation. For instance

```
BOOLEAN = '.TRUE' / '.FALSE'
```

defines boolean literals. The meta-language is able to recognize built-in symbols such as identifiers, denoted with `.ID`, strings represented by .STRING, and numbers represented by .NUMBER. These are to be intended as identifiers, strings, and numbers in the object language. For example the family of expressions:

```
A
A + B
A + B * C
(A + B) * C
```

can be encoded by the following equations in META-II

```
EX3 = .ID / '(' EX1 ')',
EX2 = EX3 ('*' EX2 / .EMPTY),
EX1 = EX2 ('+' EX1 / .EMPTY)
```

Sequences (as in the Kleene closure for regular expressions) can be expressed using the symbol `$`. For example

```
SEQA = $ 'A',
```

represents a sequence containing the letter `A`. META-II allows to associate actions to equations for the code generation. Each action generates assembly code for an interpreter called META-II machine, which is able to execute it. The action of code generation is marked with the keyword `.OUT`, for instance

```
EX3 = .ID .OUT('LD ' *) / '(' EX1 ')'
```

generates the literal output and the special symbol found in `EX3`.

META-II is a self-hosting compiler, i.e. it is implemented in META-II itself.

RML

RML [50] (*Relational Meta-Language*) uses a meta-language based on operational semantics. A program in RML consists of data definitions in a syntax similar to CamL variants [45] or F# discriminated unions [44], and relations containing axioms and inference rules. An axiom is an inference rule without premises, while an inference rule generates an output if the premises correctly evaluate. For example the following snippet defines the data type for an arithmetic expression and a symbol table for the evaluation.

```
datatype Expr =
| INT of int
| VAR of string
| ADD of Expr * Expr

type Env = (string * int) list
```

Axioms and inference rules are grouped together into a *relation*. For example, the following relation can be used to evaluate an arithmetic expression:

```
relation eval =
  axiom eval(env, INT i) => i

  rule
    lookup(env, x) => i
    ---------------------
    eval(env, VAR x) => i

  rule
    eval (env, left) => i1 &
    eval (env, right) => i2
    i1 + i1 => v
    ---------------------------------
    eval (env, ADD(left,right)) => v
```

During the code generation phase, each relation is translated into a first-order logic representation, which consists of a series of `match` structures that check the structure of the arguments. For example the rule above would be translated into:[2]

```
(and (match [(arg1 env)
             (arg2 ADD(left,right))]))
(and (call eval [env left] [result1]))
(and match [result1 i1])
(and (call eval [env right] [result2]))
(and match [result2 i2])
(and call [i1 + i2] [result3])
(and match [result3 v])
(return v)
```

This code is later translated into a continuation-passing style form, which is later generated as C code. The compiler performs heavy optimization on tail calls generated code through the use of a technique called *dispatching switches*.

[2] We use a prefix notation in Lisp style

Stratego

Stratego [17] is a metacompiler that uses a term-rewriting semantics as meta-language to define its programs. A stratego program consists of a series of terms in the form

$$t := c(t_1, t_2, ..., t_n)$$

where c is a constructor that accepts n other terms as arguments. The syntax of Stratego has been enriched with additional syntax to handle "traditional" data structures, such as string, integer, float, constants, and lists:

$$pt := s \mid i \mid |f \mid [t_1, t_2, ..., t_n] \mid (t_1, t_2, ..., t_n) \mid c(t_1, t_2, ..., t_n)$$

Terms can be extended with a list of annotations that are terms themselves:

$$t := pt \mid pt\,\{t_1, t_2, ..., t_n\}$$

Stratego requires that the meta-program specifies the signature of term constructors. For example simple arithmetic expressions can be defined as

```
signature
  sorts Id Expr
  constructors
    Var : Id -> Exp
    Plus : Exp * Exp -> Exp
```

Note that Stratego is an untyped language, so types are not statically checked and the compiler only checks that constructors are declared and have the correct ariety.

Rewrite rules define how terms are evaluated, for example the following is a rewrite rule to evaluate a binary operator in an arithmetic expression:

```
EvalBinOp : Plus(Int(i), Int(j)) -> Int(k) where k := <add>(i, j)
```

Note that rewrite rules support conditionals, i.e. in the rule above we are able to specify that `k` is the result of adding the numbers `i` and `j` given as arguments.

Stratego compiler is a self-hosting compiler, meaning that the Stratego meta-language is defined in the meta-language itself. A first version of Stratego was written in SML, which was then re-used to compile a further iteration written in Stratego. Stratego compiles programs to C, where the code generation transformations were expressed in Stratego itself.

2.7 Summary

In this chapter we presented the fundamental topics necessary to understand this thesis work. We started by defining the general architecture of a compiler and then we proceeded to show how to implement its single components. We explained how to use regular expressions to define the syntactical elements of a language and how to build a lexer for them. W explained how grammars are described and how to implement a Parser with different techniques (Parser generators based on LR(k) grammars and Monadic Parsers). We explained how to define a type system for a language and

how to express its operational semantics. We concluded by presenting three examples of existing meta-compilers. In the next chapter we present the detailed architecture of Metacasanova, a meta-compiler based on operational semantics, that is able to generate .NET code. This meta-compiler shares some similarities with RML, which was presented in this section, but has different additional features, such as multiple inheritance, custom operator notation and ariety, and higher-kinded modules and functors in its language extension presented in Chapter 5.

Chapter 3

Metacasanova

> Typing is no substitute for thinking
>
> Dartmouth Basic manual, 1964

This chapter aims to provide the reader with additional motivation about the advantage of meta-compilers and sufficient details on the Metacasanova compiler architecture. In Section 3.1 we show that there exists a common pattern when implementing the formal definition of the type system and semantics of a programming language into the abstraction of a general-purpose programming language. In Section 3.2 we define the requirements of Metacasanova and give an informal overview about the structure of a meta-program and we provide a formalization of its semantics. In Section 3.3 we explain the working principles of the meta-compiler and the involved compilation stages . In Section 3.4 we present the details of the Metacasanova grammar and parser; we then explain the subsequent parsing post-processing phase and how the post-processor re-processes the generated AST. In Section 3.5 we explain how the type checking of a meta-program is performed and in what cases it fails. In Section 3.6 we explain how the abstractions of Metacasanova are mapped into the abstractions of the C# target code.

3.1 Repetitive steps in compiler development

In Chapter 2 we gave on overview of the necessary steps involved in developing a compiler. We showed that the lexing/parsing phase is simple enough to be automated using a lexer/parser generator. Such software takes as input the grammar and the definitions of regular expressions to define the tokens of the language and produces output code containing that is able to parse a program written in a programming language defined by that grammar. However, the steps involved in the following phases, namely the *type checking* and *operational semantics* implementation follow a recurring pattern, but in general the behaviour of the type system and the code generation reflecting the behaviour of the operational semantics must be hard-coded in

the host language in which the compiler is being implemented. Below we present two examples to show how these behaviours can be implemented in two different general purpose programming languages and show that both follow the same pattern.

3.1.1 Hard-coded implementation of type rules

As shown in Section 2.4, type rules are expressed in the form of logical rules. Let us consider the type rules for the `if-then-else` and `while-do` statements presented in Section 2.4 in the version that assigns the type *unit* to the code blocks for convenience. In a programming language that supports discriminated unions as a language abstraction (like Haskell or F#), the syntactical element in the abstract syntax tree of the language can be expressed as

```
type Statement =
| If of Expr * List<Statement> * List<Statement>
| While of Expr * List<Statement>
... //other statements
```

The type checking of the `if` statement requires that to check that the condition has type `bool` and that both code blocks have type `unit` (or `void`). The type checking of the `while-do` is analogous, except only one code block is used. We can then define a function `eval` that, given the environment (here we call it *symbol table*) and a statement as input, returns the type given by the rule or an error if all type rules for that statement fail to correctly evaluate. For the `if-then-else` the implementation is the following:

```
let rec evalStmt (symbolTable : SymbolTable) (stmt : Statement) : Type =
match stmt with
... //other statements
| If (condition,_then,_else) ->
    let conditionType = evalExpr symbolTable condition
    let thenType = evalStmt symbolTable _then
    let elseType = evalStmt symbolTable _else
    if conditionType <> Boolean then
      failwith "Invalid condition type"
    elif thenType <> Unit then
      failwith "The type of then must be unit"
    elif elseType <> Unit then
      failwith "The type of else must be unit"
    else
      Unit
... //other statements
```

The function first executes pattern matching on the statement to identify the correct inference rule to use during the typing. It then proceeds to evaluate the premises (type of the condition and of the statement blocks) and to check their result. If all premises evaluate successfully the type contained in the conclusion is returned. Note that the function `evalExpr` is a function able to evaluate the type rule for expressions and return their type. The implementation of the `while-do` follows the same logic:

```
let rec evalStmt (symbolTable : SymbolTable) (stmt : Statement) : Type =
match stmt with
... //other statements
| While (condition,_do) ->
  let conditionType = evalExpr symbolTable condition
```

```
  let doType = evalStmt symbolTable stmt
  if conditionType <> Boolean then
    failwith "Invalid condition type"
  elif doType <> Unit then
    failwith "The type of the do block must be unit"
  else
    Unit
... //other statements
```

In languages that do not provide abstractions such as discriminated unions, the type for statements must exploit polymorphism to implement the same behaviour. A statement will be represented as an interface exhibiting the behaviour of a visitor pattern:

```
public interface Statement
{
  Type Visit(StatementVisitor visitor);
}

public interface StatementVisitor
{
  ... //other statements
  Type OnIf(Expression condition, List<Statement> _then, List<Statement>
        _else);
  Type OnWhile(Expression condition, List<Statement> _do);
  ... //other statements
}
```

The behaviour of the inference rule for the `if-then-else` statement is modelled by a class implementing the `StatementVisitor` interface. This class contains a method `OnIf` that implements the behaviour of the type rule itself.

```
public class StatementEvaluator : StatementVisitor
{
  ... //evaluation of other statements
  public Type OnIf(Expression condition, List<Statement> _then, List<
       Statement> _else)
  {
    Type conditionType = condition.visit(new ExpressionEvaluator());
    Type thenType = _then.Visit(new StatementEvaluator());
    Type elseType = _else.Visit(new StatementEvaluator());
    if (!conditionType.Equals(new Boolean()))
    {
      throw new TypeException("Invalid condition type");
    }
    else if (!thenType.Equals(new Unit()))
    {
      throw new TypeException("The type of then must be unit");
    }
    else if (!elseType.Equals(new Unit()))
    {
      throw new TypeException("The type of else must be unit");
    }
    else
    {
      return new Unit();
    }
  }

  ... //evaluation of other statements
}

public class If : Statement
{
```

```
  Expression Condition;
  List<Statement>  Then;
  List<Statement> Else;

  public Type Visit(StatementVisitor visitor)
  {
    return visitor.OnIf(this.Condition, this.Then, this.Else)
  }
}
```

Analogously for the `while-do` we have

```
public class StatementEvaluator : StatementVisitor
{
  ... //evaluation of other statements
  public Type OnWhile(Expression condition, List<Statement> _do)
  {
    Type conditionType = condition.Visit(new ExpressionEvaluator());
    Type doType = _do.Visit(new StatementEvaluator());
    if (!conditionType.Equals(new Boolean()))
    {
      throw new TypeException("Invalid condition type");
    }
    else if (!doType.Equals(new Unit()))
    {
      throw new TypeException("The type of do must be unit");
    }
    else
    {
      return new Unit();
    }
  }
  ... //evaluation of other statements
}

public class While : Statement
{
  Expression Condition;
  List<Statement> Do;

  public Type Visit(StatementVisitor visitor)
  {
    return visitor.OnWhile(this.Condition, this.Do);
  }
}
```

Generalization

In general, for a node of the abstract syntax tree (AST) α (like `Statements`) containing syntactical structures σ_i constructed with a certain amount of arguments of type $\epsilon_{\sigma_{i_1}}, ..., \epsilon_{\sigma_{i_m}}$ (such as the condition or the statement block in a control structure), the general representation of a hard-coded type rule in a language with discriminated unions is obtained by creating a union type α having a case σ_i with arguments $\epsilon_{\sigma_{i_j}}$ for each syntactical element.

```
type α =
|σ1 of τσ11 * ... * τσ1m
...
|σn of τσn1 * ... * τσnm
```

Evaluating the inference rule through an evaluation function requires first to find out which must be applied by matching the pattern of the syntactical structure from the node of the AST. Later, we need to evaluate each of the premises with the appropriate evaluation function: if the result of each evaluation is what the rule expects (for instance, that the condition has type boolean in the `if-then-else`) then we return the result of the evaluation rule contained in the right part of the conclusion.

Let us consider a conclusion $\sigma_j(\epsilon_{\sigma_{j_1}} \ ... \ \epsilon_{\sigma_{j_m}})$ (where each ϵ is one of the arguments used to construct the case of the discriminate union) with a result of the evaluation ρ_{σ_j} a set of premises $\pi_1, ..., \pi_k$ that are evaluated through an evaluation function φ_{π_i}, $i = 1, ..., k$ returning a result ρ_{π_i}. Let us assume that ρ'_{π_i} is the expected result for the premise evaluated through φ_{π_i}. As usual, Γ defines the environment (symbol table). The type rule that we are trying to execute will thus have the following structure:

$$\frac{\begin{array}{l}\Gamma \vdash \varphi_{\pi_1}\pi_1 : \rho'_{\pi_1} \\ \vdots \\ \Gamma \vdash \varphi_{\pi_i}\pi_i : \rho'_{\pi_i} \\ \vdots \\ \Gamma \vdash \varphi_{\pi_k}\pi_k : \rho'_{\pi_k}\end{array}}{\Gamma \vdash \sigma_j(\epsilon_{\sigma_{j_1}}, ..., \epsilon_{\sigma_{j_m}}) : \rho_{\sigma_j}}$$

Given the considerations above, the code necessary for the evaluation will be the following:

```
let rec φ_{σ_j} Γ σ =
  match σ with
  ... //other pattern matching expressions for other rules
  | σ_j(ε_{σ_{j_1}}, ..., ε_{σ_{j_m}}) ->
    let ρ_{π_1} = φ_{π_1} Γ π_1
    .
    .
    .
    let ρ_{π_i} = φ_{π_i} Γ π_i
    .
    .
    .
    let ρ_{π_k} = φ_{π_k} Γ π_k
    if ρ_1 <> ρ'_1 then
      failwith "Type error"
    .
    .
    .
    elif ρ_i <> ρ'_i then
      failwith "Type error"
    .
    .
    .
    elif ρ_m <> ρ'_m then
      failwith "Type error"
    else
      ρ_{σ_j}
  ... //other pattern matching expressions for other rules
```

Each evaluation function is recursive because a premise might need to run the same evaluation function (see the example of the statements above). The function contains a pattern matching that selects the correct inference rule to be used for that syntactical structure. For example, in the case of the statements, it will try to match all the possible syntactical structures for the statements and select the correct one for the input; for instance, if we are running the rule for the `if-then-else` then the pattern matching will select the match case for `if-then-else`. Note that γ will surely be matched by one of the match cases because at this point we have a correctly generated AST after the parsing phase.

Each premise runs the appropriate evaluation function and returns a result. This result is compared with the one expected by the inference rule, and if the comparison fails the function reports a type error. If all comparisons succeed, then the result of the conclusion is returned.

In a language that does not provide discriminated unions and pattern matching the generalization is more complex: the abstract syntax tree element must be represented by an interface containing the signature of a method `Visit` used to perform an operation on a specific (polymorphic) syntactical structure. We also require the interface for the visitor pattern with the signature of the functions to run for each polymorphic instance of `Statement`. In this version we assume that the type of the result of the evaluation function for σ_j returns a type $\tau_{\rho_{\sigma_j}}$ (which in the previous version could be omitted thanks to the type inference typical of functional programming languages):

```
public interface α
{
  public τ_{σ_j} Visit(Visitor visitor)
}

public interface Visitor<T>
{
  T φ_{σ_j}(τ_Γ Γ, τ_{σ_{j_1}} ε_{σ_{j_1}}, ..., τ_{σ_{j_m}} ε_{σ_{j_m}});
  ... //other statements
}
```

Then we have to implement the visitor for the type rule for statements and a class for a specific statement:

```
public class Evaluator : Visitor<τ_{ρ_{σ_j}}>
{
  ... //evaluation of other statements
  public τ_{ρ_{σ_j}} φ_{σ_j}(τ_Γ Γ, τ_{σ_{j_1}} ε_{σ_{j_1}}, ..., τ_{σ_{j_m}} ε_{σ_{j_m}})
  {
    τ_{ρ_{φ_1}} ρ_{π_1} = φ_{π_1}(Γ, π_1);
    .
    .
    .
    τ_{ρ_{φ_i}} ρ_{π_i} = φ_{π_i}(Γ, π_i);
    .
    .
    .
    τ_{ρ_{φ_k}} ρ_{π_k} = φ_{π_k}(Γ, π_k)
    if (!ρ_{π_1}.Equals(ρ'_{π_1}))
      throw new TypeError("Type error");
    .
    .
```

```
    .
    else if (!ρ_{π_i}.Equals(ρ'_{π_i}))
      throw new TypeError("Type Error");
    .
    .
    .
    else if (!ρ_{π_k}.Equals(ρ'_{π_k}))
      throw new TypeError("Type Error");
    else
      return new ρ_{σ_j}();
  }
  ... //evaluation of other statements
}

public σ_j : α
{
  τ_{σ_{j_1}} ε_{σ_{j_1}};
  ...
  τ_{σ_{j_m}} ε_{σ_{j_m}};

  public τ_{ρ_{σ_j}} Visit(Visitor<τ_{ρ_{σ_j}}> visitor)
  {
    return visitor.φ_{σ_j}(ε_{σ_{j_1}}, ..., ε_{σ_{j_m}});
  }
}
```

A general pseudo-code representation of the rule evaluation is shown in Algorithm 3.1.

Algorithm 3.1 Pseudocode of rule evaluation

```
function EVALUATE RULE(R inference rule , I input of the rule )
    if not R.Conclusion matches I then
        return error
    end if
    for all p in R.Premises do
        p' ← textbf evaluate p
        if not p.Result matches p then
            return error
        end if
    end for
    return R.Result
end function
```

A graphical representation of the rule evaluation can be found in Figure 3.1.

3.1.2 Hard-coded implementation of Semantics

As shown in Section 2.5, there are multiple ways to express the semantics of a programming language. In this work we choose to make use of the operational semantics representation to have a uniform way of expressing both the type system and the semantics of a language. Let us consider again the semantics rule for `if-then-else` and `while-do` presented in Section 2.5. The operational semantics can be implemented generating the code in the object language that emulates the behaviour of the semantics rule, in

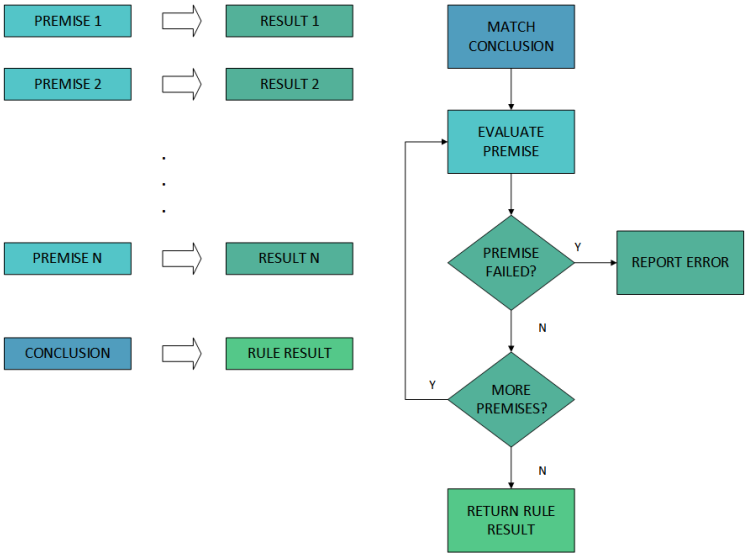


Figure 3.1: Diagram of rule evaluation: on the left side the structure of an inference rule, on the right side its evaluation expressed as a flow chart. The components of the rule are coloured to match the parts in which they are used in the flow chart.

the same fashion of a type rule. This process might first pass through an intermediate language, closer to the target language. In the case of an interpreter, the behaviour of the semantics must be implemented using the abstractions available in the host language. As an example, we show a possible implementation of the semantics of the two statements mentioned above in an interpreter both in a functional programming language and in an object-oriented language, as for the type rule.

For convenience, let us make a separate rule for the semantics of a sequence of statements from the specific semantics of the control structure. Also, we introduce the statement `skip` that performs no operation

$$\frac{}{\langle \text{skip}; ks\rangle \Rightarrow \langle ks\rangle}$$

$$\frac{\langle k\rangle \Rightarrow k'}{\langle k; ks\rangle \Rightarrow \langle k'; ks\rangle}$$

$$\frac{\langle c\rangle \Rightarrow \texttt{true}}{\langle \text{if } c \text{ then } T \text{ else } E\rangle \Rightarrow T}$$

$$\frac{\langle c\rangle \Rightarrow \texttt{false}}{\langle \text{if } c \text{ then } T \text{ else } E\rangle \Rightarrow E}$$

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

$$\frac{\langle c\rangle \Rightarrow \texttt{true}}{\langle \text{while } c \text{ do } L\rangle \Rightarrow \text{skip}}$$

As for the type rules, we assume that the data type representing a statement is implemented by a discriminate union. The evaluation function first performs the pattern matching on the argument to select the correct rule to execute, in the same fashion of the type rule, but instead of analysing the types this time executes the specific behaviour of the statement, as specified by the semantics rule. For inference rules above we use the following code:

```
let rec interpretStmt (symbolTable : SymbolTable) (stmt : Statement) :
     Statement =
  match stmt with
  ... //other statements semantics
  | Sequence(Skip,ks) -> interpretStmt symbolTable ks
  | Sequence(k,ks) ->
     let k' = interpretStmt symbolTable k
     interpretStmt symbolTable Sequence(k',ks)
  | If (cond,_then,_else) ->
      let condEvaluation = interpretExpr symbolTable cond
      if condEvaluation = True then
        _then
      else
        _else
  | While (cond,_do) ->
     let condEvaluation = interpretExpr symbolTable cond
     if condEvaluation = true then
       Sequence(_do,While(cond,_do))
      else
        Skip
  ... //other statements semantics
```

As for the type evaluation, we assum that `interpretExpr` is another function that is able to process expressions and return their value.

The function matches the kind of statement that we want to execute. In the case of a sequence of statements starting with a skip, we simply return the interpretation of the remaining part of the sequence (we indeed skip to the next statement), otherwise we have to run the first statement and then recursively evaluate the sequence formed by the result of the execution of the statement and the rest of the statements. This is needed, for instance, to correctly evaluate the body of a control structure. The body of each match case is responsible of emulating the intended behaviour described in the semantics of the control structure: the `if` returns the correct block to execute depending on the boolean value of the condition, instead the `while` returns either its body followed by the same `while` loop when the condition is `true`, otherwise `skip` to jump past the loop.

In the case of an object-oriented language, it is necessary to add a new implementation of the visitor pattern implementing the behaviour of the semantics for each statement:

```
public class StatementInterpreter : StatementVisitor
{
  ... //other statements semantics
  public Statement OnSequence(SymbolTable symbolTable, Sequence seq)
  {
    Statement k = seq.Head;
    Statement ks = seq.Tail;
    if (k.Equals(Skip))
      return ks.Visit(new StatementInterpreter());
    else
    {
      Statement k1 = k.Visit(new StatementInterpreter());
      Statement seq1 = new Sequence(k1,ks)
      return seq1.Visit(new StatementInterpreter());
    }
  }
  public Statement OnIf(Expression cond, Statement _then, Statement
       _else)
  {
    Value condValue = cond.Visit(new ExpressionInterpreter());
    if (condValue.Equals(new True()))
      return _then;
    else
      return _else;
  }
  public Statement OnWhile(Expression cond, Statement _do)
  {
    Value condValue = cond.Visit(new ExpressionInterpreter());
    if (condValue.Equals(new True()))
      return new Sequence(_do,new While(cond,_do))
    else
      return new Skip();
  }
  ... //other statement semantics
}
```

A further remark is that, for the sake of simplicity, here the interpretation only returns a new statement to execute obtained by processing the current statement, but in a real application it should also return a data structure representing the state of the program.

At this point it is possible to observe that this pattern can be generalized as well in a way analogous to that used for type rules for both implementa-

tions, which we omit for brevity.

3.1.3 Discussion

In Section 3.1.1 and 3.1.2 we have shown two implementations, one functional and one object-oriented, of type rules and semantics in a possible hard-coded compiler. We have also shown that the pattern can be generalized in both versions. Indeed, their behaviour must be hard-coded in the language chosen for the compiler implementation, regardless of the fact that the 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 generates the code in the target language necessary to execute the behaviour of the rules. In the following sections 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. The goal of this language is relieving the programmer from writing boiler-plate code when implementing a compiler for a (Domain-Specific) language.

3.2 Metacasanova overview

In this section we present the general idea behind Metacasanova. We start by defining the requirements of Metacasanova, then we proceed to give a general overview of the language, and finally we formalize the semantics of the language.

3.2.1 Requirements of Metacasanova

In order to relieve programmers of manually defining the behaviour described in Section 3.1.1 and 3.1.2 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.

3.2.2 Program structure

In this section we give an informal idea of how a Metacasanova program is organized. Further ahead this idea is expanded with additional details when we present the implementation details of the parser.

A Metacasanova program is mainly organized in three parts:

1. *Data and function declarations:* in this part it is possible to specify data structures, that define the syntactic constructs of the language, and functions used to evaluate terms of the language through rules.

2. *Type equivalence declarations:* in this part it is possible to specify polymorphism through type equivalence by stating that a type T_1 is equivalent to another type T_2.

3. *Rule definitions:* in this part the programmer defines the type or semantics rules necessary to describe the type system or behaviour of the abstractions of the programming language.

A data structure or function declaration specifies the types of the arguments to construct the data structure or to pass to the function, their name, and the type of the data structure or the function itself

```
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
```

Optionally, it is possible to specify a precedence priority and the associativity. For example, the following code specifies that the multiplication has a higher precedence over the sum and that both are left-associative.

```
Data Expr -> "+" -> Expr : Expr Priority 0 <|
Data Expr -> "*" -> Expr : Expr Priority 1 <|
```

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
```

A polymorphic data structure is defined through the keyword `is`, which specifies that a type T_1 is equivalent to another type T_2. For example the following code specifies that a data structure of type `Value`, such as a list, can be used also as an expression of type `Expr`.

```
Data "$l" -> List : Value
Value is Expr
```

Metacasanova also allows to embed C# code into the language by using double angular brackets. This code can be used to embed .NET types when defining data or functions, or to run C# code in the rules. For example in the following snippet 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 may contain a sequence of premises and a conclusion. The rule is executed if the input matches the pattern of the conclusion and all the premises return a result that matches the one specified in their rightmost part. 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
```

A more exhaustive explanation of the syntax of Metacasanova is given in Section 3.4, while an overview of the general shape of a program can be found in Figure 3.2.

3.2.3 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).

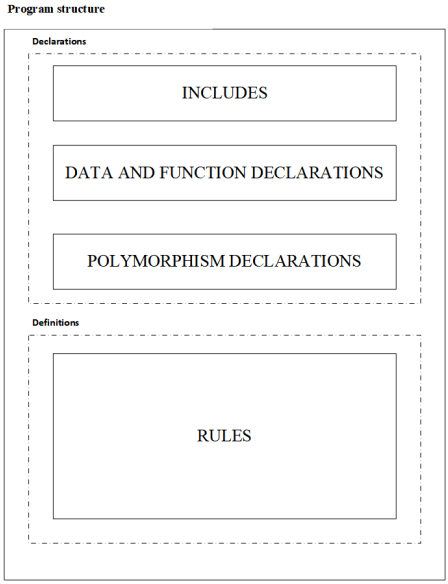


Figure 3.2: Structure of a program in Metacasanova

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.

3.3 Architectural overview

In this section we provide a general overview of the architecture of Metacasanova compiler.

The compiler has a modular structure: in the front-end we find the the lexer/parser and a parser post-processing module. The latter is required because not all information necessary to build all the elements of the AST is immediately available during the parsing phase. For instance, some data structures store the file name that is being compiled and the name of the current module, but this information is available only after the parsing itself.

The generated AST is passed to the type checker to check the type correctness. Note that the type checker of the metacompiler checks the meta-types, i.e. the types defined in the meta-program, and not the types of the terms of the language that is being implemented in Metacasanova. This module checks the correctness of the declarations and the terms used in rules. The type checker outputs a data structure containing information about the types of the declarations and terms used in rules (in short a *typed program definition*).

The output of the AST is passed to the code generator that uses information about the types to correctly generate the target code. This is needed because Metacasanova generates C# code, which is a typed high-level language that requires information about the types to define variables, methods, and classes representing the elements of the meta-program.

Note that also, with this implementation choice, it is possible to support different high-level programming languages, both typed and untyped: the only component that changes will be the generation of the behaviour of the rules in the abstractions provided by the different target languages. A possible improvement of this architecture is generating a common intermediate language that is later translated into the target code, but this falls outside the scope of this work.

3.4 Parsing

In this section we explain in detail the grammar of Metacasanova and we present the architecture of its parser. The parser has been built in FsYacc (see Section 2.3.2) and completed by a post-processing module that executes some required transformation on the generated AST that are not convenient to perform during the parsing phase. As explained informally in Section 3.2, a Metacasanova program is made of four main sections: (*i*) a part containing inclusion directives, (*ii*) a part containing the declarations of the meta-data stracutures and evaluation functions used in the program, (*iv*) a part containing type equivalences, and (*iii*) a part containing evaluation rules that define the behaviour of the meta-program. The definition of the first part is trivial and we do not examine it in detail. We will instead describe in detail the other parts.

3.4.1 Declarations

Declarations contain *function* or *data* declarations. A meta-data structure is a meta-language representation of an abstraction of the language implemented in the metacompiler and contains both syntactical and structural information. For example, an arithmetic operator in a programming language can be represented as a meta-data structure containing both its symbol and the values of its arugments. Meta-data structures can be recursive, i.e. it is possible to have arguments that are instances of the same meta-data structure. This is done in order to allow recursive data structures such as lists. A meta-data structure declaration begins with the keyword `Data` and is followed by a series of arguments, which are separated by arrows, that can be both type names and strings representing the name of the meta-data. It is possible to declare an infix or suffix operator by placing its name after the first position of the arguments. For instance, the following code defines a sum operator for arithmetic expressions with an infix notation.

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

Type names are identifier that begins with an alphabetic character followed by one or more alphanumeric characters or underscores. The regular expression defining this syntax is expressed as

```
ID ::= ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '_' '0'-'9']+
```

Type names can also contain external code enclosed by double angular brackets to make use of external types such as .NET primitive types (`int`, `float`, etc.). The operator names are strings that may contain any symbol usually allowed in strings in usual programming languages.

The arguments are followed by a type name defining the type of the meta-data structure. Optionally it is possible to specify a priority and the associativity, otherwise the default priority will be -1 and the operator will be left-associative.

Function declarations have the same structure except they begin with the keyword `Func` instead.

Both data and function declarations may define generic arguments. In order to specify generic arguments, they must be enclosed between square brackets after the declaration keyword. For instance the following code defines a data structure representing a tuple

```
Data[a,b] a -> "," -> b : Tuple[a,b]
```

Finally, each declaration must end with a line break. Line breaks are used in Metacasanova to separate different language elements, as in this case.

The grammar production used to describe the syntax of declarations is the following:

```
declaration:
| FUNC genericSeq typeOrNameDeclarations COLON typeDeclaration priority
     associativity newLineSeq {
Func(processParsedArgs $3 $5 (fst $1) (snd $1) $2 $6 $7) }
| DATA genericSeq typeOrNameDeclarations COLON typeDeclaration priority
     associativity newLineSeq {
Data(processParsedArgs $3 $5 (fst $1) (snd $1) $2 $6 $7) }
```

Since type names and data or function names can appear in any order, the parser generates a support polymorphic data structure in F#.

```
type TypeDeclOrName =
| Type of TypeDecl
| Name of string
```

This data structure is later transformed by the parser post-processor in a *symbol declaration*. A symbol declaration contains all the information about a declaration, including the data or function name, the types of the arguments, the priority, and the generic types. This information is later exploited by the type checker to verify the consistency of the declarations and type check the rest of the program.

Type equivalence declarations

Type equivalence has been presented as a separate part of the program but it has a tight relationship with the declarations. A type equivalence has the form

```
T1 is T2
```

where T1 and T2 are two meta-type names. They are used to specify that meta-type T1 is equivalent to meta-type T2 and can replace any argument of type T2 while constructing meta-data structures or calling functions. The grammar rule that defines a type equivalence declaration is

```
ID IS ID newLineSeq
```

where ID is the same regular expression used for type names. Again successive equivalence declarations should be separated by one or more line breaks. The grammar production generates a list of pairs in the AST containing the types involved in the equivalence. This data structure will be processed at a later stage by the type checker to generate an equivalence table.

3.4.2 Rules

Rules in Metacasanova are the language elements used to define the behaviour of the meta-program. A rule consists of a set of premises followed by a conclusion. Premises and conclusion are separated by a fraction line. Premises and conclusion are made of a left part consisting of a sequence of arguments, and a right part that can contain either a variable or a sequence of arguments. We call the left part of this syntactical structure *function call*, while the right part is the *result*. Premises differ from the conclusion as, besides function calls, they can also contain *bindings* and *clause*. Bindings are simply ways to rename values in the premises in the fashion of bindings in functional languages, while clauses are boolean predicates. Moreover, premises can also contain .NET code to directly emit, which can contain any C# code. The syntax of emitted code is the same as that of a normal premise, except the *function call* is replaced by the code to emit enclosed in double angular brackets. The following is the grammar production defining a premise[1]:

```
premise:
| emit premises { $1 :: $2 }
| functionCall premises { $1 :: $2 }
| ID BIND arg newLineSeq premises { (Bind({ Namespace = ""; Name = fst
     $1 },Position.Create(snd $1,""),$3)) :: $5 }
| arg comparisonOp arg newLineSeq premises { (Conditional($1,$2,$3)) ::
     $5 }
| { [] }

functionCall:
| argSeq ARROW argSeq newLineSeq { FunctionCall($1,$3) }
```

Note that premises are optional (axioms do not have any premise in a logical rule), so an empty list is returned when none is given. Note that the namespace required for variables, such as in the binding, is left empty because at this point the namespace of the program is not available yet. The namespace will be later filled in by the parser post-processor. Also note that, in this stage, we do not check if each function call actually contains a function name, and we simply parse a premise (and a conclusion) as a sequence of arguments, that could be function or data names, variables, literals, or nested expressions. Nested expressions are expressions enclosed in brackets and are themselves other sequences of arguments. The actual control that premises and conclusions contain a function name is performed by the type checker, because to correctly identify the function name a complete symbol table, unavailable in this moment, is required. A conclusion has the same syntax of a function call:

```
conclusion:
| argSeq ARROW argSeq newLineSeq { ValueOutput($1,$3) }
```

[1] BIND is the symbol :=

The parser generates a data structure for a rule containing a representation of the premises and the conclusion:

- *function call*: A function call is simply a pair of list of arguments, where the left element is the call itself, while the right element is the result.
- *emit*: Emitted code contains the code in string format and the variable it is assigned to (used to save the result of expressions or function calls).
- *bind*: Bindings contain the variable name used for the binding and its argument, which can be a literal, the constructor of a meta-data structure, or another variable.
- *conditional*: Conditionals are boolean expressions that may contain comparison operators. Their representation stores their left and right argument and the comparison operator.

3.4.3 Parser post-processor

The parser post-processor is responsible to integrate in the AST all the information that is not directly available during the parsing phase. It is also responsible to re-arrange the terms appearing in the data and function declarations, and those appearing in function calls. Its main functions are the following: (*i*) insert the namespace and file information in the elements of the AST, (*ii*) rearrange the terms parsed from a declaration in a *symbol declaration* data structure, and (*iii*) parenthesize the function call terms according to their priority and associativity and rearrange them in a prefix notation.

The first task is trivial and will not be explained in details. Suffice to say that the process scans the AST starting from the root and recursively add the namespace and file name into all the nodes that must contain such information until a leaf is reached. Below we explain in detail the other how to accomplish the other two tasks.

Building the declaration data structure

As anticipated in Section 3.4.1, the name of the data or function and the types of its arguments can appear in any order. For convenience, the AST stores a data structure called *symbol declaration* that separates all the information about a declaration for further use, which is made of the following components:

- The name of the data or function.
- The type of the arguments of the data or function.
- The return type of the declaration: in the case of a data declaration this defines the type of the meta-data structure itself, while in a function declaration this defines the type of the result returned by the function.
- The operator arguments order, which can be prefix, infix, or suffix.
- The priority of the operator.
- The associativity of the operator.
- Possible generic arguments.

- The arity (amount of arguments) to the left and right of the operator symbol.

When the parser processes the arguments of a declaration, it creates a list of terms that can be either a type declaration or a name assigned to the meta-data or function in `string` format. The different arguments must be recognized and stored appropriately in the symbol declaration. In order to do so, the post-processor scans the arguments and, if the argument is a string, then it places it as a first element of a pair, otherwise it places the type declaration in a list of declarations. Algorithm 3.2 shows the details of this process. Note that it might be possible that the programmer commits the mistake of defining more than one name for the declaration, since we are still checking the syntax of the program. Thus the algorithm checks that the result of the function does not already contain a declaration name as, if it does, it means that a name argument has already been encountered while scanning the list.

The symbol declaration needs also to store the order of the declaration, its left and right arity, and the type of the declaration.

For the declaration order we have three options:

1. The first element of the parsed arguments is the declaration name. The declaration is then **prefix**.
2. The last element of the parsed arguments is the declaration name. The declaration is **suffix**.
3. If both 1 and 2 are false, then the name is in the middle and the declaration is **infix**.

Finding the left and right arity of an operator can be done simply by splitting the list of arguments in correspondence of the declaration name and then counting the elements of the two lists obtained by the split.

Finally, the post-processor must build the type of the declaration. Types in Metacasanova are represented in a way similar to typed lambda calculus [13, 19]: if a declaration as type arguments $T_1, T_2, ..., T_n$, then its type representation is given as $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$. This will allow at a later stage the type checking of partial function applications. The post-processor scans the arguments list and recursively add the element to a data structure representing an *arrow type*. The arrow type contains two elements corresponding to the elements to the left and right of the arrow. It is possible to build a chain of arrow types by recursively adding an arrow type as right argument of another arrow. For instance, to represent the type $A \rightarrow B \rightarrow C$ we use.

```
Arrow(A,Arrow(B,Arrow(C)))
```

The algorithm to build the type arrow simply scans the argument list and recursively add the current argument to the left of an arrow type and the result of the recursive call to the remaining part of the list as its right part. This process is shown in Algorithm 3.3. Note that in Metacasanova a declaration might contain no type arguments (only the name), thus we return an empty type as placeholder. If a declaration contains only one type argument then there is no need to build an arrow type and the type argument itself is returned. This is also used as a base case for the recursive build of an arrow type.

Algorithm 3.2 Arguments construction in a symbol declaration

function ARGSEPARATION(A list of arguments returned by the parser)
  $name \leftarrow$ ""
  $D \leftarrow \emptyset$
  for all $a \in A$ **do**
    if a is a string **then**
      if $name \neq$ "" **then**
        error: duplicate function name
      else
        $name \leftarrow a$
      end if
    else
      $D \leftarrow D \cup \{a\}$
    end if
  end for
  return $(name, D)$
end function

Algorithm 3.3 Type construction in a symbol declaration

function BUILDDECLARATIONTYPE(A list of arguments returned by the parser)
  if $A = \emptyset$ **then**
    return *Empty*
  else if $A = \{x\}$ **then**
    return x
  else
    $h \leftarrow$ HEAD(A)
    $t \leftarrow$ TAIL(A)
    $r \leftarrow$ BUILDDECLARATIONTYPE(t)
    return $h \rightarrow r$
  end if
end function

Parenthesization of function calls

As explained above, Metacasanova allows the declaration of functions and data types expressed with any notation (prefix, suffix, or infix) and with arbitrary associativity and precedence. Furthermore, this is possible regardless of the amount of arguments the data type or the function call use. Parsing operators according to precedence is a well-known problem that must be solved in order to avoid ambiguity in the language grammar. For instance, the expression `3 + 5 / 4` can generate two different parse trees: `(3 + 5) / 4` or `3 + (5 / 4)`. This ambiguity can be solved by setting the division operator to have a higher precedence over the sum operator, as in traditional arithmetic.

The first attempt to solve this problem was Dijkstra's Shunting-Yard algorithm [26] that takes an expression containing operators and a priority table and returns the same expression in reverse polish notation. This approach was later generalised by operator-precedence parsing which is available in all LALR(1) parser generators. Unfortunately, these approaches deal with binary operators, but are insufficient for operators of arbitrary arity. Moreover, parse generators such as Yacc allows to define a set of pre-defined language operators but do not allow to specify "custom" operators to extend the language with. A notable effort in parsing *mixfix operators* (i.e. operators with an arbitrary position in an expression) using precedence graphs has been done in [22].

In this work we used an AST-transformation technique that changes a function call expressed as a sequence of arguments into a parenthesized version based on defined priorities and associativity. A function call, as defined in Section 3.4.2, can be the left part of a premise or a conclusion, and is parsed as a sequence of arguments. Each argument can be represented in the following way:

- A literal.
- An identifier, that can start with an alphabetic character (*simple id*), or a symbol (such as `%, #, &, @, ...`) followed by a sequence of alphanumeric characters.
- A nested expression, which is any sequence of arguments enclosed between brackets (such as `(5 +++ x)`).

We now give the definition of parenthesization of an argument sequence. In what follows we use the term *symbol* to indicate the name of *functions* or *meta-data* structure used in an argument sequence.

Definition 3.1. An argument sequence is parenthesized if all its arguments are (*i*) literals, (*ii*) identifiers, or (*iii*) a nested expressions containing a parenthesized argument sequence, and the nesting depth of a symbol is directly proportional to its priority (i.e. the highest-precedence operator is at maximum nesting depth).

For instance, given the precedence relation in Table 3.1, the following argument sequence is parenthesized

```
x -> ($ a1 (b1 % b2 b3) a2)
```

Listing 3.1: Example of parenthesization

Symbol	Priority	Left arity	Right arity	Associativity
%	2	1	2	Left
$	1	0	3	Left
->	0	1	1	Left

Table 3.1: Precedence relation for Listing 3.1

The algorithm takes as input a precedence table with the structure of Table 3.1 and the argument sequence to parenthesize, and returns the parenthesized argument sequence. We adopt a recursive approach to the parenthesization problem, whose base case is that the sequence contains only identifiers and literals. Note that a sequence that contains a nested expression is in general not parenthesized because the sequence enclosed in brackets has not been parenthesized yet.

At this point we have two different possibilities for an argument sequence: (*i*) the sequence contains no symbol, or (*ii*) the sequence contains one ore more symbols.

Parenthesization of sequences with no symbols When there are no symbols in the sequence of arguments, each argument can be a literal or identifier corresponding to no definition, or a nested expression. In the first case the argument is automatically parenthesized according to Definition 3.1. In the second case we must recursively parenthesize the sequence contained in the nested expression. If the result of this parenthesization is a single nested expression, than we take its content and store it into a single nested expression, to avoid redundant nesting parentheses of the form `((...(a1 a2, ..., an)...))`. If the result is a series of arguments than we just place it inside a nested expression, obtaining something of the form `(a1 a2 (b1 b2 ...) a3 (c1 c2 ...) ...)`. It might be possible that the recursive algorithm tries to parenthesize a sequence containing no arguments (see the details of the algorithm below). In this case the algorithm simply returns an empty sequence.

Parenthesization of sequences with symbols Dealing with sequences containing symbols is more complex, since we must parenthesize them keeping into account the symbols priorities and associativity. According to Definition 3.1, in a parenthesized sequence the operator with the lowest priority must be at the top of the sequence nesting. The idea is then that at the current depth we should find the symbol with the lowest priority and a series of parenthesizations containing sequences with symbols of higher priority. The algorithm extracts the symbol in the sequence with the lowest priority and the positions in the sequence where it appears. Note that the same symbol might appear more than once. If the symbol associativity is left then we consider its rightmost occurrence in the sequence, otherwise its leftmost one. Now we split the sequence in two parts using this occurrence as separator and we recursively try to parenthesize these two parts. For instance, let us consider again the Precedence Table 3.1 and the following sequence of arguments

```
x -> $ a1 b1 % b2 b3 a2
```

The algorithm will select the symbol `->` as a separator, as it is the symbol with the lowest priority (there is only one occurrence, thus we neglect the part that selects the appropriate occurrence). It will then recursively parenthesize the sequences `x` and `$ a1 b1 % b2 b3 a2`. At this point we have two parenthesizations of the left and right part: $\{l_1, l_2, ..., l_n\}$ and $\{r_1, r_2, ..., r_m\}$ respectively for the left and right sequence. Assuming that the left arity of the current symbol is a_l and the right one is a_r, then the parenthesization of the current operator will contain the elements $\{l_{n-a_l+1}, ..., l_n\}$ and $\{r_1, ..., r_{a_r}\}$. Assuming that the current symbol is σ, we now consider three cases:

1. The symbol uses a prefix notation: in this case the parenthesization will not contain any elements from $\{l_{n-a_l+1}, ..., l_n\}$, thus the final parenthesization will be $\{l_1, l_2, ..., l_n\ (\sigma, r_1, ..., r_{a_r})\ r_{a_r+1}, ..., r_m\}$.

2. The symbol uses an infix notation: in this case the parenthesization will contain elements from both $\{l_{n-a_l+1}, ..., l_n\}$ and $\{r_1, ..., r_{a_r}\}$. The final parenthesization will be

$$l_1, ..., l_{n-a_l}\ (l_{n-a_l+1}, ..., l_n, \sigma, r_1, ..., r_{a_r})\ r_{a_r+1}, ..., r_m$$

3. The symbol uses a suffix notation: in this case the parenthesization will not contain any elements from $\{r_1, ..., r_{a_r}\}$ and the final parenthesization will be $l_1, ..., l_{n-a_l}\ (l_{n-a_l+1}, ..., l_n, \sigma)\ r_1, ..., r_m$.

In order to clarify this process, let us consider again the argument sequence

```
x -> $ a1 b1 % b2 b3 a2
```

and let us apply the algorithm to it (again using the Priority Table 3.1). The algorithm will test the whole sequence looking for symbols, and of course will find one. We then fall in the second part of the algorithm. The symbol with the lowest priority is `->`, so the algorithm will recursively parenthesize `x` and `$ a1 b1 % b2 b3 a2`. The left one is a base case of the recursion since the sequence contains only one identifier that is not a symbol, thus its parenthesization is the sequence itself. The right one contains other symbols so we have to recursively apply the algorithm. The operator with the lowest priority is now `$`. The symbol is the first element of the sequence, thus the left subsequence obtained by the partitioning phase is empty (and the result of its parenthesization an empty sequence as well). The right subsequence is `a1 b1 % b2 b3 a2`. In this sequence there is only one symbol, which is `%`, thus the algorithm will try to parenthesize `a1 b1` and `b2 b3 a2`. Their parenthesization is trivial and returns the sequences themselves. At this point we have to consider the arity of `%`, which accepts one left argument and two right arguments. The algorithm will then enclose between brackets `b1 % b2 b3`. The full parenthesization leads then to `a1 (b1 % b2 b3) a2`. At this point this result is used to build the parenthesization of `$`. This symbol accepts 3 right arguments (and no left argument), so the parenthesization will be `($ a1 (b1 % b2 b3) a2)` . Finally we have to use the result to build the parenthesization of `%`. This symbol accepts 1 left argument and 1 right argument. The parenthesization of its left subsequence

was `x`, while its right parenthesization is `($ a1 (b1 % b2 b3) a2)` , thus the final parenthesization will be `(x -> ($ a1 (b1 % b2 b3) a2))`. At this point, the outer parenthesization can be removed for better readability. The details of the algorithm are shown in Algorithm 3.4.

3.5 Type checking

The type checker of Metacasanova is responsible of two tasks: (*i*) checking that the declarations are correctly formed and (*ii*) check that the premises and conclusions of rules respect the meta-types defined in the declarations. We will now proceed to explain in detail how the two processes are implemented in the metacompiler.

3.5.1 Checking declarations

Checking declarations requires to check the consistency of *meta-type declarations* in each one of the function or data declarations. Note that, from now on, we will refer to meta-types simply as *types* for simplicity. Also we assume that, at this point, we have already built the *symbol table* for the meta-program containing all the symbol declarations with the complete information about the declaration itself. A type declaration in Metacasanova can have four different forms:

- *Zero:* A place-holder type used for function or meta-data that do not use any arguments.
- *External:* Used for embedded types from .NET.
- *Unsafe:* Unsafe type is a place-holder for external function calls, i.e. function defined in an external embedded language.
- *Argument:* A type argument is a simple type identifier followed by an optional list of generic arguments used for generic types. For example the type `Tuple[a,b]` is a type argument whose identifier is `Tuple` and whose generic arguments are *a* and *b*.
- *Arrow:* An arrow type has the form `T1 -> T2 -> ... -> Tn` and is used to represent the type of the arguments used when calling a function or when constructing a meta-data. For example the function declaration `Func Num -> "+" -> Num : Num` has the *Arrow* type `Num -> Num`. Note that the symbol declaration, for convenience, stores two different type declarations: the arguments types separated from the function returned type or the meta-data type and a *full type* that combines the type of the arguments and the returned type or data type into a single arrow. For example, for the function above, its full type would be `Num -> Num -> Num`.

The algorithm to check the type declarations behaves differently depending on the form of the type declaration. For *Zero* or *External* type declarations the check always succeeds. For *Arrow* the algorithm recursively check the left and right part of the arrow. In the case of an *Argument* we have two sub-cases: (*i*) the argument is an identifier with no generic arguments,

Algorithm 3.4 Parenthesization of a sequence of arguments. The operators :: and @ are respectively prepend and append on a list. With the notation $\langle S \rangle$ we denote a sequence S enclosed by parentheses.

```
function PARENTHESIZE(S symbols in the sequence, A arguments sequence)
    if A = ∅ then
        return A
    else
        if s ≠ ∅ then
            let s′ be the symbol with lowest priority in S
            let I be the set of indices at which s′ occurs in S.
            l ← ∅
            r ← ∅
            if s′ is left-associative then
                I′ ← LAST(I)
                l, r ← SPLITAT(I′)
                r ← TAIL(r)
            else
                I′ ← FIRST(I)
                l, r ← SPLITAT(I′)
                r ← TAIL(r)
            end if
            let lsym be the symbols in l
            let rsym be the symbols in r
            lpar ← PARENTHESIZE(lsym,l)
            rpar ← PARENTHESIZE(rsym,r)
            let larity be left arity of s′
            let rarity be right arity of s′
            largs, plargs ← SPLITAT(|largs| − larity)
            rargs, prargs ← SPLITAT(rarity)
            if larity + rarity > 0 then
                if s′ is prefix then
                    e ← s′ :: prargs
                else if s′ is suffix then
                    e ← plargs @ s′
                elses′ is infix
                    e ← plargs @ s′ @ prargs
                end if
                return largs @ ⟨e⟩ @ rargs
            else
                return largs @ s′ @ rargs
            end if
        else
            p ← ∅
            for all a ∈ A do
                if a = ⟨e⟩ then
                    let S′ be symbols in e
                    p′ ← PARENTHESIZE(S′) e
                    if p′ = ⟨e′⟩ then
                        p ← p @ e′
                    else
                        p ← p @ ⟨p′⟩
                    end if
                else
                    p ← p @ {a}
                end if
            end for
        end if
    end if
end function
```

or (*ii*) the argument is an identifier followed by a number of generic arguments. The first case is simple, as it is enough to check whether the type is defined in the symbol table or not. If the type cannot be found in the symbol table then it is undefined and an error is returned. In the case of a type with generic arguments we must check that the amount of provided generics matches the amount required for the generic type; then we must check if the provided generic arguments have a correct form. A generic argument can be an identifier or again a type accepting other generic arguments. This is the case, for instance, of a type declaration such as `Tuple[List[int],Tuple[a,List[float]]]`. In the case of a simple generic identifier we must only check that the generic identifier is in the scope of the declaration. For instance the declaration

```
Func[a,b] "foo" -> a -> b : b
```

would be a valid declaration since the generic identifier are in the scope of the declaration, while

```
Func[a,b] "foo" -> a -> b : c
```

would be invalid if `c` is not a data type defined in the meta-program. In the case of nested types the algorithm must recursively check again that the type exists and that the generic arguments are valid. The procedure details are described in Algorithm 3.5.

Algorithm 3.5 Type checking of a symbol declaration

```
function checkDeclaration(S symbol table, G declared generic arguments, d type declaration)
    if d = Empty or d = Zero then
        return
    else
        let G' be the generics required for d
        if G' ≠ ∅ then
            if |G'| ≠ |G| then
                error: invalid amount of generic arguments
            else
                for all g ∈ G' do
                    checkDeclaration(S,G,g)
                end for
            end if
        else
            if d ∈ S or d ∈ G then
                return
            else
                error: undefined type
            end if
        end if
    end if
end function
```

3.5.2 Checking rules

Type checking a rule requires to type check its conclusion and all of the premises. In what follows we assume that the parser post-processor has already parenthesized and normalized all the function calls, so that every function call is parenthesized according to symbol priority and associativity and that the symbol name is in the first position of an argument sequence. Moreover, we use the following definition relative to meta-data arguments:

Definition 3.2. An argument is said to be an *explicit data argument* when it is an expression constructing a meta-data.

For example, in the following meta-program

```
Data Expr -> "+" -> Expr : Expr
Func "eval" -> Expr -> Environment : Value
...

eval a env -> a'
eval b env -> b'
<<a' + b'>> -> v
------------------
eval (a + b) env -> v
```

Listing 3.2: Example of an explicit data argument in Metacasanova

the first argument of `eval` in the conclusion is an explicit data argument. Moreover we use the following definition to define the compatibility of two types.

Definition 3.3. Let T be the set of types defined in a meta-program and $E = \{(t_i, t_j) \mid t1 \in T \wedge t2 \in T\}$ the set of type equivalence declarations, then we say that the type t_1 is *compatible* with t_2 if either $t_1 = t_2$ or $\exists (t_i, t_j) \in E \mid t_1 = t_i \wedge t_2 = t_j$.

Checking the conclusion

A conclusion must always contain a function call. In order to type-check the function call correctly, we must check that (*i*) the arity of the function is respected, i.e. that the arguments are not more than what the function expects (the type system of Metacasanova supports partial function application, so it is allowed to pass less arguments), and (*ii*) that the type of each argument is compatible with what provided in the declaration. Moreover, since a conclusion might contain explicit data arguments, we have to recursively add all the variables contained in the explicit data arguments to the local variables of the current function, as they could be used in the premises. As an example, refer to Listing 3.2, where the variables `a` and `b` are defined in the argument of `eval` and later used in the premises. Checking *i* is trivial: it is sufficient to compare the length of the given argument sequence with the length of the arguments provided in the declaration. If the length of the argument sequence is greater than the arguments defined in the declaration then an error is returned. Checking (*ii*) is more complex and the details are explained below.

The right hand-side of a conclusion might contain a variable or an explicit data argument. In both cases its type checking must be delayed until all premises are processed because a variable appearing in one of them might

be used. For example, in Listing 3.2, variable `v` is used in the right hand-side of the conclusion and defined in the result of the last premise.

Checking a premise

A premise, as explained in Section 3.4.2, can be (*i*) a function call, (*ii*) a binding, or (*iii*) a clause.

In case *i* we have to type check the arguments of the function call in the same fashion of the conclusion but with a slight difference: when encountering a variable this must not be added to the local variable set but rather looked up in it. If the lookup fails then the variable is undefined and cannot be used. The same happens when checking the arguments of explicit data arguments. If the call is correctly typed, then we must check its result. The result of a call can be either a variable or an explicit data argument. In the first case the variable is added to the local variables and its type set to the return type of the function. In the case of an explicit data argument then all the arguments that are variable are added to the local variables with the appropriate type read from the meta-data structure declaration and, in case of a nested explicit data argument, the procedure is recursively applied.

In case *ii* the binding is correctly typed if its right argument is correctly type. The right argument can be a variable or an explicit data argument so we apply the same method that we used to check variables and explicit data arguments in function calls. If this test succeeds then the left argument of the binding, which is always a variable, is added to the local variables with the type of the right argument.

In case *iii* the clause is correctly typed if the types of the arguments used in the comparison operator are compatible with respect to the operator itself and if they are themselves correctly typed. For example, for the equality comparison, we must ensure that both arguments are correctly typed. As always, if external types are involved the test automatically succeeds because nothing can be known at this point about the compatibility of the provided types. Of course the test fails if we use a comparison operator combining external types with types defined in the meta-program, because they will always be incompatible.

Checking a single argument

When checking the type of an argument we have to consider several cases depending on what kind of argument we are inspecting. The reader can find the details of each case below. Note that we will never consider external types as their type checking is delegated to the external code compiler that is used when compile the generated code, so their type checking in the Metacasanova type checker is always successful.

Checking literals If the argument is a literal then we have to consider three sub-cases:

1. The expected type is generic. In this case, since we are providing

explicitly a literal, the generic can be assigned the specific type of the literal.

2. The expected type is non-generic. In this case we have to check if the type of the literal is compatible with the expected type. This only happens if the expected type is one of the native types supported by Metacasanova.

3. The expected type is a meta-data structure requiring generic arguments. In this case the type is always incorrect, since a literal is always incompatible with meta-data.

Checking identifiers When we have an identifier as argument, this might be either a symbol for a meta-data structure taking no arguments, or simply a variable. In the first case we simply check that the type of the meta-data structure is compatible with the expected type. In the second case the result depends on whether we are type checking a conclusion or a premise. If we are checking a conclusion, then the variable must be added to the local variables in the scope of the rule, otherwise we must look up the local variables to check if it was previously defined. If the lookup fails then the variable is undefined and an error is returned.

Checking nested expressions Checking a nested expression requires, in the first place, to recursively type check the arguments used in the nested expression. If this check succeeds than we must also ensure that the type of the meta-data structure that we are analysing is compatible with the expected type.

Checking the type compatibility According to Definition 3.3, a type is compatible with another if they are equal or if, in the symbol table, there exists a specified equivalence between the first type and the second. We have thus to distinguish two cases: (*i*) check if the types are equal and, if this fails, (*ii*) check if the provided type is paired with the expected type in an equivalence table. For the following considerations refer to the type structure defined in Section 3.5.1.

Testing type equality Checking type must consider three options: (*i*) the type is a simple identifier, (*ii*) the type is an Arrow type , and (*iii*) the type is External, Unsafe or Zero.

In case i we must compare two Type Arguments t_1 and t_2 other wise the test fails immediately. In this case it is enough to simply check that $t_1 = t_2$.

In case ii we must compare two Arrow types otherwise the test fails immediately. Let us assume that we have $T ::= t_1 \to t_2 \to ... \to t_n$ and $U ::= u_1 \to u_2 \to ... \to u_m$, then we check if $t_1 = t_2$ and recursively apply the type equality test on $t_2 \to ... \to t_n$ and $u_m \to ... \to u_m$. Note that if $n \neq m$ at some point the test will fail because we will compare a Type Argument with an Arrow Type, which will always fail.

In case *iii* the test succeeds if one of the types (either the provided one or the expected one) is external or unsafe; it also succeeds if both types are Zero types.

Testing type compatibility Type equivalence might succeed in only two cases: (*i*) when testing two Argument Types, and when testing two Arrow Types; in all other cases the test fails immediately. In what follows we write $t_1 \equiv t_2$ to say that t_1 is compatible with t_2.

In case *i* we have to check if the provided type is paired with the expected type in a type equivalence map stored in the symbol table. If the lookup in the map is unsuccessful then an error is returned.

In case *ii* again we consider $T ::= t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n$ and $U ::= u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_m$. This time we check if $t_1 equiv t_2$ and then we recursively check the equivalence of $t_2 \rightarrow ... \rightarrow t_n$ with $u_m \rightarrow ... \rightarrow u_m$. Again, the equivalence test fails if $n \neq m$ because we will end up comparing an Argument Type with an Arrow Type.

3.6 Code generation

Metacasanova uses C# as target language and generates code compatible with any library compiled in the .NET framework. In this phase we must (*i*) generate the appropriate abstractions in C# to represent meta-data structures and their type equivalence, and (*ii*) generate the code to implement the semantics of the rule evaluation, as described in Section 3.2.3.

3.6.1 Meta-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;

  5
  public override string ToString()
  {
```

```
  return "(" + __name + " " + __arg0 + ")";
  }
}
```

3.6.2 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 meta-program.

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 3.6.2. 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

> A language that doesn't affect the way you think about programming is not worth knowing.
>
> Alan J. Perlis

In this chapter we show how to implement languages in Metacasanova. The first language that we implement is a small imperative language called C--. Although tiny, this language contains many common features typical of imperative languages such as control structures, program states, variable scoping, and type annotations. We then proceed to re-implement the semantics of Casanova, a DSL for game development, in Metacasanova, work shown also in [23]. We finally evaluate the length of the language implementation in Metacasanova against a hard-coded implementation of the same language in a general-purpose programming language, and the runtime performance of programs written in the meta-compiled version against Python.

4.1 The C-- language

In this section we present the implementation of a small imperative language called C--. Note that, although the name might suggest this, we do not claim any resemblance with the C programming language, as it lacks several features such as pointer arithmetic, arrays, and functions.

C-- allows the use of four built-in values: integer, strings, booleans, and floating-point numbers in double precision. The memory is represented using a dictionary that pairs variable names with their value. In what follows we omit the details of the lookup of entries in the dictionary for brevity. Suffice to say that the meta-program makes use of the `ImmutableDictionary` data structure available in .NET. Also note that C-- defines scopes for variables, so that if a variable is declared inside the scope of a code block in a control structure, that is usable only within the scope itself.

The core of the meta-program is made of the evaluation of both expressions and statements. We proceed below to present the details of both kinds of evaluation.

4.1.1 Expression Semantics

As explained above C-- supports boolean, string, integer, and floating-point values. These are represented through the following meta-data structures in the meta-program.

```
Data "$i" -> <<int>> : Value Priority 300
Data "$d" -> <<double>> : Value Priority 300
Data "$s" -> <<string>> : Value Priority 300
Data "$b" -> <<bool>> : Value Priority 300
```

Note that we are using the .NET data types to represent the actual values stored in the meta-data structures. We also define the following type equivalence, since values are atomic cases of expressions and can be used as such:

```
Value is Expr
```

Expressions can also contain variables, thus we need a meta-data structure to represent them as well.

```
Data "$" -> <<string>> : Id Priority 300
```

Variables can be used as atomic expressions as well, so we need an additional type equivalence

```
Id is Expr
```

We now define a data structure to represent the state of the program. The state is simply a map where the key is a variable name and the stored element a valid value in C--. In the declaration we will define the meta-type `SymbolTable`, and from now on we will refer use the term "symbol table" as a synonym of "state".

```
Data "$m" << ImmutableDictionary<Id, Value> >> : SymbolTable
```

Since we want to allow variable scoping, the state of the program is not represented by a single map, but by a list of maps. Each time the program enters a different scope context, an empty map is added to this list, and removed when the program exits the scope. This process will be further explained below. We define a meta-data structure to represent this list of states (note that the operator for the construction of the list is infix).

```
Data SymbolTable -> "::" -> TableList : TableList
Data "[]" -> TableList
```

We can finally proceed to define a meta-data structure to represent the operations for expressions. First we define the arithmetic operators in the language:

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

then we can define operators for boolean expressions:

```
Data Expr -> "&&" -> Expr : Expr
Data Expr -> "||" -> Expr : Expr
Data "!" -> Expr : Expr
```

and finally comparison operators:

```
Data Expr -> "equals" -> Expr : Expr
Data Expr -> "neq" -> Expr: Expr
Data Expr -> "ls" -> Expr : Expr
Data Expr -> "leq" -> Expr : Expr
Data Expr -> "grt" -> Expr : Expr
Data Expr -> "geq" -> Expr : Expr
```

We now have to define the function that evaluates an expression through rules in the program. This function takes as input the list of symbol tables (needed to read possible variables), an expression, and returns the value after computing the expression.

```
Func "evalExpr" -> TableList -> Expr : Value
```

Now we have to proceed to define the rules to compute the actual evaluation of an expression. Clearly the base cases of the evaluation are the atomic values, where we immediately return the value itself.

```
-----------------------------
evalExpr tables ($i v) -> ($i v)

-----------------------------
evalExpr tables ($d v) -> ($d v)

-----------------------------
evalExpr tables ($s v) -> ($s v)

-----------------------------
evalExpr tables ($b v) -> ($b v)
```

Evaluating variables is more complex: we have to look at the table currently in the head of the list of tables (which is the one relative to the current scope). If we do not find the required variable we have to recursively look it up in the tail of the list, since we could have an arbitrary amount of nested scopes. When the variable is found we return its value. This behaviour is implemented by the following code:

```
symbols contains ($name) -> Yes
symbols lookup ($name) -> val
-----------------------------------------
evalExpr (symbols :: tables) ($name) -> val

symbols contains ($name) -> No
evalExpr tables ($name) -> val
-----------------------------------------
evalExpr (symbols :: tables) ($name) -> val
```

We proceed now to define the evaluation of arithmetic operators. We show only the example of the sum for brevity, the other rules differ only in the operator. Evaluating the arithmetic expression requires to recursively call `evalExpr` on the right and left argument. These recursive calls will eventually return two values that are the result of the two evaluations. After we obtain these values, we can compute their sum and return it as result.

```
evalExpr tables expr1 -> ($i val1)
evalExpr tables expr2 -> ($i val2)
<<val1 + val2>> -> v
---------------------------------------
evalExpr tables expr1 + expr2 -> ($i v)
```

Note that we have used .NET external code in the third premise to compute the result of the arithmetic operation. Evaluating arithmetic operations involving floating-point expressions can be done in an analogous way, except in the premises we expect to have the meta-data structure for floating-point values as result of `evalExpr`:

```
evalExpr tables expr1 -> ($d val1)
evalExpr tables expr2 -> ($d val2)
<<val1 + val2>> -> v
---------------------------------------
evalExpr tables expr1 + expr2 -> ($d v)
```

The same can be said for the string concatenation. The evaluation of boolean expression is analogous: we show again only the evaluation for AND as the other rules are analogous:

```
evalExpr tables expr1 -> ($b val1)
evalExpr tables expr2 -> ($b val2)
<<val1 && val2>> -> b
-------------------------------------------------------------------------
evalExpr tables expr1 && expr2 -> b
```

Again we rely on external code to compute the actual boolean value. As for the comparison operators, we can use a clause in the premise to avoid using external code in the following way:

```
evalEpxr tables expr1 -> val1
evalExpr tables expr2 -> val2
val1 == val2
-------------------------------------------------
evalExpr tables (expr1 equals expr2) -> $b true

evalEpxr tables expr1 -> val1
evalExpr tables expr2 -> val2
val1 != val2
-------------------------------------------------
evalExpr tables (expr1 equals expr2) -> $b false
```

The first rule checks that the values computed by evaluating the left and right argument of the equality comparison are the same. If this happens then the rule returns a meta-data structure containing the boolean representation of `true`. Otherwise the first rule fails and the second one is executed. This one will return a boolean representation of `false` when the values are different.

For inequality operators we must rely on external code for the computation is Metacasanova only allows equality comparisons in clauses:

```
evalExpr tables expr1 -> ($i val1)
evalExpr tables expr2 -> ($i val2)
<< val1 < val2 >> -> boolResult
-----------------------------------------------------------
evalExpr tables (expr1 ls expr2) -> ($b boolResult)
```

The evaluation of the other comparison operators is implemented through analogous rules, which differ only in the operators.

4.1.2 Statement Semantics

Statement evaluation requires the definition of a different function, `eval`, that processes each statement and returns the result of the statement evaluation and the updated state. Note that, even if the evaluation of statements does not always change the state, in general we have to assume that this will happen.

The function `eval` takes as input a statement to process and the current state (list of symbol tables), and returns the updated list of symbol tables

```
Func "eval" -> TableList -> Stmt : TableList
```

We now proceed to define the meta-data structures necessary to represent the statements of the language: C-- supports (*i*) variable declarations, (*ii*) variable assignment, (*iii*) if-then-else, (*iv*) while loops, and (*v*) for loops.

Variable declarations follow the same structure of standard C, that is a type name followed by an identifier. Thus, the corresponding meta-data structure can be defined as:

```
"variable" -> Type -> Id : Stmt
```

Analogously, variable assignment follows the C convention and uses the `=` symbol.

```
Id -> "=" -> Expr : Stmt
```

The control structure if-then-else does not follow the standard C representation, rather we use the keywords `then` and `else` to delimit its code blocks. Note that nothing would prevent to implement the conventional C syntax, but we prefer this "lightweight" representation. The keywords `then` and `else` are meta-data structures that take no arguments and do not have any functional utility other than syntactical mark-ups.

```
Data "then" : Then
Data "else" : Else
Data "if" -> Expr -> Then -> Stmt -> Else -> Stmt : Stmt
```

Analogously we can define the meta-data structure for `While` and `For`

```
Data "do" : Do
Data "while" -> Expr -> Do -> Stmt : Stmt
Data "for" -> Expr -> Expr -> Epxr -> Do -> Stmt : Stmt
```

Up to this point we are able to define single statements in the language, but we need a way to concatenate a sequence of statements to form code blocks, in the fashion of C. This is done by introducing an additional meta-data structure, which is the ";" symbol. For convenience, we also introduce a `nop` statement, which does not do anything, but it will be useful to express the semantics of statements evaluation.

```
Data Stmt -> ";" -> StmtList : StmtList
Data "nop" -> :Stmt

StmtList is Stmt
```

We now proceed to define the semantics of statement evaluation.

Evaluating a sequence of statement

The evaluation of a sequence of statements require to evaluate the first statement in a sequence and then recursively evaluate the rest of the sequence. The recursive evaluation returns the final program state. The base case of the recursion is met when the sequence contains only `nop`. In this case we terminate the evaluation and return the unchanged program state.

```
-------------------------
eval tables nop -> tables

eval tables a -> tables'
eval table' b -> res
-------------------------
eval tables (a;b) -> res
```

Variable Declarations and Assignments

Evaluating a variable declaration simply adds the variable to the symbol table of the current scope. Note that we allow variable shadowing, so it is possible to redefine the same symbol in different scopes.

```
symbols defineVariable id -> symbols'
-----------------------------------
eval (symbols nextTable tables) (variable t id) -> symbols' nextTable
     tables
```

This rule is executed whenever the processed statement matches the structure of a variable declaration statement. The premise adds the symbol to the symbol table of the current scope (we omit the details for brevity), and returns an updated symbol table. The list of symbol tables is rebuilt to include the updated table and returned as result.

Variable assignment is more complex, since the variable we are trying to use might not be in the symbol table of the current scope. We must then define two lookups functions, that behave differently depending on whether the variable is in the symbol table in the head of the symbol table list or not. We declare the function `updateTable` that performs this lookup and updates the table list accordingly.

```
Func "updateTables" -> TableList -> TableList -> Id -> Expr :
     EvaluationResult
```

In the case that the variable is in the symbol table in the head of the list of tables we have the following rule:

```
symbols contains id -> Yes
evalExpr vars expr -> val
symbols add id val -> symbols'
------------------------------
updateTables vars (symbols :: tables) id expr -> symbols' :: tables
```

The first premise checks if the symbol is contained in the table in the head of the list. If the answer is `Yes` (a meta-data structure returned by the function `contains`, not described here again for brevity), then the second premise proceeds to evaluate the expression in the right hand-side of the assignment. The third premise adds the value obtained as result of the

second premise to the current symbol table and returns the modified table. The new table is than placed in the head of the table list and the whole list is returned. Note that, at this point, all the tables in the list remain unchanged except the one that was in the head. Note that `updateTables` carries two copies of the list of tables. One of them is passed to `eval` because the right-hand side of the assignment might contain other variables. The process of looking up the left hand-side variable pops symbol tables from the head of list (see next rule) but the original list of tables is necessary when assigning the values of variables located in inner scopes. For instance, consider the following program in C--:

```
int x;
...
if (x > 0) then
  int y;
  y = 4
  x = y;
else
  x = x - 1;
```

Listing 4.1: C-- sample program

assume that before the if-then-else $x > 0$. The program will enter the `then` block and declare `y`. In the current state we have to symbol tables, one for the scope of the `if-then-else` and one for the outer scope. When assigning `y` to `x` the symbol table tries to look up `x` in the table of the current scope and fails. This will pop the head of the list of tables (which is the table of `if-then-else`) and recursively look in the tail. During the second attempt `x` is retrieved but now we do not have the symbol table where `y` is defined anymore to evaluate the right hand-side. We thus need the original list of tables to be able to retrieve `y`. In general, if we call d_l the depth of scoping of the left hand-side and d_r the depth of scoping of the right hand-side, the process pops the table of the right hand-side whenever $d_l > d_r$ and this is when we need the original list of tables to retrieve the value of the right hand-side.

If the variable is not contained in the head of the list, i.e. it has not been declared in the current scope, we have the following rule:

```
symbols contains id -> No
updateTables vars tables id expr -> tables'
-------------------------------------------
updateTables vars (symbols :: tables) id expr -> symbols :: tables'
```

The first premise tries to lookup the variable in the symbol table of the current scope and does not find it. Thus we recursively call `updateTables` with the tail of the list. The recursive call will eventually find the variable in one of the symbol tables associated with outer scopes. This process will produce an updated list of tables that is returned as a new tail for the current list.

At this point, the rule for the evaluation of the variable assignment simply class the `updateTables` function in its premise:

```
updateTables tables tables id expr -> res
-----------------------------------------
eval tables (id = expr) -> res
```

Variable	Value
x	undefined

Variable	Value
y	4

Variable	Value
x	4

Table 4.1: Symbol table at the beginning and after the execution of the program in Listing 4.1 with $x > 0$

Conditionals

Evaluating `if-then-else` requires two rules, depending on the result of the evaluation of its condition. The following rule implements the semantics when the condition is false:

```
evalExpr tables condition -> $b true
emptyDictionary -> table
eval (table :: tables) thenBlock -> table' :: tables''
-----------------------------------------------------------------
eval tables (if condition then thenBlock else elseBlock) -> tables''
```

The first premise evaluates the condition of the control structures and succeeds if the result is a meta-data structure containing the boolean value `true`. The second premise uses an utility function to initialize an empty symbol table. This is required to define a new table for the scope of the conditional. The third premise evaluates the statements contained in the then block after pushing the symbol table for the current scope in the list of symbol tables. This process will eventually produce a new list of symbol tables. The result returns only the tail of this list, since when we exit the scope of the conditional we must pop its symbol tables.

For instance, consider again the program in Listing 4.1 and again assume that $x > 0$. After executing the then block, the state of the program is made of the symbol tables shown in Table 4.1
After exiting the then block, variable `y` exits the scope, thus we have to pop the symbol table for the current scope. However, the symbol table of the outer scope has been changed because `x` got the value of `y`. Thus the evaluation returns the list containing this updated table. In general, the process should consider that an arbitrary amount of symbol tables for each outer scope have been changed, thus we return this updated list.

The rule that evaluates conditionals when the condition is false is analogous, except this time we evaluate the else block:

```
evalExpr tables condition -> $b false
emptyDictionary -> table
eval (table :: tables) elseBlock -> table' :: tables''
-----------------------------------------------------------------
eval tables (if condition then thenBlock else elseBlock) -> tables''
```

While Loops

Evaluating the while loops require to check its condition first. When the condition is false we simply skip the loop without changing the state. The rule to implement this behaviour is thus straightforward:

```
evalExpr tables condition -> $b false
-----------------------------------------
eval tables (while condition do block) -> tables
```

The semantics when the condition is true is more complex:

```
evalExpr tables condition -> $b true
emptyDictionary -> table
eval (table :: tables) block -> table' :: tables''
eval tables'' (while condition do block) -> res
-------------------------------------------------
eval tables (while condition do block) -> res
```

The first premise succeeds when the evaluation of the condition returns a true boolean value in C--. Analogously to what we did for conditionals, we initialize an empty symbol table for the current scope and we push it into the list of symbol tables. We then evaluate the body of the loop. This process will, in general, produce an updated list of symbol tables. Again we pop the symbol table for the current scope because we are exiting the loop. We then evaluate again the whole loop to test its condition again.

For Loops

For loops follow the C convention and are made of four parts: (*i*) an initialization, (*ii*) a condition (*ii*), (*iii*) a step, and (*iv*) a block of code. The initialization is evaluated once before entering the loop, the condition is tested before each iteration of the loop, and the step is evaluated at the end of each iteration. In order to implement this behaviour we make use of an additional support function called `loopFor`:

```
Func "loopFor" -> TableList -> Expr -> Stmt -> Stmt : TableList
```

The evaluation of the `for-loop` evaluates the initialization in its premise. It then calls `loopFor` after the initialization has been evaluated. Again the initialization might define additional variables that enter the scope of the loop, so the updated table of the current scope is pushed into the list of symbol tables. Note that possible variables defined in the initialization part of the loop might be used after the loop itself, according to the semantics of C, so we have to insert them into the symbol table of the current scope and not the one of the loop itself.

```
eval tables init => tables'
loopFor tables' condition step block => res
-----------------------------------------------------
eval tables (for init condition step do block) => res
```

The rules for `loopFor` are two, since we must consider the case when the condition is false and the one where it is true. When the condition is false the loop is completely skipped, thus we simply return the current state without any changes, in the same fashion of the `while-loop`:

```
evalExpr tables condition -> $b false
-------------------------------------------------------------------
loopFor tables condition step block -> tables
```

When the condition is true, we create as usual a symbol table for the scope of the loop and we push it into the list of symbol tables. The third premise evaluates the block of the loop returning an updated list of tables. As usual we pop the table of the scope of the loop and we evaluate the step. This again might change the list of symbol tables. We then run again the loop with the updated list of tables.

```
evalExpr tables condition -> $b true
emptyDictionary -> table
eval (table :: tables) block -> table' :: tables''
eval tables'' step -> tables3
loopFor tables3 condition step expr -> res
-----------------------------------------------
loopFor tables condition step block -> res
```

4.1.3 Type Checker

Type checking can be performed by using a representation of the type system of C-- in terms of rules, in the same fashion of the semantics. In this section we explain the details of how each language construct is type-checked according to its type rule. We begin by defining an alternative version of the symbol table defined in Section 4.1.1 that contains a mapping between variable names and types:

```
Data "$m" << ImmutableDictionary<Id, Type> >> : TypeTable
```

and a constructor for the meta-data representing a sequence of type tables.

```
Data TypeTable -> "::" -> TypeTableList : TypeTableList
Data "[]" : TypeTable
```

We now start by defining the meta-data structures for the types in C--:

```
Data "t_int" : Type
Data "t_double" : Type
Data "t_string" : Type
Data "t_bool" : Type
Data "t_unit" : Type
```

We also defined a special meta-data representing a type error to correctly report errors if the program contains invalid types:

```
Data "error" -> <<string>> : Type
```

Typing expressions

We now proceed to define the type rules for expressions. We initially need to define a function to use in the conclusion of a type rule that is able to evaluate type of an expression:

```
Func "typeExpr" -> TypeTableList -> Expr : Type
```

The axioms of expression typing are those that return the type of a literal. In this case the rule immediately returns the type associated to the specific literal.

```
-----------------------------
typeExpr tables ($i v) -> t_int

-----------------------------
typeExpr tables ($d v) -> t_double

-----------------------------
typeExpr tables ($s v) -> t_string

-----------------------------
typeExpr tables ($b v) -> t_bool
```

Type checking variables require to perform a lookup for the variable name in the list of type tables that we carry along during the typing process. The variable could be in the table associated with the current scope or in the table of an outer scope. Therefore, we start by first looking in the table of the current scope, and if we do not find the variable we recursively look it up in the subsequent table. If we traverse the whole list of tables without finding the variable, then it means that the program contains an undefined variable and an appropriate error notifying the problem should be returned.

```
-------------------------------------
typeExpr [] ($ name) -> error <<"Undefined variable:" + name>>

types contains ($ name) -> Yes
types lookup ($ name) -> varType
-----------------------------------------------
typeExpr (types :: tables) ($ name) -> varType

types contains ($ name) -> No
typeExpr tables ($ name) -> error msg
-----------------------------------------
typeExpr (types :: tables) ($ name) -> error msg

types contains ($ name) -> No
typeExpr tables ($ name) -> varType
-----------------------------------------
typeExpr (types :: tables) ($ name) -> varType
```

Note that we had to include a rule in whose premise we check whether the recursive lookup returned an error. If this is the case the entire rule returns the error message rather than the type of the variable.

Type-checking expression operators require to perform the following steps:

1. Type-check the left and right argument.
2. Check that the types obtained at the previous step are compatible with the operator definition.
3. Return the type of the operator.

The process fails when the type-checking of one of the two expressions fails or when the types are incompatible with the operator definition.
For brevity we only present the case of the sum, the rules for the other operators are analogous:

```
typeExpr tables expr1 -> error msg
----------------------------------
typeExpr tables expr1 + expr2 -> error msg

typeExpr tables expr2 -> error msg
----------------------------------
```

```
typeExpr tables expr1 + expr2 -> error msg

typeExpr tables expr1 -> t_int
typeExpr tables expr2 -> t_int
-------------------------------------
typeExpr tables expr1 + expr2 -> t_int

typeExpr tables expr1 -> t_double
typeExpr tables expr2 -> t_double
-------------------------------------
typeExpr tables expr1 + expr2 -> t_double

typeExpr tables expr1 -> t_string
typeExpr tables expr2 -> t_string
-------------------------------------
typeExpr tables expr1 + expr2 -> t_string

-----------------------------------
typeExpr tables expr ->
  error << "Incompatible types given to operator +" >>
```

Note that the last rule is executed only if all the previous failed, so when the recursive check did not fail or when the returned types were incompatible with the sum operator.

Typing a sequence of statements

Typing a sequence of statements requires to type check the first statement in the sequence and then recursively type check the remaining statements in the sequence. We also need a different type-checking function that is able to process statements and meta-data structure for its result.

```
Data TypeTableList -> "," -> Type : TypeResult
Func "typeStmt" -> TypeTableList -> Stmt : TypeResult
```

This function in general returns an updated list of type tables and a type, since variable declarations might change them. We use `t_unit` for the type of statements, which is a place holder for language constructs that just change the state of the program.

The base case of the recursion is when the sequence contains only `nop`, which returns immediately the same type tables.

```
-----------------------------------
typeStmt tables nop -> tables,t_unit
```

Type-checking a sequence of statements initially checks the first statement. This might return an updated list of tables. Then it recursively checks the other statements with the result of the first step and returns the final type tables. If either of the process returns an error we just propagate the error.

```
typeStmt tables a -> tables',error msg
-----------------------------------------
typeStmt tables (a;b) -> tables',error msg

typeStmt tables a -> tables',t_unit
typeStmt tables' b -> finalTables,error msg
-------------------------------------------------
typeStmt tables (a;b) -> finalTables,error msg


typeStmt tables a -> tables'
```

```
typeStmt tables b -> finalTables,t_unit
-------------------------------
typeStmt tables (a;b) -> finalTables,t_unit
```

Note that we are sure that the final rule succeeds because the type-checking of a statement always returns `unit` if the type-checking succeeds, according to the type rules of the language; this is further explained in the sections below.

Typing variable declarations and assignments

When we encounter a variable declaration we have to add the variable name and its type to the table of the current scope, unless the variable is already defined in the current scope, in which case we return an error. We must also prevent the declaration of variable with type `unit`, because that is a reserved type for statements. This is implemented with the following rules:

```
---------------------------------
typeStmt types (variable t_unit id) -> [],error << "The type unit cannot
      be used as a variable type" >>

types contains id -> Yes
---------------------------------
typeStmt (types :: tables) (variable t ($ name)) -> [],error << "
     Variable " + name " already defined" >>

types add id t -> types'
-----------------------------------------
typeStmt (types :: tables) (variable t id) -> types' :: tables
```

In the case of a variable assignment, the type checker must first look up in the type tables for the variable type. If the variable cannot be found then an error is returned because the program is trying to use an undefined variable. Otherwise we check the type of the right expression, and if it is compatible with the type of the variable then the declaration succeeds. Note that the process of checking the right side of the assignment might fail and, in this case, we have to propagate the error.

```
-------------------------------
typeStmt [] (($ name) = expr) -> [],error << "Variable " + name + "
     undefined" >>

typeExpr tables expr -> error msg
-----------------------------------------
typeStmt tables (id = expr) -> [],error msg

types contains id -> No
typeStmt tables (id = expr) -> res
-----------------------------------
typeStmt (types :: tables) (id = expr) -> res

types getValue id -> tvar
typeExpr (types :: tables) expr -> te
tvar <> te
-----------------------------------
typeStmt (types :: tables) (($ name) = expr) -> [],error << "Trying to
     assign an incompatible value to " + name >>

types getValue id -> tvar
-------------------------------------------
typeStmt (types :: tables) (id = expr) -> (types :: tables),tvar
```

Typing conditionals

Type-checking `if-then-else` requires to first check the type of the expression provided as condition. This process might fail and in this case we propagate the returned error. If the type checking of the expression succeeds but the returned type is not boolean, we have to return an error as well. Otherwise we can proceed to type-check the body of `then` and `else`. This process can again fail and we must again propagate a possible error. If no errors are returned after this step we return a possible updated list of type tables and the type `unit`.

```
typeExpr tables condition -> error msg
------------------------------------------------------------------
typeStmt tables (if condition then thenBlock else elseBlock) ->
  [],error msg

emptyDictionary -> table
typeStmt (table :: tables) thenBlock -> t,error msg
------------------------------------------------------------
typeStmt tables (if condition then thenBlock else elseBlock) ->
  [],error msg

emptyDictionary -> table
typeStmt (table :: tables) elseBlock -> t,error msg
------------------------------------------------------------
typeStmt tables (if condition then thenBlock else elseBlock) ->
  [],error msg

typeExpr tables condition -> tc
tc <> t_bool
------------------------------------------------------------
typeStmt tables (if condition then thenBlock else elseBlock) ->
  [],error << "The condition of an if-then-else must be boolean" >>

------------------------------------------------------
typeStmt tables (if condition then thenBlock else elseBlock) ->
  t_unit,tables
```

Note that the last rule does not type check again the code blocks of `if-then-else` because the only statement that can change a type table is a variable declaration, but after we exit the scope of the block the local declarations are removed. At this point we are sure that the type-checking of the blocks has succeeded, otherwise we would have triggered one of the rules above returning an error, thus we can immediately return the result.

Typing while-loops

Type-Checking a while loop is similar to the procedure of evaluating a conditional statement. We must first check that the provided condition is boolean. This might fail either because type-checking the condition itself returns an error or because the type of the expression is not boolean. After this step we have to check the body of the loop, which might fail as well. If no error is reported then we can safely return the correct result.

```
evalExpr tables condition -> error msg
-----------------------------------------------
typeStmt tables (while condition do block) -> [],error msg

evalExpr tables condition -> tc
tc <> t_bool
-----------------------------------------------
```

```
typeStmt tables (while condition do block) ->
  [],error << "The condition of a while loop must be boolean" >>

evalExpr tables condition -> tc
emptyDictionary -> table
typeStmt (table :: tables) condition -> t,error msg
-----------------------------------------------------
typeStmt tables (while condition do block) -> [],error msg


-----------------------------------------
typeStmt tables (while condition do block) -> tables,t_unit
```

Again note that the last rules can immediately return the result because we know that, at this point, we cannot have any error and we do not need to keep the type table of the scope of the code block.

Typing for loops

Type-checking a for loops requires to first type-check the initialization. This might fail and we must propagate the error. We must then type-check the condition and the step. This process can fail either because of an error in the condition or in the statement in the step, or because the condition is not boolean. If this succeeds we then proceed to type-check the body of the loop.

```
typeStmt tables init -> t,error msg
-----------------------------------------------
typeStmt tables (for init condition step do block) -> [],error msg

typeExpr tables condition -> error msg
-----------------------------------------------------
typeStmt tables (for init condition step do block) -> [],error msg

typeExpr tables condition -> tc
tc <> t_bool
-----------------------------------------------------
typeStmt tables (for init condition step do block) ->
  [],error << "The condition of a for loop must be boolean" >>

typeExpr tables condition -> tc
tc <> t_bool
-----------------------------------------------------
typeStmt tables (for init condition step do block) ->
  [],error << "The condition of a for loop must be boolean" >>

emptyDictionary -> table
typeStmt (table :: tables) step -> t,error msg
-----------------------------------------------------
typeStmt tables (for init condition step do block) -> [],error msg

emptyDictionary -> table
typeStmt (table :: tables) block -> t,error msg
-----------------------------------------------------
typeStmt tables (for init condition step do block) -> [],error msg

-----------------------------------------------------
typeStmt tables (for init condition step do block) -> tables,t_unit
```

4.2 The Casanova language

In the previous section we have shown how to implement a small imperative language using Metacasanova. In this section we show the implementation in Metacasanova of Casanova, a Domain-Specific Language for game development. We first give an informal explanation about how the language works and then we show an implementation of the language semantics.

4.2.1 The structure of a Casanova program

In this section we give an informal overview of a program in Casanova, leaving aside for brevity many of the details about the language itself, which can be found in [3, 5, 6, 4].

A program in Casanova is structured as a tree of *entities* that represent the dynamic elements of a game, where the root entity is special and called `world`. For instance, the following code snippet shows an entity depicting a movable character:

```
entity Character = {
  Position : Vector2
  Velocity : Vector2

  ...

  Create(p : Vector2) = {
    Position = new Vector2(3.0f, 5.0f)
    Velocity = Vector2.zero
  }
}
```

An entity is similar to a class in an object-oriented programming language, containing fields and a constructor. However, the difference lies in how the language implements the dynamic behaviour of an entity: each entity defines a set of *rules* that describe the temporal evolution of an entity instance. A rule operates on a set of fields of an entity called *domain*, and it is allowed changed only the values of the fields in its domain. A rule can write in a field of the domain only through a dedicated statement called `yield`. On the other hand, reading fields outside the domain is always possible. Each rule in an entity is run periodically up to a maximum refresh rate, which is usually set to 60Hz. One update cycle is called *frame*. Each rule is automatically passed two special identifiers, `this` and `dt`, where the former is a reference to the current instance of the entity and the latter the time elapsed between the last and the current frame.

Rules have mechanics similar to threads: they can be paused for a specific amount of time or until a certain condition is met. Furthermore, every time the rule executes a `yield` statement (thus changing the values of the fields in the entity) or its body has been completely evaluated, it is suspended until the next frame. Casanova also features interruptible control structures, such as `if-then-else`, `while-do`, and list comprehensions in a syntax similar to SQL or Linq (`from-where-select`).

The Casanova compiler generates the code to simulate the rule suspension and restart in the form of states machines. In the following section we show how to implement the same behaviour in the form of natural semantics in Metacasanova by using continuation-passing style.

4.2.2 Casanova semantics in Metacasanova

The memory in 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`. 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. For example, consider the following rule:

```
rule X,Y =
  while X > 0 do
    wait 1.0f
    yield X - 1,Y + 1
```

The code is executed atomically until the `wait` statement (assuming that the `while` condition is true). At that point we rebuild a new rule containing the code to execute at the next iteration:

```
rule X,Y =
  wait (1.0f - dt)
  yield X - 1, Y + 1
  while X > 0 do
    wait 1.0f
    yield X - 1,Y + 1
```

Note that `while` is placed at the end of the continuation because it must be re-evaluated after the first iteration is complete, and that we have decreased the waiting time by `dt` (the time elapsed between one frame and the previous one). This is analogous to the semantics of `while` implemented in Section 4.1.2. We now proceed to describe the implementation of Casanova semantics in detail. In what follows we assume that we already have evaluation rules for expression and for the symbol table as shown for C--, which we will not repeat for brevity.

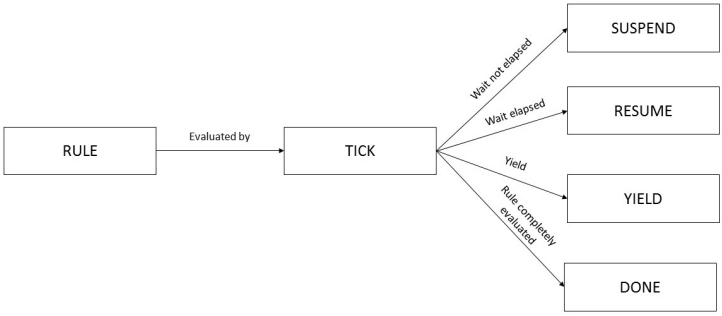


Figure 4.1: Possible results of the `tick` function

4.2.3 Rule update

As explained in Section 4.2.2, the rule update is implemented through a `tick` function that executes all the statements of the rule until an interruption statement (i.e. a statement that might pause the rule execution) is met. Thus, 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 rule 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 (see the semantics of the `wait` statement). In this case the argument of `Resume` is only the remaining statements to be executed. (*iii*) `Yield` stops evaluation for one frame. This is summarized in Figure 4.1. We use the continuation to store the rule body that has yet to be evaluated. The function definition is thus the following:

```
Data "rule" -> List[<<string>>] -> stmt -> stmt -> <<ImmutableDictionary
     <string, Value> >>  -> <<float>> : Rule
Data "Done" -> ctxt : ExecutionResult
Data "Suspend" -> stmt -> ctxt : ExecutionResult
Data "Yield" -> stmt -> List[Value] -> ctxt : ExecutionResult
Data "Resume" -> stmt -> ctxt : ExecutionResult
Data "Atomic" -> stmt -> ctxt : ExecutionResult
Func "tick" -> List[Rule] -> List[Rule] ->
  <<ImmutableDictionary<string, Value> >> -> <<ImmutableDictionary<
       string, Value> >> -> <<float>> : GameState
```

Note that in the implementation we use a generic meta-data structure `List` instantiated with the meta-type `Rule`. A `rule` is a meta-data structure containing a list of strings representing the domain, a sequence of statements representing the rule body, a second sequence of statement representing the continuation (i.e. the statements to be evaluated in the next frame), a symbol table of local variables, and the frame time difference.

As stated above, the `tick` function stops when all the rules have been evaluated, thus when both lists of rules are empty. In this case we return

the unchanged state of the program:

```
-----------------------------------------------------------
tick nil nil fields globals dt -> (State nil fields globals)
```

When the rule evaluation returns `Resume`, we build a rule containing the code to execute at the next frame, when the rule restarts, an empty continuation, because the current one has been moved into the body of the new rule, and the updated symbol table, since generally the rule evaluation might define some local variables. We then recursively update the remaining rules, and finally we build a new state with the rule that has to been resumed and all the other updated rules, that are stored in the state returned by the recursive call. Note that we will present the detail of `evalRule` further ahead.

```
evalRule (rule dom body k locals delta) fields globals -> Resume cont (
     Context newLocals newFields newGlobals)
r := rule dom cont nop 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
```

For instance, consider the rule in Listing 4.2 and assume that `dt = 1.0`.

```
rule X =
  wait 1.0f
  yield X + 1
```

Listing 4.2: Rule example with interruption

After evaluating the `wait` statement, the rule evaluation would return `Resume` containing the following continuation:

```
cont = yield X + 1
```

The new rule that will be generated is thus

```
rule X =
  yield X + 1
```

In the case of `Yield` the procedure is analogous, since `yield` pauses the rule execution for one frame and thus the continuation must be used to rebuild a new rule with the continuation in its body.

```
evalRule (rule dom body k locals delta) fields globals -> Yield cont
     values (Context newLocals newFields newGlobals)
r := rule dom cont nop 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
```

For instance, let us consider again the rule

```
rule X =
  yield X + 1
```

Its evaluation will generate a rule with an empty body, such as

```
rule X = nop
```

When the rule evaluation returns `Done`, it means that the rule statements have been completely evaluated. In this case the rule must pause for one frame. It is also necessary to rebuild the body of the rule as it was before its execution started. Indeed, during the execution, the rule body is "broken" when evaluating the body because the executed statements are thrown away during the recursive calls. In the previous examples we have seen this process in action (see Listing 4.2). As we can see in the meta-language rule below, this time we build a new set of rules by placing the rule in its original state.

```
evalRule r fields globals -> Done (Context newLocals newFields
     newGlobals)
tick originals rs newFields newGlobals dt -> (State updatedRules
     updatedFields updatedGlobals)
st := State (original::updatedRules) updatedFields updatedGlobals
-----------------------------------------------------------------
tick (original::originals) (r::rs) fields globals dt -> st
```

Finally, when the rule evaluation returns `Suspend`, we obtain the updated state of the `wait` statement (when the timer is updated) and a continuation. In this case we rebuild a rule whose body contains the updated `wait` statement and the continuation.

```
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
```

For instance, consider again the rule in Snippet 4.2 but this time with `dt = 0.5`. The rule update this time returns `Suspend` (because the timer has not elapsed yet) with:

```
s = wait 0.5f
cont = yield X + 1
```

thus the new rule will look like:

```
rule X =
  wait 0.5f
  yield X + 1
```

A summary of this process can be seen in Figure 4.2.

4.2.4 Rule evaluation

The function `evalRule` takes as input a rule and the symbol tables for the current entity and the `world` and returns an execution result, as seen in Section 4.2.3.

```
Func "evalRule" -> Rule -> <<ImmutableDictionary<string, Value> >> -> <<
     ImmutableDictionary<string, Value> >> : ExecutionResult
```

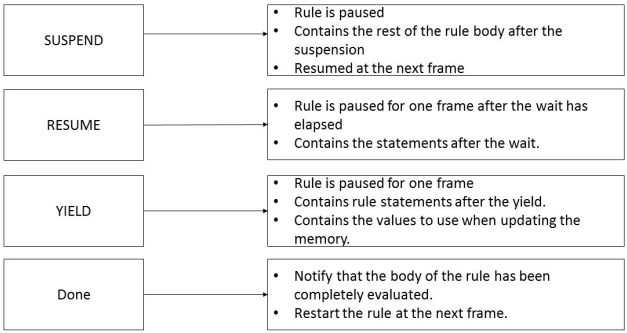


Figure 4.2: Cases of rule update

Semantics rules having `evalRule` in their conclusion call in one of their premises the function `eval_s`. This function is able to process a sequence of statements and return a result depending on the current statement being executed. When `eval_s` returns `Done`, `Suspend`, or `Resume`, `evalRule` simply forwards the result to `tick` as it is. On the other hand, `eval_s` can also return `Yield` and an additional result called `Atomic`. This kind of result represents a statement that does not pause the rule execution. `Atomic` statements are evaluated within the current frame until an interruption statement or the end of the rule is reached.

In the case of `Yield`, the function must update the fields of the entity before returning the result to `tick`, as shown below:

```
eval_s 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
     updatedContext
```

We omit the implementation details of `updateFields` for brevity; suffice to say that this function evaluates the expressions contained in `yield` and writes their values in the symbol table.

In the case of `Atomic`, `evalRule` the rule must immediately be re-evaluated in the current frame. This is obtained by recursively calling `evalRule` again with the current rule whose body has been replaced by the continuation returned by `Atomic` (which is simply the remaining code in the rule body).

```
eval_s b k (Context locals fields globals) dt -> Atomic z (Context
     newLocals newFields newGlobals)
evalRule (rule dom z nop newLocals dt) newFields newGlobals -> res
------------------------------------------------------------------
evalRule (rule dom b k locals dt) fields globals  -> res
```

A schematic representation of the interaction between `tick` and `evalStatement` can be seen in Figure 4.3.

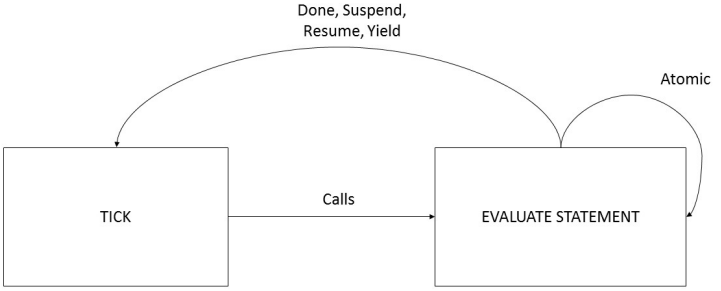


Figure 4.3: Rule update in Metacasanova

4.2.5 Statement evaluation

Statement evaluation is implemented through the function `eval_s`. This function takes as input a sequence of statements or a single statement and returns a different result depending on the statement semantics. This function takes as input the current body of the rule, its continuation and the context of the program made by the symbol tables of `world`, the current entity, and the local variables of the rule.

```
Func "eval_s" -> stmt -> stmt -> ctxt -> <<float>> : ExecutionResult
```

The base case of `eval_s` is when the body of the rule is empty and there is no continuation. This is the case when the whole rule body has been executed and thus we have to return `Done`.

```
-------------------------------
eval_s nop nop ctxt dt -> Done ctxt
```

When the rule body is non-empty, then we must extract the first statement in the statement sequence. We then combine the remaining body of the rule with the current continuation into a single statement sequence by using the function `addStmt`. This function has two cases: (*i*) both the remaining body of the rule and the continuation are empty, or (*ii*) the body of the rule is non-empty. The first case happens when we are executing the last statement in the rule body. In this case we generate an empty continuation containing `nop`. In the second case we simply combine the remaining body of the rule and the continuation into a single statement sequence.

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

--------------------
addStmt nop nop -> nop
```

Note that the case where only `b` is `nop` cannot be generated, because executing a statement will always generate a non-empty continuation, unless it

is the last statement of the rule to be executed. This case is captured by `eval_s` (as shown above), which will return `Done`. When `Done` is forwarded as result to `tick`, the body of the rule will be regenerated by replacing it with the initial code of the rule (we reset the rule) as shown in Section 4.2.3.

After the new continuation has been generated, we recursively call `eval_s` by giving it as input the first statement in the rule body.

```
addStmt b k -> cont
eval_s a cont ctxt dt -> res
------------------------------
eval_s (a;b) k ctxt dt -> res
```

We now proceed to show how the semantics of the statements is implemented

Interruption statements

Interruptible statements are statements that can pause the execution of a rule: `wait` and `yield`. As briefly pointed out before, `yield` returns as result a meta-data structure `Yield` containing the continuation of the rule to resume at the next frame, the values to write in the domain fields, and the current program context (the symbol tables). Since the arguments of `yield` can be expressions, its semantics must evaluate them one by one and return their values.

```
-------------------------
evalYield nil ctxt -> nil

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


evalYield exprs ctxt -> values
-----------------------------------------------------
eval_s (yield exprs) k ctxt dt -> Yield k values ctxt
```

The statement `wait` in Casanova has double semantics: one waits for a timer to elapse and the other until a certain condition is met. In Metacasanova we do not have overloading, thus we are forced to use a different name to model both cases of its semantics. We use `wait` for the timed and `when` for the conditional version of the statement.

For the timed version we have two cases: (i) the timer has elapsed and we can resume the execution of the rule at the next frame, or (ii) the timer is still running, thus we have to suspend the rule. In the first case we return `Resume` containing the current continuation of the rule body. In the other case we have to suspend the rule, thus we return `Suspend` where the continuation contains `wait`, whose timer has been updated by removing `dt`, concatenated to the current continuation of the rule.

```
eval expr ctxt -> ($f t)
t > dt
<<t - dt>> -> t'
-----------------------------------
eval_s (wait expr) k ctxt dt -> Suspend (wait $f t');k ctxt

eval expr ctxt -> ($f t)
```

```
t <= dt
---------------------------------
eval_s (wait expr) k ctxt dt -> Resume k ctxt
```

The implementation of `when` is analogous: if the condition is not met then we simply return `Suspend` where the continuation contains `when` concatenated with the previous continuation. Otherwise we return `Resume` containing the current continuation.

```
eval expr ctxt -> ($b true)
-------------------------------------------
eval_s (when expr) k ctxt dt -> Atomic k ctxt

eval expr ctxt -> ($b false)
----------------------------------------
eval_s (when expr) k ctxt dt -> Suspend (when expr);k ctxt
```

Control strucutres

Casanova supports a conditional control structure (`if-then-else`), and two iterative control structures (*while-do* and `for-do`). The only difference with the usual semantics of control structures lies in `for-do`, which is like that of Python and F# as it takes as input a variable that is used to iterate through the elements of a list.

The implementation of the control structures is similar to that of C-- with the difference that their body can be interrupted as well. At this purpose, the semantics rules must generate an appropriate continuation that will be handled by `tick`. The conditional control structure checks if the condition is true or false to select the appropriate code block to execute. After that it returns an `Atomic` result containing the concatenation of the selected code block with the current continuation of the rule. This is because the evaluation of the condition is an atomic process, i.e. must not pause the execution of the rule. The first statement of the selected code block will be executed immediately after.

```
eval cond ctxt -> $b true
-----------------------------------------------------------
eval_s (if cond then b else c) k ctxt dt -> Atomic b;k ctxt

eval cond ctxt -> $b false
-----------------------------------------------------------
eval_s (if cond then b else c) k ctxt dt -> Atomic c;k ctxt
```

`while-do` follows the same behaviour described in C--, thus if the condition is false we simply skip the loop, otherwise we execute the body followed by the same loop. This time we must encapsulate the code built after the evaluation of the condition in an `Atomic` result, because the body must be immediately evaluated after checking the condition, as for conditionals.

```
eval cond ctxt -> $b true
-------------------------------------------------------------------
eval_s (while cond b) k ctxt dt -> Atomic b;((while cond b);k) ctxt

eval cond ctxt -> $b false
-------------------------------------------------------------------
eval_s (while cond b) k ctxt dt -> Atomic k ctxt
```

The semantics of `for-do` loop are quite different than what we had defined for C--. The loop defines a variable that is used to iterate each element of a list. The list can be given directly or be an expression that returns a list. Thus, we have to first add to the local variables the one defined in the loop, then evaluate the expression for the list, and finally evaluate the body of the loop itself. Note that lists here are considered lists in the Casanova language and not lists of Metacasanova (thus they are values in Casanova and not meta-data structures).

```
eval expr ctxt -> ($1 nil)
-----------------------------------------
eval_s (for v in expr b) k ctxt dt -> Atomic k ctxt

eval expr (Context locals e w) -> ($1 (x :: xs))
locals add var x -> updatedLocals
-----------------------------------------------------------------
eval_s (for ($ var) in expr b) k (Context locals e w) dt -> Atomic b;((
     for ($ var) in ($1 xs) b);k) (Context updatedLocals e w)
```

The base case of the evaluation is when the list is empty. In this case we simply return `Atomic` containing the current continuation because the loop can be skipped. The recursive case is when the list is non-empty: in this case we first evaluate the expression of the list, then we add the variable defined in the loop to the local variables, assigning it the value of the head of the list. We finally return an `Atomic` that contains a continuation where the body of the loop is concatenated to the loop itself and the current continuation. `Atomic` will also contain a program context where the locals now contain the new variable defined in the loop.

Note that, for simplicity, we do not have code block scoping like in C--. This feature can be implemented by replacing the local symbol table with a list of tables as previously shown in Section 4.1.1.

4.3 Evaluation

In this section we compare the runtime performance of a program written for C-- and Casanova implemented in Metacasanova with their equivalent implementation in Python. Moreover, we evaluate the length of the language definition in Metacasanova with respect to their hard-coded implementation. We begin by describing the experimental set-up, we proceed by explaining how we analyse the results, and then we discuss them

4.3.1 Experimental Set-up

We evaluated C-- and the meta-compiled Casanova runtime performance against an equivalent implementation of equivalent programs in Python. C-- was tested running a program to compute the factorial, while we implemented a program in Metacasanova where some entities patrol an area according to pre-defined checkpoints. In the case of Casanova, this language was chosen based on its use in game development: Python has been used extensively in several games such as Civlization IV [28] or World in Conflict [40] because of the native support for coroutines that allow to implement a behaviour similar to that of Casanova rules. In the case of C--, we still

use Python because, as we discuss further ahead, the behaviour of this language is much more similar to that of a dynamic language (the name was chosen mainly because of a lack of creativity from the author than because of its similarity with C). As for the code length, we compare the length of the semantics definition of C-- with a hard-coded implementation, while we compare the definition of Casanova in Metacasanova with respect to its hard-coded compiler written in F#.

For Casanova we use a program where a Casanova entity patrols a set of checkpoints. When the entity reaches the position of a checkpoint it will move to the next one. The same code has been re-implemented in Python using coroutines to simulate the interruption mechanism of rule statements that is built-in in Casanova. The code generated by the version of Casanova implemented in Metacasanova was imported in a C# program for Monogame but tested in isolation to actually measure only the running time of the logic, which would otherwise be influenced by the rendering time and the overhead of Monogame itself. We run the Casanova program and the Python version with a variable number of entities (that will be updated) ranging from 100 to 250. For each execution we measure the time taken to update them all for each frame, and we average this time on the number of total frames. As for the code length of the language definition, we measure the length of the language specification in Metacasanova and we compare it with the relative parts of code in the hard-coded version of the compiler.

4.3.2 Performance

From Table 4.3 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#, while the C-- implementation is 11 times shorter (Table 4.4). 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. In the hard-coded compiler implementation for Casanova 2.0 the semantics are lost in the code for generating finite state machines. Just for comparison, Figure 4.4 shows the code from the Casanova hard-coded compiler to generate part of the state machine necessary to simulate the behaviour of the timed version of `wait` (the code generation of `when` has about the same size).

The performance results are shown in Table 4.2. We see that the generated code has performance on the same order of Python, although 3 times slower. This gap is accentuated in the case of C--, which is 50 times slower than Python, because in the case of a simple imperative program, where

```
| t_expr, OptimizedQueryAST.Wait((tp, expr), _) ->
  let float_type = TypeDecl.ImportedType(typeof<float32>, t_expr.Position)
  let bool_type = TypeDecl.ImportedType(typeof<bool>, t_expr.Position)
  e.CountdownCounter <- e.CountdownCounter + 1
  let tp = TypeDecl.ImportedType(typeof<float32>, tp.Position)
  let lb, lb_expr = get_fresh_label t_expr.Position e
  let count_down_id = {idText = "count_down" + (string e.CountdownCounter); idRange = t_expr.Position}

  let f_expr =
    match expr with
    | f_expr when tp = TypedAST.TypeDecl.ImportedType(typeof<float32>, t_expr.Position) -> f_expr
    | i_expr when tp = TypedAST.TypeDecl.ImportedType(typeof<int>, t_expr.Position) -> TypedAST.ImplicitIntCast(i_expr)
    | _ -> raise "Sate machines error. Wait arguments can be either of type int or float."

  let count_down =  TypeDecl.ImportedType(typeof<float32>,t_expr.Position), Expression.Var(count_down_id , tp, traverseTrivialTypedExpr (tp, f_expr) |> Some)
  let wait_lb, wait_lb_expr = get_fresh_label t_expr.Position e
  let goto_wait_lb = (TypeDecl.Unit t_expr.Position, Goto.Create wait_lb |> Expression.Goto)
  let _then =
    [(TypeDecl.Unit t_expr.Position,
      Expression.Set(count_down_id,
                     (float_type,
                      StateMachinesAST.Sub((float_type, StateMachinesAST.Id(count_down_id)),
                                           (float_type, StateMachinesAST.Id({idText = "dt"; idRange = t_expr.Position}))))));
     (TypeDecl.Unit t_expr.Position, GotoSuspend.Create wait_lb |> Expression.GotoSuspend)]
  let _else = [exit_expr]
  let cond = bool_type, Greater((float_type, Id(count_down_id)), (float_type,StateMachinesAST.Expression.Literal(BasicAST.Float(0.0f, t_expr.Position))))
  let if_then_else = TypeDecl.Unit tp.Position, Expression.IfThenElse(cond,None, _then, _else)
  lb, [lb_expr; count_down; goto_wait_lb; wait_lb_expr; if_then_else]

| t_expr, OptimizedQueryAST.ReEvaluateRule(p) ->

  let lb, lb_expr = get_fresh_label t_expr.Position e
  let goto_exit =
    match exit_expr with
    | t, GotoSuspend(g) -> t, Goto(Goto.Create(g.Label))
    | t, Goto(g) -> exit_expr
```

Figure 4.4: Code generation of `wait` in the Casanova compiler

the use of virtual tables for polymorphic types (as coroutines) is limited, the speed of Python greatly increases.

4.3.3 Discussion

Even though the size of the code required to implement the language has been drastically reduced (almost 1/5 shorter), performance dropped dramatically. The problem lies in the fact that, in order to implement a memory model, in the current version of Casanova we must rely on dynamic access to a symbol table at runtime. Indeed, when we define a new variable or read its value, the semantics contain a rule defining the insertion or the lookup of the variable. Metacasanova generates the code able to run those rules, but the memory operations are thus executed at runtime as dictionary operations.

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.

The same applies to type checking: in Section 4.1.3 we showed the type rules that check the types of a C-- program. In statically-typed languages,

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 4.2: 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 4.3: meta-compiler vs standard compiler

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

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

Table 4.4: Code length implementation of C-- and run-time performance

type checking is usually performed at compile time and not at runtime. However, in this case Metacasanova will again generate the code to run the type rules, but the actual execution is performed when the program is run, thus the behaviour of C-- is more similar to that of a dynamic language rather than a static language (and its performance as well).

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 means that, as it is, the meta-language is unable to define a statically-typed language. 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.

The same applies for the lookup: the access to symbol tables needs not to be dynamic because the symbol table does not grow when the program runs, thus the access to a specific variable could be directly inlined in the code. For example, if we want to access variable `x`, which is the third entry of the symbol table, we will always perform the same lookup. Thus, this lookup could be simply inlined as an access to the third element of the symbol table. An analogous situation happens for Casanova entities: their structure does not change at runtime, so if we access, for instance, a field `Velocity` of an entity and that is the third one, then we always perform a lookup on the third element of the symbol table, and this can be inlined directly as well.

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.

4.4 Summary

In this chapter we showed two examples of how to use Metacasanova to implement programming languages. We started by showing how to implement a small imperative language called C--. We showed an implementation of its semantics and then of its type system. Later we re-implemented the Casanova language, a DSL for game development. We showed how to implement the semantics of interruptible code, which in Casanova had been implemented with state machines, by using continuation-passing style. Metacasanova implementation of Casanova results to be 5 times shorter than that of the hard-coded compiler for Casanova written in F#. In the case of C-- the gap in terms of lines of code is even larger, being the code for its semantics 11 times shorter than a hard-coded implementation. How-

ever, the code performance drops dramatically: testing the meta-compiled version of Casanova against Python results in its code being 3 times slower (although on the same order of magnitude), while the C-- code is even 50 times slower. The cause of this is that, even thought these languages could be statically-typed, the rule evaluation performs the lookup of variables and types at runtime. This cannot be changed in the current state of the language because the meta-type system of Metacasanova is unaware of the types of the embedded language (i.e. the language that is being implemented in Metacasanova). In the next chapter we will show a language extension for Metacasanova that relies on *Functors* to embed the type system of languages implemented in Metacasanova in its type system, and to inline the access to variables at compile time.

Chapter 5

Metacasanova optimization

In Chapter 3 and 4 we have presented the Metacasanova metacompiler and its meta-language and shown how to implement with it a small imperative language, C--, and a DSL for game development, Casanova. The performance analysis showed that, although the development effort for the language compilers was greatly reduced by using Metacasanova, this has come to the cost of performance. The performance decay is due to the fact that the meta-type system of Metacasanova is unaware of the type system of C-- or Casanova. This requires all the type checking and access to data structures being performed at runtime, thus making a statically-typed language exhibit the behaviour and performance of dynamically typed languages. In this Chapter we propose a language extension [24] for Metacasanova that is thought to overcome the problem of performance decay and dynamic checks. In this context we use the term *embedded language* to refer to a language that is being implemented in Metacasanova and *embedded program* for a program implemented in an embedded language.

5.1 Language extension idea

The experimental results from Chapter 4 showed that the performance of Metacasanova is strongly affected by the dynamic type checks and symbol table access at runtime. This is due to the fact that Metacasanova generates the code necessary to evaluate the semantics of accessing the value of a variable in the symbol table that mimics the behaviour of rules in natural semantics, but such evaluation is performed at runtime. However the runtime evaluation is only due to the limitations of the language presented so far, which is not able to build a symbol table while while compiling the meta-program, since

1. The symbol table of a statically-typed language does not grow at runtime because it is built during the compilation.

2. The position of an entry for a variable in the symbol table does not change during the program execution, thus every time we perform an

access to the same variable, we access the very same element in the symbol table.

Analogously type checking in a statically-typed language is performed at compilation time rather than at runtime, which is a behaviour typical of dynamic languages such as Python. Metacasanova is forced to do runtime type checking because, at compilation time, the metacompiler only checks for the meta-types, i.e. the types of the language abstractions defined in the meta-language, but not for the program structures of the embedded program itself. This would require to be able to embed the type system of the embedded language into the meta-type system of Metacasanova. In this way the type checker of Metacasanova would be able to check at the same time the types of both the meta-program and of the embedded program.

To better clarify what stated so far we show in the following section an example of what happens when accessing the field of a Casanova entity with the implementation given in Chapter 4. We then proceed to show the idea of a possible solution to overcome the performance decay.

5.1.1 Field access in Casanova

As we showed in Section 4.2.2, an entity in Casanova embedded in Metacasanova is represented through a map where the key is the field name and the value is the value currently stored in the field. This representation is very similar to that of records or classes. Let us consider the following entity representing a physical body consisting of a `Position` and a `Velocity` in a 2D space:

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

and the following rules for `PhysicalBody`

```
rule Position = Position + Velocity * dt

rule Position =
  if Position.X > 500f then
    yield new Vector2(500f,Position.Y)
  elif Position.X < 0f then
    yield new Vector2(0f,Position.Y)
  elif Position.Y < 0f then
    yield new Vector2(Position.X,0f)
  elif Position.Y > 500f then
    yield new Vector2(Position.X,500f)
```

The first rule simply updates the position using the Euler approximation of the differential equation for the velocity

$$v(t) = \frac{ds(t)}{dt}$$

while the second rule ensures that the physical body does not exit a specific area, which could represent the playable area in a 2D game.

Assuming that the physical body is in position $(10, 10)$, it is represented in Metacasanova through a map as shown in Table 5.1.

Field	Value
Position	10
Velocity	10

Table 5.1: Meta-representation of the physical body

⇒

Field	Value
Position	10,10
Velocity	10,0

⇒

Field	Value
Position	10,10
Velocity	10,0

⇒

Field	Value
Position	11,10
Velocity	10,0

Table 5.2: Memory access in the first rule of the Physical Body. We assume `dt = 0.1` and `Velocity = (10,0)`

The Metacasanova semantics rule that evaluates the first Casanova rule will evaluate the expression in its body by accessing respectively the field `Position` and `Velocity` to compute the expression value. It then stores the expression value in `Position` as shown in Table 5.2.
As for the second rule, assuming that `Position.Y > 500f`, the rule will access `Position` three times: (*i*) to evaluate the expression in the conditional, (*ii*) to read `Position.Y` when instantiating a new vector, and (*iii*) to write the new vector in `Position`. This situation is shown in Table

It should now appear clear that every time we need to read or write `Position` we access the first element of the table, while for `Velocity` we always access the second. In the following snippet we provide an alternative version of the code for the Casanova rules above that shows what really happens in Casanova embedded in Metacasanova :

```
rule Position = PhysicalBodyTable[0] + PhysicalBodyTable[1] * dt
```

⇒

Field	Value
Position	501,10
Velocity	10,10

⇒

Field	Value
Position	501, 10
Velocity	10,10

⇒

Field	Value
Position	500,10
Velocity	10,10

Table 5.3: Memory access in the second rule of the Physical Body. We assume `Position.X = 501`

```
rule Position =
  if PhysicalBodyTable[0].X > 500f then
    yield new Vector2(500f,PhysicalBodyTable[0].Y)
  elif PhysicalBodyTable[0].X < 0f then
    yield new Vector2(0f,PhysicalBodyTable[0].Y)
  elif PhysicalBodyTable[0].Y < 0f then
    yield new Vector2(PhysicalBodyTable[0].X,0f)
  elif PhysicalBodyTable[0].Y > 500f then
    yield new Vector2(PhysicalBodyTable[0].X,500f)
```

Let us now assume that the program provides an invalid value for the update of `Position`:

```
rule Position = "(10,10)"
```

what would happen in embedded Casanova is that the type checker evaluates the type of the expression in the rule body, obtaining `string`. This type is then compared with that of `Position`, which is `Vector2`, and at this point an error would be reported. Again, this would require at runtime to access the first element of a symbol table containing type information about the entity fields. Note that all these lookups are not array accesses but rather dictionary accesses.

5.1.2 Inlining the entity fields

From the example above we can notice that, when the program runs, the symbol table used to represent a Casanova entity does not change, nor its entries change position. This means that every time we read or write the same field we perform the same access in the table. In the implementation provided in Section 4.2.2 this access requires to evaluate a Metacasanova rule that is able to traverse the dictionary used for the entity symbol table and return the stored value. The traverse is performed every time, regardless of the fact that the field we are trying to access is indeed the same. Moreover, as it was also stated in Section 4.3, we are looking at the very optimistic scenario where we make use of external .NET dictionaries to actually model the entity. If one had to rely solely on language abstractions defined in Metacasanova the symbol table should be modelled as a list of pairs containing field names, represented as strings, and meta-data structures representing values in the embedded language, introducing even a greater overhead. The physical body modelled in such way would then look like

```
[("Position",(10,10)),("Velocity",(10,0)]
```

Accessing `Position` would then be performed by a Metacasanova rule that looks for the correct field name and stops when the field in this tuple has been reached:

```
name = fieldName
---------------------------------
getField name ((fieldName,value) :: t) -> value

name <> fieldName
getField name t -> v
---------------------------------
getField name ((fieldName,value) :: t) -> v
```

However the traversal of the tuple would always be the same when looking for a specific field, namely for `Position` the first Metacasanova rule will always be executed, while for `Velocity` the first time the second rule will be executed, which in turn recursively evaluates the remaining part of the list. The recursive call will then trigger the first rule at the second step. That being said, since the table does not grow and the access patterns are always the same, we could represent an entity as a nested tuple of pairs, in the fashion of Church encoding [53, 37], and inline in the code `fst PhysicalBodyTable` for `Position` and `fst(snd PhysicalBodyTable)` for `Velocity` whenever we require to access the respective fields, without repeating the same traversal every time. In this way the entity would look like:

```
PhysicalBodyTable = ("Position",(10,10)),(("Velocity",(10,0)),())
```

and thus `fst PhysicalBodyTable` (access to `Position`) would return `("Position",(10,10))` and `fst(snd PhysicalBodyTable)` (access to `Velocity`) would return `("Velocity",(10,0))`.

In the following sections we present the language extension required to allow this form of inlining and we show their usage implementing the example above.

5.2 Modules and Functors

In order to implement the idea about inlining symbol table access and embed the type system of a language inside Metacasanova type system we extend the language with *functors* and *modules*. Functors are a concept borrowed form category theory that here are used in a more narrow sense. Formally a category is defined as follows [11, 46, 52]:

Definition 5.1. A category $\mathcal{C}$ is made of

- A collection of *objects*.
- A collection of *arrows* or *morphism* between objects. Each morphism starts from a source object and ends into a target object.
- For every triplet of objects, there exists a composition operation $\circ$, such that, given the morphisms $f : a \to b$ and $g : b \to c$ then $g \circ f : a \to c$.
- The composition operation is associative, i.e. $f \circ (g \circ h) = (f \circ g) \circ h$.
- For each object x There exists a morphism $1_x : x \to x$, called *identity*, such that for every morphism $f : a \to x$ and $g : x \to b$ we have that $f \circ 1_x = f$ and $g \circ 1_x = g$.

Functors are mapping between two categories defined as follows:

Definition 5.2. Given two categories $\mathcal{C}_1$ and $\mathcal{C}_2$, a *functor* $\mathcal{F}$ from $\mathcal{C}_1$ to $\mathcal{C}_2$ is a mapping such that:

- Each object x of $\mathcal{C}_1$ is mapped to an object $\mathcal{F}(x)$ of $\mathcal{C}_2$.
- Each morphism $f : a \to b$ of $\mathcal{C}_1$ is mapped to a morphism $\mathcal{F}(f) : \mathcal{F}(a) \to \mathcal{F}(b)$ such that
  - $\mathcal{F}(1_x) = 1_{\mathcal{F}(x)}$.

– For all morphism $f : a \to b$ and $g : b \to c$ of $\mathcal{C}_1$ we have that $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$.

Informally, functors are transformations between categories that preserve the identity and the associativity properties. In the scope of programming languages the term functor is used with a more narrow sense: they usually define transformations between types. These transformations are functors (actually *endofunctors* since they transform elements of the category of types in elements of the same category) at all effects but not all functors from category theory coincide with functors in a programming language. Popular programming languages that provide functors in this sense are Haskell with *Type Classes* [34, 36, 41, 59, 63] and Caml with *Modules* [39, 49, 64]. Functors in Metacasanova are no different: they define transformations between types. Modules are simply collections of function and functor declarations grouped together under the same name that can be used as types themselves.

5.2.1 Language Extension

Modules can be defined through the keyword `Module` followed by a module name and series of construction parameters that are used to create an instance of the module. Constructions parameters have a form similar to parameters in normal functions with the difference that, besides specifying the type, we can also specify an identifier for that parameter. The special symbol `*` (*kind*) can be used if any type is suitable for that specific argument. Elements of a module can be accessed with the `.` access operator.

```
Module "M" => ma1 : t1 => ma2 : t2 => ... => ma_k : tk : M {
  Func "f1" -> ...
  Func "f2" -> ...
  Func "f_k" -> ...

  ...
}
```

Functors are defined similarly to function but using the double arrow instead of the single arrow:

```
Functor "foo" a1 => a2 => ... => an : T
```

Moreover, since the result of calling a functor is a type, functors can be used wherever a type annotation is required, for example in the declaration of a function

```
Func "bar" b1 -> b2 -> ... -> (foo a1 a2 ... an) -> ... : U
```

Functors are evaluated through rules whose behaviour is identical to those used to evaluate functions. The difference lies in the fact that results of functors are evaluated at compile time rather than runtime. Functors results are evaluated by an interpreter that mimics the semantics of rules in natural semantics, in the fashion of the semantics used in the code generation explained in Section 3.6. Since the evaluation is performed at compile time, all the values passed to a functor call must be known when compiling the meta-program. This means that the arguments of a functor call can be either types or constants. When an evaluation rule for a functor is called,

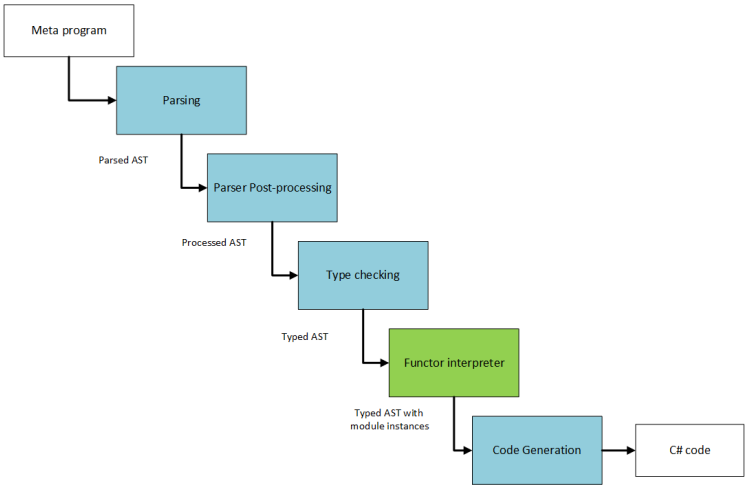


Figure 5.1: Compiler architecture with functor interpreter

this is run through the interpreter and a module instance is returned as result. Figure 5.1 shows the steps performed by the new compiler architecture to include functors interpretation. Functors can be called both in the premises of rules for functors and for rules that evaluate regular functions. In the latter case, the premise will simply instantiate the module that can then be used within the rule itself. This process is shown in Figure 5.2: the functor call is processed by selecting the possible candidate rules to execute it, in the same fashion of what is done for regular functions. At this point the interpreter run the rule and the result of the first one that succeeds is taken. The result of such rule is a module instantiation. The module instantiation is bound to the variable contained in the result of the premise. From that point on, the module instance can be referred by the caller rule.

In the following sections we show how to implement the mechanism of inlining for the record getter and setter described in Section 5.1 that makes use of the compile-time interpretation of functors.

5.3 Record implementation with modules

In Section 5.1.2 we showed how Casanova entities can be expressed, at meta-language level, as a tuple of field names and values. We also showed that getters and setters always perform the same steps when looking up for the same field because the entity structure does not change at runtime. In this section we proceed to give an implementation based on functors to implement a Casanova entity. We refer to this implementation as “Record”, since a Casanova entity is simply a record from the point of view of the data representation and since this solution works in general for any data structure that is isomorphic to a record. From now on we also use, as example, the physical body entity described in Section 5.1.1.

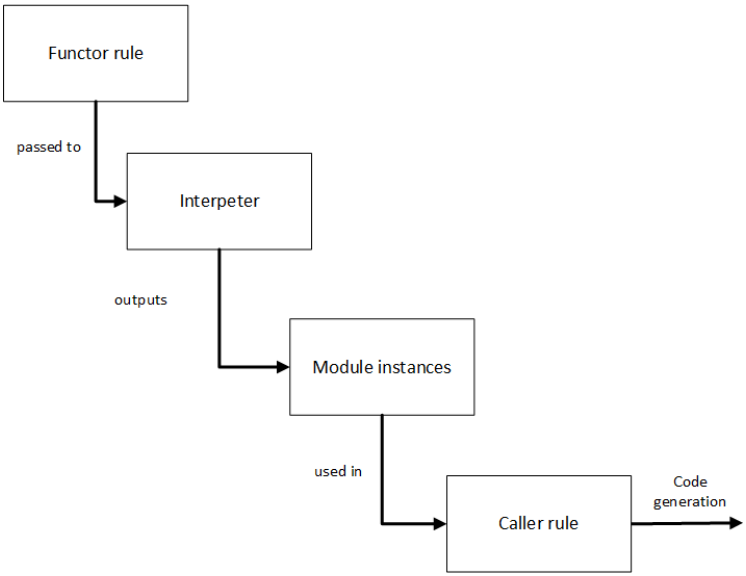


Figure 5.2: Functor processing

A module for records simply contains a functor that returns the type of the record. This functor, in general, can return any type since the type of the record can be "customized" and depends on the specific definition given by the programmer (thus it cannot be known beforehand). For this reason we use *kind* as return type for this functor. The functor itself is parameterless since nothing is required to generate the type of a record.

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

The data representation of the record will be a tuple as shown in Section 5.1.2. For this purpose, we need two functors that are able to represent the type of a record in a recursive way with one being the type of an empty record (a record with no fields) and another a record field followed by the rest of the record representation. The functor for the empty record simply returns the type of the record module, while the functor to represent a record field takes as input a `string`, representing the name of the field, *kind* because a record field can have any type, and a `Record` which represents all the other fields coming after the current one.

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

After declaring the functors necessary to build a record, we proceed to define their implementation in the form of rules. The functor for an empty record simply generates a module containing a function `cons`, that is the constructor for the record, that simply returns unit (because an empty record

does not contain any field). Consistently, the functor `RecordType` implemented by the module will simply return `unit` as type. Note that a module instantiation must implement **at least** all the declarations of the module (like for an interface), but can add other declarations and implementations that are not shared by all the module instantiations. For example `cons` for an empty record is different than the one for a non-empty one.

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

  Func "cons" : unit

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

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

}
```

A record field must be constructed through a functor that takes the field name, the type of the field, and the type of the rest of the record. This functor will construct the type of a record as a `Tuple`, where the first element is the type of the current field and the second the type of the rest of the record. The constructor of the record field will be a function that takes as input an argument of the type of the current field, a tuple representing the remaining part of the record and returns a tuple combining the current field and the rest of the record.

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

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

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

Consider now the physical body representation given above. We show how to use the functors we have just defined to build an instance of a physical body. First of all we defined a functor `PhysicalBodyType` that returns a `Record`.

```
Functor "PhysicalBodyType" : Record
```

The final representation of the type that should be returned by `PhysicalBodyType` is `Tuple[Vector2,Tuple[Vector2,unit]]` because the field `Position` and `Velocity` have type `Vector2`. Note that `Vector2` can be trivially implemented in Metacasanova as a tuple containing two floating point values. Here we use this type assuming that has already been defined above. The same appiles to `unit`, which can be defined as a meta-data with no arguments.

The rule to evaluate `PhysicalBodyType` will call in its premises `EmptyRecord` and `RecordField` to generate the type of the tuple appropriately:

```
EmptyRecord => empty
RecordField "Velocity" Vector2 empty => velocity
```

```
RecordField "Position" Vector2 velocity => body
----------------------------
PhysicalBodyType => body
```

Let us now analyse in detail what the premises generate: the first premise will generate an instance of `EmptyRecord` and bind it to the variable `empty`. The instance of this module is parameterless and thus will always be the same every time the functor is invoked. The second premise will instantiate `RecordField` by using the string `"Velocity"` as field name, `Vector2` as field type, and `empty` as argument for the remaining part of the record (there is no other field after `Velocity` in the physical body). The instantiation of `RecordField` produces a rule for the functor `RecordType`. According to the definition above this functor generates `Tuple[type,r.RecordType]`. By replacing the argument values provided in the premise, we have

```
type := Vector2
r := empty := EmptyRecord
```

Thus `r.RecordType` uses the functor `RecordType` in the instance of `EmptyRecord` which returns the type `unit` (the call can be seen as `empty.RecordType`). Thus `r.RecordType` can be replaced with `unit`, thus leading to `Tuple[Vector2, unit]`. Thus the rule for the functor `RecordType` generated in the module returned by the second premise will be.

```
-----------------------
RecordType => Tuple[Vector2,unit]
```

By replacing the argument variables with the values provided in the second premises we can also get the declaration and rule for `cons`. By replacing again `type` and `r.RecordType` as done before, we have that the declaration for `cons` in the current instance of the module becomes:

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

while the corresponding rule will be generated as

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

The complete module instance will then look like:

```
velocity := Record {
  Func "cons" -> Vector2 -> unit: Tuple[Vector2,unit]

  -----------------------
  RecordType => Tuple[Vector2,unit]

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

The third premise calls `RecordField` with

```
name := "Position"
type := Vector2
r := velocity
```

Now in the definition of the `RecordField` module again the functor `RecordType` returns `Tuple[type,r.RecordType]`. Now `r.RecordType` can be rewritten as `velocity.RecordType` that returns (see the instantiation of `velocity` above) `Tuple[Vector2,unit]`. Thus
`RecordType` for the field `Position` will be instantiated as

```
-----------------------
RecordType => Tuple[Vector2,Tuple[Vector2,unit]]
```

Analogously the declaration of `cons` will be instantiated as

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

while its rule is the same. The full module instance will then be

```
body := Record {
  Func "cons" -> Vector2 -> Tuple[Vector2,unit]: Tuple[Vector2,Tuple[
       Vector2,unit]]

  -----------------------
  RecordType => Tuple[Vector2,Tuple[Vector2,unit]

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

which is returned by the functor `PhysicalBodyType`. In order to build an instance of the physical body, we define a function that returns a value of type `PhysicalBodyType` (which in turn is simply `Tuple[Vector2,Tuple[Vector2,unit]]`):

```
Func "PhysicalBody" : PhysicalBodyType.RecordType

-----------------------
PhysicalBody -> PhysicalBodyType.cons((10.0,10.0),((10.0,0.0),()))
```

The rule creates a physical body in position $(10, 10)$ moving at velocity $(10, 0)$.

One of the main arguments in favour of using functors was that they should allow to embed the type system of the embedded language in the meta-type system of Metacasanova. This means that, at compile time, the meta-compiler should be able to detect a physical body that is constructed in the wrong way. Let us then assume that we define another function to build a physical body where the programmer uses a scalar for the velocity instead of a vector:

```
Func "WrongPhysicalBody" : PhysicalBodyType.RecordType

-------------------------------------
WrongPhysicalBody ->  PhysicalBodyType.cons((10.0,10.0),(10.0,()))
```

What happens is that `PhysicalBodyType.RecordType` is equal to `Tuple[Vector2,Tuple[Vector2,unit]]`. At this point the type checker of Metacasanova will successfully match the first element of the tuple returned by the rule, which is correctly provided as a value of type `Vector2`, but will fail to check the second, which is `double` where it expects a `Vector2`.

With the implementation based on dictionaries given in Section 4.2.2 this check happens dynamically at runtime by means of type rules defined in the meta-program, rather than statically like in this case.

5.4 Getting Values from Record Fields

Getting a value from a record field requires to define a module `Getter` containing a functor `GetField` that returns the type of the field that we need to get. This type will be used as the return type of the function`get` that is able to get that specific field. This function is also contained in this module and takes as argument the record from which we are getting the value and returns, as said above, the value of the field. The module `Getter` is built using the name of the field that is able to read and the record to read from.

```
Module "Getter" => (name : string) => (r : Record) : Getter { ... }
```

The functor `GetType` returns kind because in general the type of the field of a record is arbitrary. The function `get` uses in its declaration the functor `RecordType` to determine the type of the record to use and `GetType` to determine the type of the field to read. The complete module will look like

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

The getter has two implementations of the rule that instantiates the `Getter` module: one is used when the current field in the module tuple is the one we are trying to read, and the other that is able to build the correct getter if the field is in the remaining part of the record. The first happens when the name of the current field is the same as the field name provided as argument of the functor `GetField`. In this case we have the following rule for the functor:

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

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

  ---------------
  GetType => type

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

Listing 5.1: Getting a field (case 1)

The functor `GetType` simply returns the type of the current record field, because it is the correct field to read, and `get` returns the first element of the record tuple, which represents the value stored in the field itself.

When the field we are trying to read is not the one we are currently examining in the record tuple, we must build a getter functor that is able to recursively get the field from the remaining part of the record. At this purpose, we have to extend the `Getter` with an additional functor `GetAnotherField` that returns a module instance capable of reading the value from the correct field in the remaining part of the record and its type. The implementation of the rule for this case is the following:

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

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

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

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

Listing 5.2: Getting a field (case 2)

The rule for the functor `GetAnotherField` simply calls recursively the rule for `GetField` with the remaining part of the record. If the next field is the correct one then this time we will use the rule in Listing 5.1, otherwise the rule in Listing 5.2 will be re-applied until the correct field is reached. The functor `GetType` simply calls `GetAnotherField` to obtain the module instance necessary to get the field from the rest of the record, and then calls the functor `GetType` from that instance. Finally, the function `get` will again use `GetAnotherField` and then call the `get` function from the getter returned by `GetAnotherField` with the remaining part of the record tuple. This function call will return the desired value that will be used also as result of the current `get`.

Let us now consider again the physical body implemented with functors in Section 5.3 and let us assume that we want to get the value of the field `Position`. We define a function `getPos` that takes as input a physical body and returns `Vector2`. This function will use `GetField` in its premises to generate the getter for `Position` and will then call the `get` function from the generated module instance.

```
Func "getPos" : Vector2

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

Listing 5.3: Getter for the Position field

Note that in the code above we are using the functor `PhysicalBodyType` and the function `PhysicalBody` defined in Section 5.3. Now let us analyse step-by-step what happens when we call `getPos`. The first premise will call the functor `GetField` with

```
fieldName := "Position"
r := PhysicalBodyType := RecordField "Position" Vector2 (RecordField "
     Velocity" Vector2 EmptyRecord)
```

At this point the rule for `GetField` will deconstruct `RecordField` in its conclusion by means of pattern matching and set

```
name := "Position"
type := Vector2
r := RecordField "Velocity" Vector2 EmptyRecord
```

Since `name` = `fieldName` we fall in the case in Listing 5.1. Thus we instantiate the module `Getter` by setting the construction arguments to

```
name := "Position"
r := RecordField "Position" Vector2 (RecordField "Velocity" Vector2
     EmptyRecord)
```

In this module instance the functor `GetType` returns `type := Vector2` and `get` returns the first element of the record tuple. At this point, the third premise will call `get` from this module instance by passing the record tuple `((10.0,10.0),((10.0,0.0),()))`, which in turn returns correctly `(10.0,10.0)`.

Now let us assume that we want to retrieve the value of "Velocity" instead. We define an analogous function `getVel` as follows

```
Func "getVel" : Vector2

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

This time the functor `GetField` is called with

```
fieldName := "Velocity"
r := PhysicalBodyType := RecordField "Position" Vector2 (RecordField "
     Velocity" Vector2 EmptyRecord)
```

thus the rule in case 2 is triggered. This rule will generate an instance of `Getter`

```
fieldName := "Velocity"
type := Vector2
r := RecordField "Velocity" Vector2 EmptyRecord
```

This rule will generate a module containing the auxiliary functor `GetAnotherField` that is capable to retrieve the correct field in the remaining part of the record. The rule that processes `GetAnotherField` will call, in its premise, `GetField` with

```
fieldName := "Velocity"
r := RecordField "Velocity" Vector2 EmptyRecord
```

Since now the name of the field of the getter coincides with the name of the field in `RecordField`, this premise will now trigger the rule in Listing 5.2 that in turn generates an instance of `GetField` containing the following:

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

--------------------
GetType => Vector2


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

Listing 5.4: Getter for Velocity generated by `GetAnotherField`

This module instance will be the one returned by the rule of `GetAnotherField`. The rule of the functor `GetType` for `Velocity` will use `GetAnotherField` to retrieve the correct field type using the module instance generated in Listing 5.4 and return it (which is `Vector2`). The rule for the function `get` will use `GetAnotherField` to call the the `get` function from Listing 5.4 passing the second element of the record tuple as argument and returns the result of this function call.

Note that the module instantiation will be again performed at compile time, thus the only operations performed at runtime are the calls to the `get` functions contained in the module instantiations.

5.5 Setting Values of Record Fields

The setter module is analogous to the getter, except that this time the module must generate a function that, in addition to the record, takes as input the value to write in the field. This function returns a modified copy of the record tuple where the value associated to the field has been changed. For this purpose we need a module containing a functor `SetType` that returns the type of the field to set. This functor will be used to build the declaration of a function `set` that is able to set the specified field.

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

The declaration of the function `set` uses `r.RecordType` to define the type of the record argument, `SetType` to define the type of the field to set, necessary for the argument containing the value to set, and returns `r.RecordType`, which is the modified version of the record.

Analogously to the getter, we need a functor that instantiates `Setter` and has two different implementations of the instantiation rule: one where the field of the current element of the record tuple coincides with the one we want to set, and the other where the field is different and that is able to build a setter for the remaining part of the record tuple.

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

The first case is implemented as follows:

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

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

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

Listing 5.5: Setting a field (case 1)

The function `SetType` simply returns the type of the field in the current `RecordField`, while the rule for `set` replaces the first value in the record tuple with the new value. The second rule is implemented as follows

```
name <> fieldName
thisRecord := RecordField name type r
-------------------------------------
SetField fieldName (RecordField name type r) => Setter fieldName
     thisRecord {
  Functor "SetAnotherField" : Setter

  SetField fieldName r => setter
  ------------------------------
  SetAnotherField => setter

  SetAnotherField => s
  ------------------------
  SetType => s.SetType

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

Listing 5.6: Setting a field (case 2)

The functor `SetAnotherField` is an auxiliary functor that recursively calls `SetField` with the remaining part of the record. Eventually `SetField` will trigger the rule in Listing 5.5 when the correct field is encountered. This auxiliary functor is then used in `SetType` to retrieve the correct type of the record field and in the function `set` to call the correct `set` function for the field. The `set` function in the setter generated by `SetAnotherField` returns a modified version of the record that is replaced in the tuple.

Let us now consider again the physical body and assume that we want to define a setter for `Position`. Again we define a function `setPos` for the field:

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

SetField "Position" PhysicalBodyType => setter
PhysicalBody -> body
setter.set body -> body'
----------------------
setPos v -> body'
```

Again we are using the functor `PhysicalBodyType` and the function `PhysicalBody` defined in Section 5.3. The first premise of this rule will call `SetField` with

```
fieldName := "Position"
name := "Position"
type := Vector2
r := RecordField "Velocity" Vector2 EmptyRecord
```

which, in turn, instantiates `Setter` with

```
name := "Position"
r := RecordField "Position" Vector2 (RecordField "Velocity" Vector2
     EmptyRecord)
```

In this module instance `r.RecordType` will be `Tuple[Vector2,Tuple[Vector2,unit]]` and `SetType` returns `Vector2`. The whole instance will look like

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

------------------
SetType => Vector2


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

Now let us assume that we want to build a setter for `Velocity`. We have to define a function `setVel` as follows

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

SetField "Velocity" PhysicalBodyType => setter
PhysicalBody -> body
setter.set v -> body'
-------------------------
setVel v -> body'
```

The first premise of this rule will now invoke `SetField` with

```
fieldName := "Velocity"
name := "Position"
type := Vector2
r := RecordField "Velocity" Vector2 EmptyRecord
```

which will trigger the rule in Listing 5.6 instead. This will create an instance of `Setter` containing the auxiliary functor `SetAnotherField`. The rule for this functor will call in turn `SetField` with

```
fieldName := "Velocity"
name := "Velocity"
type := Vector2
r := EmptyRecord
```

that will generate a different instance of `Setter`, this time using the rule in Listing 5.5. The auxiliary setter will contain a functor `SetType` returning `Vector2` and a function `set` that inserts the value for `Velocity` in the record tuple. The `set` in the auxiliary setter will be used in the setter of `Velocity` to obtain the modified copy of the record containing the new value. Again

all the modules are generated at compile time, thus the only operations performed at runtime are the calls to the `set` functions of the field setter and eventual auxiliary modules.

5.6 Handling errors in getters and setters

In Section 5.4 and 5.5 we explained how to use functors to implement getters and setters for the fields of a record. The explanation however did not take into account possible mistakes that could be committed during the definition of setters and getters for a specific record.

A possible mistake that could arise in the process of defining getters and setters would be to provide an incompatible type for the `get` function of a field. For instance, let us assume that we define `getPos` as

```
Func "wrongGetPos" : double

GetField "Position" PhysicalBodyType => getter
PhysicalBody -> body
getter.get body -> v
-----------------
wrongGetPos -> v
```

As explained above the getter module will contain a function `get` that returns a `Vector2`, because that is the type of the field `Position`. In the process of building the module, this type is automatically retrieved from the definition of `PhysicalBodyType`. At this point the meta-compiler would report an error message because this wrong definition of `getPos` returns `double` but `get` returns a `Vector2`. Note that, as previously explained in Section 5.3, this is possible because functors are able to embed the type system of the embedded language into the Metacasanova type system.

Another possible mistake is accessing a field that is not defined for the record. For instance, let us assume that someone tries to build a getter for a field that does not exist in physical body, namely a field `Acceleration`. As usual we would define a function for the getter such as

```
Func "getAcc" : Vector2

GetField "Acceleration" PhysicalBodyType => getter
PhysicalBody -> body
getter.get body -> v
---------------
getAcc -> v
```

The first premise of this rule will call the functor `GetField` with

```
fieldName := "Acceleration"
name := "Position"
type := Vector2
r := RecordField "Velocity" Vector2 EmptyRecord
```

As seen above, this rule will generate an auxiliary module by recursively call `GetField` with

```
fieldName := "Acceleration"
name := "Velocity"
type := Vector2
r : = EmptyRecord
```

but at this point, since again `fieldName` $\neq$ `name` we will recursively call `GetField`. At this point we will fail to run a suitable rule for the functor, since the only two versions we have so far are able to process `RecordField` and not `EmptyRecord`, thus the pattern matching in the conclusion would fail. The meta-compiler will in any case fail to generate code, since the functor evaluation will fail and thus the whole code generation, but this approach is not "clean", since the meta-compiler will report a generic error regarding the rule execution failure. An alternative to this, is to include a case for the rule that processes `EmptyRecord`. A getter for an `EmptyRecord` returns `()` and `GetType` returns the type `unit`. This rule can be implemented as follows:

```
fieldName <> name
thisRecord := RecordField name type EmptyRecord
-------------------------------
GetField fieldName (RecordField name type EmptyRecord) => Getter
     fieldName thisRecord {

  ----------------
  GetType => unit

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

In this way the first premise of the rule of `getAcc` would generate a getter that takes a physical body and returns `unit`. When the rule calls this getter and returns `unit`, this will be incompatible with the return type of `getAcc` that should be a `Vector2`. The same can be done for `setAcc`, that is we extend the rule for `SetField` with an additional case:

```
fieldName <> name
thisRecord := RecordField name type EmptyRecord
--------------------------
SetField fieldName (RecordField name type EmptyRecord) => Setter
     fieldName thisRecord {

  ----------------
  SetType => unit

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

In this way, when invoking `set` in the premise `setAcc` the compiler will signal a type error because at some point it will try to use the `set` for the `EmptyRecord` with `Vector2`. Another alternative, which goes beyond the scope of this chapter, would be to allow rules in meta-casanova to output custom compilation errors. In this way we could write the very same rule case done above but this time we would return a compilation error message reporting that the field does not exist.

5.7 Evaluation

In the previous sections we presented an extension to the meta-language of Metacasanova that allows to embed the type system of an embedded language, whose definition is implemented in Metacasanova, in the meta-type

system of Metacasanova. We claimed that this would improve the runtime performance of a program written in the embedded language because we could get rid of all the dynamic checks and accesses to meta-data structures used to implement data structures in the embedded language, such as records. In this section we present the experimental evaluation that should produce the evidence to back up this claim. We proceed to describe the experimental set-up and then we comment the results.

5.7.1 Experimental Set-up

In this experiment we use the implementation of records with functors described in this chapter and we compare its runtime performance with its dynamic counter part, i.e. the implementation that uses dictionaries, that was used to implement the Casanova memory model in Section 4.2.2. The sample measures the run-time of both the functor implementation and the implementation with dynamic tables. We run the test by varying both the amount of record instances and the amount of fields per record. The test is run with a sample of 10000, 100000, and 1000000 record instances and with a number of fields from 1 to 10. We neglect to consider different field types, as the performance of look-up operations is not affected by the type of the fields themselves.

5.7.2 Results

In Table 5.4, 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 the implementation with functors is particularly effective for records with high number of fields. This is due to the fact that the runtime complexity of a dynamic table depends on the number of entries stored in it (which would be the fields in our case) and thus, when the fields are few, this number is very close to the complexity of the functorial implementation, which is constant. When the amount of fields increases, the performance of the functorial implementation remains the same while the dynamic table one worsens visibly. The constant complexity of the functorial implementation is due to the fact that the metacompiler builds the functions to look up for a specific field at compile time by generating the appropriate module instances with the functors. The `get` and `set` functions described above can either immediately get or set the value of the field and return the result of this operation, or call the getter or setter of an auxiliary module instance that is able to read or write the appropriate field. The only overhead is the overhead of chaining calls to Metacasanova functions, thus the overhead of creating and executing the code that implements the rule behaviour described in Section 3.6.

Figure 5.3 shows a chart of the overall performance of the two techniques (the data points are taken from Table 5.4). 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

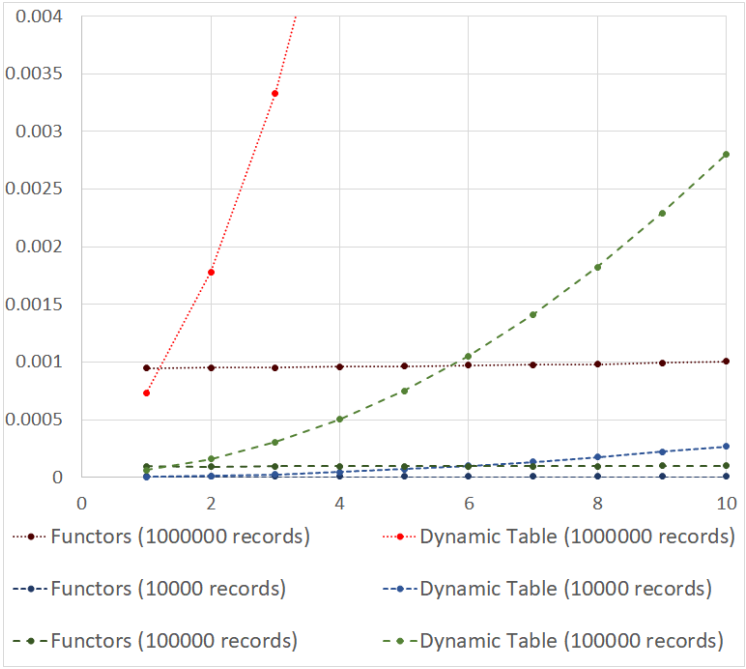


Figure 5.3: Execution time of the different memory models

updated. Moreover, note that the performance of the dynamic table is improved by the fact that we are using a dictionary implemented in .NET. 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 (which is not accounted for here, for example to prevent that a record is constructed with incompatible values for its fields), while this check is done at compile time with functors, thus drastically improving the performance

5.8 Summary

In this chapter we addressed the problem arisen in Chapter 4.3 about the performance of the generated code and the forced dynamic behaviour of languages implemented in Metacasanova. We started by informally state that this issue was due to the fact that it was not possible to embed the type system of a language in the meta-type system of Metacasanova, and this caused all the dynamic lookups and accesses at runtime. This could

Table 5.4: 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

be avoided by using a meta-language abstraction that allows both to define the type system of the embedded language in terms of the meta-type system of Metacasanova and to generate the code for the accesses to the data structures of the embedded language at compile time. At this purpose, we showed a language extension that provides such abstraction in terms of modules and functors. We then proceeded to provide an example of their usage in the context of record getters and setters for their fields. We then measured the performance gain by comparing the implementation with functors with the one using dynamic tables that was employed for the Casanova language implementation shown in Chapter 4.3. The results show that the performance of operations on records in the case of functors is faster than up to 30 times with respect to the dynamic table implementation. We have also shown that the performance of such operations in the case of functors is constant with respect to the number of fields to update, while the performance of the dynamic table drastically worsens when the number of fields in a record increase. In the next chapter we will show a further example of use of functors to re-implement Casanova semantics and extend the language with abstractions for the networking behaviour of multiplayer games.

Chapter 6

Networking primitives in Casanova

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 [31]. 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[65]) 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* [20]. 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" [35]), 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 [57]. 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 [16]. 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 [30] 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 [55] 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 [60] and Erlang [10]), 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

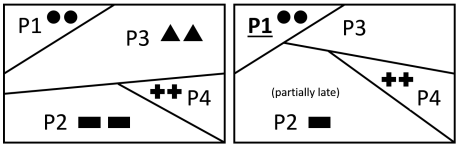
the state of the master instances without having to validate this state change by synchronizing with other players through the network.

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 [12].

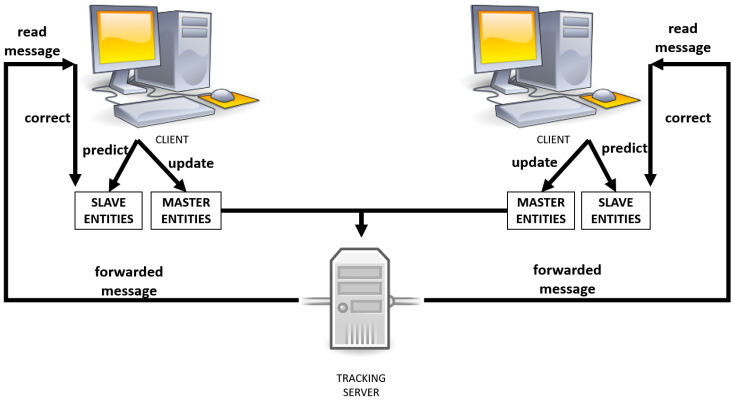


Figure 6.2: master/slave architecture

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.

Appendix A

List operations with templates

Appendix B

Metacasanova grammar in BNF

Bibliography

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

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

[3] M. Abbadi, F. Di Giacomo, R. Orsini, A. Plaat, P. Spronck, and G. Maggiore. *Resource Entity Action: A Generalized Design Pattern for RTS games*, pages 244–256. Springer, 2014.

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

[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 *Joint International Conference on Serious Games*, pages 123–134. Springer, 2015.

[6] Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi, Pieter Spronck, Costantini Giulia, and Giuseppe Maggiore. High performance encapsulation in casanova 2. In *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pages 201–206. IEEE, 2015.

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

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

[9] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, 2002.

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

[11] Andrea Asperti and Giuseppe Longo. *Categories, types, and structures: an introduction to category theory for the working computer scientist.* MIT press, 1991.

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

[13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[14] Michael Barr and Charles Wells. *Toposes, triples and theories*, volume 278. Springer-Verlag New York, 1985.

[15] Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

[16] 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.

[17] 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.

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

[19] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

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

[21] Krzysztof Czarnecki and Ulrich W Eisenecker. *Generative programming: methods, tools, and applications*, volume 16. Addison Wesley Reading, 2000.

[22] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *IFL*, pages 80–99. Springer, 2008.

[23] Francesco Di Giacomo, Mohamed Abbadi, Agostino Cortesi, Pieter Spronck, and Giuseppe Maggiore. Building game scripting dsl's with the metacasanova metacompiler. In *INTETAIN 2016*, pages 231–242. Springer, 2017.

[24] Francesco Di Giacomo, Mohamed Abbadi, Agostino Cortesi, Pieter Spronck, and Giuseppe Maggiore. Metacasanova: An optimized metacompiler for domain-specific languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 232–243. ACM, 2017.

[25] Atze Dijkstra, Jeroen Fokker, and S Doaitse Swierstra. The architecture of the utrecht haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 93–104. ACM, 2009.

[26] E. Dijkstra. Algol-60 translation. Technical report, Matematisch Centrum - Amsterdam, November 1961.

[27] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235 – 271, 1992.

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

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

[30] 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.

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

[32] Cordelia Hall, Kevin Hammond, Will Partain, Simon L Peyton Jones, and Philip Wadler. The glasgow haskell compiler: a retrospective. In *Functional Programming, Glasgow 1992*, pages 62–71. Springer, 1993.

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

[34] Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*, pages 97–136. Springer, 1995.

[35] 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.

[36] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[37] Stephen C Kleene. A theory of positive integers in formal logic. part ii. *American journal of mathematics*, 57(2):219–244, 1935.

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

[39] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

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

[41] Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

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

[43] Microsoft. F# language reference. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/let-bindings`, 2018.

[44] Microsoft. F# language reference. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions`, 2018.

[45] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses.* " O'Reilly Media, Inc.", 2013.

[46] Barry Mitchell. *Theory of categories*, volume 17. Academic Press, 1965.

[47] John C Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.

[48] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[49] Lawrence C Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

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

[51] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[52] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[53] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[54] Willard Van Orman Quine, Patricia S Churchland, and Dagfinn Føllesdal. *Word and object*. MIT press, 2013.

[55] 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.

[56] 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.

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

[58] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

[59] Simon Thompson. *Haskell: the Craft of Functional Programming . International Computer Science Series*. Addison-Wesley, March, 1999.

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

[61] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

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

[63] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[64] Stefan Wehr and Manuel MT Chakravarty. Ml modules and haskell type classes: A constructive comparison. In *Asian Symposium on Programming Languages and Systems*, pages 188–204. Springer, 2008.

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