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

1.1 Algorithms and problems

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

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

Pseudocode Pseudocode is a semi-formal language that might contain also statements expressed in natural language and omits system specific

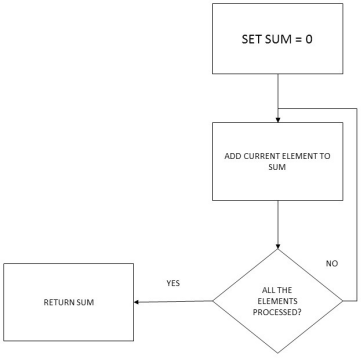

Figure 1.1: Flow chart for the sum of a sequence of numbers

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
```

1.2 Programming languages

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

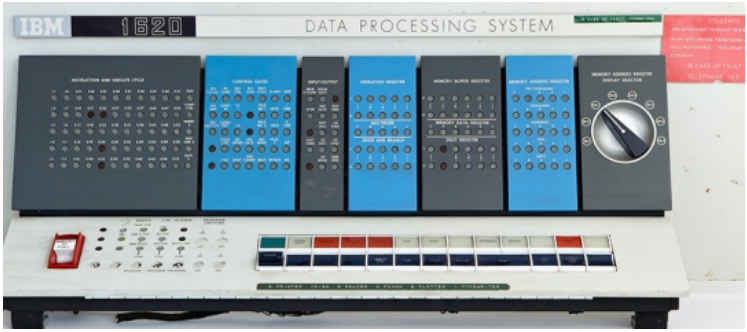

Figure 1.2: Front panel of IBM 1620

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 [...]:

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.

```
Here put an example of hexadecimal machine code generated using C++ (for
      example sum of an array of integers)
```

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.

```
Example of Assembly x86 code generated using c++
```

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

1.2.2 High-level programming languages

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

Imperative programming languages

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

```
Code for integer sequence sum in C
```

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

Declarative programming

Section layout

- Classification of programming languages.
- Motivation for programming languages.
- Understanding of programming languages

1.3 Compilers

- Origin of compilers.
- Reason of compilers.
- Requirements of compilers.

Connect to the previous section by explaining why compilers were born and how you design and build a compiler.

1.4 Meta-compilers

- History of meta-compilers
- Motivation behind meta-compilers.
- Requirements of meta-compilers.

Idea behind meta compilers. Introduction of the first META-languages family of meta-compilers, parser generators, then RML and other meta-compilers based on natural semantics. Main requirements of a meta-compiler.

1.5 Scientific relevance

Discuss about the few existing results in the field, and performance vs language customization issues (fixed rule representation, fixed symbol table structure, etc.).

1.6 Problem statement

Can we build a meta-compiler that is able to implement a language in less lines of codes than the hard-coded compiler, and at the same time have good performance?

Chapter 2

Background

2.1 Compilers

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

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

2.2 Meta-compilers

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

Describe what it is existing in literature.

Chapter 3

Meta-casanova

3.1 Meta-casanova language

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

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

3.2 Meta-casanova compiler architecture

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

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

3.3 Example language in the meta-compiler

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

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

3.4 Casanova 2

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

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

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

3.5 Casanova 2.5 in Meta-Casanova

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

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

Chapter 4

Meta-casanova optimization

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

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

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

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

```
TypeFunc EmptyRecord => Record

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1 Improve the performance of Metacasanova

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

4.2 Type functions

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

Explain the background about type functions.

4.3 Type functions in Meta-casanova

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

4.4 Type function interpeter

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

4.5 C– optimization

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

4.6 Casanova 2.5 optimization

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

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

Chapter 5

Networking primitives in Casanova 2

5.1 General idea

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

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

5.2 Syntax and semantics

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

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

5.3 Send and receive primitives in Casanova 2

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

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

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

5.4 Meta-compiler implementation of networking operators

Chapter 6

Final evaluation

6.1 Performance evaluation of unoptimized Metacasanova

Evaluation from the INTETAIN paper. Comparison with Python.

6.2 Performance evaluation of optimized Metacasanova

6.3 Networking evaluation

Chapter 7

Discussion and conclusion

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