

Metacasanova: a High-performance Meta-compiler for Domain-specific Languages

Francesco Di Giacomo

Abstract

Programming languages are at the foundation of computer science, as they provide abstractions that allow the expression of the logic of a program independent from the underlying hardware architecture. In particular scenarios, it can be convenient to employ Domain-Specific Languages, which are capable of providing an even higher level of abstraction to solve problems which are common in specific domains. Examples of such domains are database programming, text editing, 3D graphics, and game development. The use of a domain-specific language for the development of particular classes of software may drastically increase the development speed and the maintainability of the code, in comparison with the use of a general-purpose programming language. While the idea of having a domain-specific language for a particular domain may be appealing, implementing such a language tends to come at a heavy cost: as it is common to all programming languages, domain-specific languages require a compiler which translates their programs into executable code. Implementing a compiler tends to be an expensive and time-consuming task, which may very well be a burden which overshadows the advantages of having a domain-specific language.

To ease the process of developing compilers, a special class of compilers called “meta-compilers” has been created. Meta-compilers have the advantage of requiring only the definition of a language in order to generate executable code for a program written in that language, thus skipping the arduous task of writing a hard-coded compiler for the new language. A disadvantage of meta-compilers is that they tend to generate slow executables, so they are usually only employed for rapid prototyping of a new language. The main aim of this thesis is to create a meta-compiler which does not suffer from the disadvantage of inefficiency. It presents a meta-compiler called “Metacasanova”, which eases the development cost of a compiler while simultaneously generating efficient executable code.

The thesis starts by analysing the recurring patterns of implementing a compiler, to define a series of requirements for Metacasanova. It then explains the architecture of the meta-compiler and provides examples of its usage by implementing a small imperative language called C--, followed by the reimplementing of a particular, existing domain-specific

language, namely Casanova, which has been created for use in game development. The thesis presents a novel way to optimize the performance of generated code by means of functors; it demonstrates the effect of this optimization by comparing the efficiency of Casanova code generated with and without it. Finally, the thesis demonstrates the advantages of having a meta-compiler like Metacasanova, by using Metacasanova to extend the semantics of Casanova to allow the definition of multiplayer online games.

Contents

1	Introduction	1
1.1	Algorithms and problems	2
1.2	Programming languages	4
1.2.1	Low-level programming languages	4
1.2.2	High-level programming languages	6
1.2.3	General-purpose vs Domain-specific languages	8
1.3	Compilers	11
1.4	Meta-compilers	12
1.4.1	Requirements	12
1.4.2	Benefits	13
1.4.3	Scientific relevance	14
1.5	Problem statement	15
1.6	Thesis structure	16
2	Background	17
2.1	Architectural overview of a compiler	18
2.2	Lexer	19
2.2.1	Finite state automata for regular expressions	20
2.2.2	Conversion of a NFA into a DFA	21
2.3	Parser	22
2.3.1	LR(k) parsers	24
2.3.2	Parser generators	26
2.3.3	Monadic parsers	30
2.4	Type systems and type checking	34
2.5	Semantics and code generation	36
2.6	Metaprogramming and metacompilers	38
2.6.1	Template metaprogramming	38
2.6.2	Metacompilers	40
2.7	Differences with Metacasanova	45
2.8	Summary	46

3	Metacasanova	49
3.1	Repetitive steps in compiler development	49
3.1.1	Hard-coded implementation of type rules	50
3.1.2	Hard-coded implementation of Semantics	56
3.1.3	Discussion	60
3.2	Metacasanova overview	60
3.2.1	Requirements of Metacasanova	61
3.2.2	Program structure	61
3.2.3	Formalization	64
3.3	Architectural overview	65
3.4	Parsing	66
3.4.1	Declarations	66
3.4.2	Rules	68
3.4.3	Parser post-processor	70
3.5	Type checking	76
3.5.1	Checking declarations	76
3.5.2	Checking rules	79
3.6	Code generation	83
3.6.1	Meta-data structures code generation	84
3.6.2	Code generation for rules	84
3.7	Summary	89
4	Language Design in Metacasanova	91
4.1	The C-- language	91
4.1.1	Expression Semantics	92
4.1.2	Statement Semantics	95
4.1.3	Type Checker	101
4.1.4	Discussion	109
4.2	The Casanova language	109
4.2.1	The structure of a Casanova program	109
4.2.2	Casanova semantics in Metacasanova	110
4.2.3	Rule update	111
4.2.4	Rule evaluation	115
4.2.5	Statement evaluation	117
4.3	Evaluation	120
4.3.1	Experimental Set-up	121
4.3.2	Performance	122
4.3.3	Discussion	122
4.4	Summary	127
5	Metacasanova Optimization	129
5.1	Language extension idea	129
5.1.1	Field access in Casanova	130
5.1.2	Inlining the entity fields	133
5.2	Modules and Functors	134

CONTENTS

5.2.1	Language Extension	135
5.3	Record implementation with modules	136
5.4	Getting Values from Record Fields	141
5.5	Setting Values of Record Fields	145
5.6	Handling errors in getters and setters	148
5.7	Evaluation	151
5.7.1	Experimental Set-up	151
5.7.2	Results	151
5.8	Summary	155
6	Language Design with Functors	157
6.1	Casanova entity update	157
6.2	Update in Metacasanova	159
6.3	Updater Modules	160
6.4	Updatable elements	161
6.5	Updatable Fields and Records	164
6.6	Physical Body Simulation with Functors	166
6.7	Interruptible rules with functors	170
6.7.1	Multiple rules updating the Same Field and Local variables	174
6.8	Evaluation	177
6.8.1	Experimental Setup	177
6.8.2	Results	177
6.9	Summary	179
7	Networking in Casanova	181
7.1	Multi-player Support in Games	182
7.2	Motivation for a Language-based Solution	183
7.3	Related work	184
7.4	The master/slave network architecture	185
7.5	Case study	187
7.6	Implementation	188
7.7	Networking Primitives with Functors	193
7.7.1	Network Record	193
7.7.2	Connection	194
7.7.3	Local and Remote Entities	196
7.8	Evaluation	197
7.8.1	Experimental setup	198
7.8.2	Performance Evaluation	198
7.8.3	Code Size	199
7.8.4	Compiler Implementation Code Size	199
7.9	Summary	199

8	Discussion and Conclusion	201
8.1	Answer to research questions	201
8.1.1	Ease of development	201
8.1.2	Performance trade-off	202
8.1.3	Optimization	203
8.2	Answer to the problem statement	204
8.3	Future Work	205
A	List Operations with Templates	207
A.1	Element Getter	207
A.2	Element Existence	208
B	Metacasanova Grammar in BNF	211