WWU
MÜNSTER

An Integrated Constraint-Logic and Object-Oriented Programming Language

Jan Christoph Dageförde

Münster, 2020

Inauguraldissertation zur Erlangung des akademischen Grades eines Doktors der Wirtschaftswissenschaften durch die Wirtschaftswissenschaftliche Fakultät der Westfälischen Wilhelms-Universität Münster

An Integrated Constraint-Logic and Object-Oriented Programming Language

The Münster Logic-Imperative Language

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
durch die Wirtschaftswissenschaftliche Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Jan Christoph Dageförde

Münster, 2020

D6

Dekanin Prof. Dr. Theresia Theurl

Berichterstatter Prof. Dr. Herbert Kuchen
Prof. Dr. Markus Müller-Olm

Datum der Disputation 07. Juli 2020

Abstract

Sometimes, business applications perform constraint-logic search, e. g., for finding solutions to planning problems. Most business applications are written in object-oriented programming languages that are not particularly suited for search applications. In contrast, logic and constraint-logic programming languages offer useful features for search. However, such programming languages are less suited for the development of arbitrary (business) software. Consequently, there currently is a gap that established programming languages can only bridge by using impractical solutions. Intending to improve this situation, this work describes the multi-paradigm programming language Muli (short for the **Mü**nster **L**ogic-**I**mperative Language). Muli is based on Java, adding support for constraint-logic features. Most notably, Muli offers logic variables, symbolic execution, and encapsulated search. It is accompanied by a compiler and a sophisticated runtime environment. So far, Muli has been successfully applied to several domains, namely to logistics, the development of neural networks, and classical search problems.

Contents

List of Figures VII

List of Tables XI

List of Listings XIII

List of Acronyms XVII

I Research Overview 1

1 Exposition 3
- 1.1 Motivation . . . 3
- 1.2 Problem Statement . . . 5
- 1.3 Research Methodology . . . 6
- 1.4 Dissertation Outline . . . 8

2 Foundations 9
- 2.1 Logic Programming and Non-Deterministic Execution . . . 9
- 2.2 Constraint-Logic Programming . . . 13
- 2.3 Symbolic Execution of Java Bytecode . . . 16
- 2.4 Constraint Solving in Object-Oriented Languages . . . 19
- 2.5 Concepts from Declarative Programming in Imperative Languages . . 20

3 The Münster Logic-Imperative Language as a Multi-Paradigm Language 23
- 3.1 Design Principles . . . 23
- 3.2 Language Features of Muli . . . 25
- 3.3 The Muli Compiler . . . 30
- 3.4 The Muli Runtime Library . . . 34
- 3.5 Applications of Muli . . . 37
- 3.6 Summary . . . 40

4 Non-Deterministic Execution of Constraint-Logic Object-Oriented Applications 43
- 4.1 A Virtual Machine for Constraint-Logic Object-Oriented Programs . 43
- 4.2 Representation of Symbolic Expressions . . . 46

4.3 The Solver Component 48
4.4 A Structure that Encodes Non-Deterministic Execution Paths 50
4.5 Making Side Effects of Imperative Execution Reversible 54
4.6 Continuous Testing and Integration of the MLVM 58
4.7 Summary 59

5 Search in a Constraint-Logic Object-Oriented Language 61
5.1 Interrupting and Resuming Search 61
5.1.1 Using Dual Trails for the Retrieval of Individual Solutions . . 62
5.1.2 Obtaining Individual Solutions in Muli Applications 66
5.2 Search Strategy Selection at Runtime 68
5.3 Summary 73

6 Free Objects 75
6.1 Reference Types 75
6.2 Types of Interaction with Free Objects 76
6.2.1 Instantiation and Initialization 77
6.2.2 Field Access 78
6.2.3 Method Invocation 78
6.2.4 Type Check and Type Cast 81
6.2.5 Equality 82
6.3 Summary 85

7 Conclusion 87
7.1 Contributions 87
7.2 Limitations 89
7.3 Perspectives for Future Research 89

References 91

II Included Publications 101

8 Publication Overview 103

9 A Compiler and Virtual Machine for Constraint-Logic Object-Oriented Programming 105
9.1 Motivation 106
9.2 Muli Language 107
9.2.1 Language Concepts 108
9.2.2 Syntactic Extension of Java 112
9.2.3 Muli Classpath 113
9.3 A Future-Proof Muli Compiler 113
9.4 Operational Semantics of Muli Programs 115
9.4.1 Semantics of Expressions 119

9.4.2 Semantics of Statements 120
9.4.3 Evaluation of an Example Program 122
9.5 A Backtracking, Symbolic Virtual Machine 124
9.5.1 Data Structures 126
9.5.2 Symbolic Types 127
9.5.3 Solver Component 128
9.5.4 Symbolic Execution, Encapsulated Search, and Choice Points 128
9.6 Discussion 132
9.7 Related Work 140
9.8 Conclusions and Future Work 143
References 144

10 Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming 149
10.1 Motivation 150
10.2 Constraint-Logic Object-Oriented Programming 150
10.3 Generation of Graph Structures for Neural Networks 153
10.3.1 Generating Neural Network Graph Structures from a Muli Application 154
10.3.2 Using Generated Neural Networks to Solve the Pole Balancing Problem 157
10.3.3 Experiments 159
10.4 Solving a Dynamic Scheduling Problem with Constraint-Logic Object-Oriented Programming 161
10.5 Related Work 162
10.6 Conclusion and Outlook 164
References 164

11 Free Objects in Constraint-Logic Object-Oriented Programming 169
11.1 Programming with Free Objects 170
11.2 Constraint-Logic Object-Oriented Programming with Muli 172
11.2.1 Setting the Stage for Free Objects 175
11.3 Method Invocations on Free Objects 180
11.4 Field Access on Free Objects 183
11.5 Other Operations on Free Objects 184
11.5.1 Type Operations 184
11.5.2 Equality 186
11.6 Demonstration 187
11.7 Related Work 190
11.8 Concluding Remarks 192
References 196

12 Structured Traversal of Search Trees in Constraint-Logic Object-Oriented Programming 199
12.1 Motivation 200
12.2 Constraint-Logic Object-Oriented Programming 201
12.3 Muli Logic Virtual Machine 203
12.4 Search Trees 205
12.4.1 Representation 206
12.4.2 Construction 206
12.4.3 Traversal 208
12.5 Search Strategies 209
12.6 Discussion 211
12.7 Related Work 214
12.8 Conclusion and Future Work 216
References 217

13 Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space 219
13.1 Motivation 220
13.2 Constraint-Logic Object-Oriented Programming 221
13.3 Infinite or Large Search Spaces 222
13.4 Individual Retrieval of Solutions from Encapsulated Search 224
13.4.1 Copy-Based Backtracking 225
13.4.2 Trail-Based Backtracking 228
13.5 Full Backtracking in the SJVM Using an Inverse Trail 232
13.6 Evaluation 235
13.7 Related Work 238
13.8 Conclusions and Future Work 239
References 241

14 Reference Type Logic Variables in Constraint-logic Object-oriented Programming 245
14.1 Motivation 246
14.2 Constraint-Logic Object-Oriented Programming with Muli 248
14.3 Reference Type Logic Variables (or Free Objects) 249
14.3.1 Accessing a Field of a Free Object 251
14.3.2 Invoking a Method on a Free Object 252
14.3.3 Comparing Reference Equality of Reference Type Logic Variables 253
14.3.4 Comparing Value Equality of Reference Type Logic Variables 254
14.3.5 Performing Type Operations on a Free Object 255
14.3.6 Imposing a Constraint for Structural Equality between Two Objects 256
14.4 Implementation 257
14.5 Related Work 258
14.6 Concluding Remarks 259

References 259

15 A Constraint-Logic Object-Oriented Language 263
15.1 Motivation 264
15.2 Muli Language 265
15.2.1 Language Concepts 266
15.2.2 Syntactic Extension of Java 269
15.2.3 Muli Classpath 270
15.2.4 Implementing a Compiler for Muli 270
15.3 A Backtracking, Symbolic VM 271
15.4 Discussion 277
15.5 Related Work 281
15.6 Conclusions and Future Work 283
References 284

16 An Operational Semantics for Constraint-Logic Imperative Programming 289
16.1 Introduction 290
16.2 Language Concepts 290
16.3 A Non-Deterministic Operational Semantics of Muli 293
16.3.1 Semantics of Expressions 295
16.3.2 Semantics of Statements 296
16.4 Example Evaluation 299
16.5 Discussion 300
16.6 Related Work 303
16.7 Conclusions and Future Work 304
References 306

Curriculum Vitae 309

List of Publications 311

List of Figures

Figure 1.1 Relationships between DSRM steps and the structure of this dissertation. 7

Figure 2.1 Rectangles r_1 and r_2 must intersect. 13
Figure 2.2 Comparison of the effects of non-branching instructions on the operand stack. 17
Figure 2.3 A symbolic execution tree, with branch constraints denoted at the edges. 18

Figure 3.1 Search trees that represent non-deterministic search for the search region specified in Listing 3.3. 27
Figure 3.2 Deterministic main execution control flow of a Muli application. Parts that are executed non-deterministically are restricted to encapsulated search. Results are collected and returned to the surrounding application. 29
Figure 3.3 Attribute structures that are generated into the compiled bytecode. Types used in this figure follow the JVMS [Lin+15], where u*n* signifies an *n*-byte unsigned integer and `CONSTANT_Utf8_info` is a string constant. 33
Figure 3.4 Public API offered by the Muli runtime library. 34

Figure 4.1 Components of the MLVM. 44
Figure 4.2 Class structure for the representation of symbolic expressions in the MLVM. 46
Figure 4.3 Effects of the `Iadd` instruction on the execution state. 47
Figure 4.4 Interface and implementations of solver managers that integrate libraries in the solver component. 49
Figure 4.5 MLVM class structure for the representation of search trees. 52
Figure 4.6 Intermediate stages of the construction of the search tree for the search region from Listing 4.1, assuming a depth-first search strategy. An edge label specifies the constraint of the corresponding subtree. . . 55
Figure 4.7 Expected search tree for `FailCoin`, as checked by the JUnit test in Listing 4.2. 58

Figure 5.1 Processing the backward trail in order to establish a former execution state while creating the corresponding forward trail. 63
Figure 5.2 Using backward trails (dashed) and forward trails (dotted) in order to achieve specific execution states. 64

Figure 5.3 Relationships between the stream-related classes of the Java Platform SE API and the encapsulated search operator `Muli.muli()`. 67
Figure 5.4 Simplified class structure showing the search strategy implementations and their relation to the VM class. 69
Figure 5.5 Efficient navigation between two nodes in arbitrary subtrees is possible via the closest common ancestor (blue path) instead of navigating via the root node (red path). 71

Figure 6.1 Application class structure comprising an interface and four implementations. 76
Figure 6.2 Types that a free object with the declaration `A a free` may assume, before and after choosing a method implementation for invocation. 80
Figure 6.3 Partial search tree created from the invocation of `a.m()` in the example from Figure 6.2. 81
Figure 6.4 Representation of type constraints in the solver component of the MLVM. 81
Figure 6.5 Search trees created from the invocation of `equals()` on free or regular objects. 84

Figure 9.1 Attribute structures generated into the compiled bytecode. Types used in this figure follow the specification of [Lin+15], where un signifies an n-byte unsigned integer and `CONSTANT_Utf8_info` is a string constant. 115
Figure 9.2 Components structure of the Muli runtime environment. 125
Figure 9.3 Effect of backtracking on execution state. As a result of backtracking, the operand stack, trail stack, constraint stack, and program counter have changed (highlighted blue in the web version of this article). Operand stack elements with dashed lines reside on the heap, which we omitted for simplicity. 130
Figure 9.4 Excerpt from the search tree that is effectively generated by executing the search region depicted in Listing 9.2. 131
Figure 9.5 Unrestricted symbolic execution versus encapsulated symbolic execution. 132
Figure 9.6 Comparison of sample implementations by mean execution time (in milliseconds), each averaged over 500 executions. 134
Figure 9.7 Comparison of implementations solving the n-Queens problem with increasing n by mean execution time (in milliseconds), each averaged over 500 executions. 135

Figure 10.1 The pole balancing problem as simulated by the CartPole-v1 implementation from OpenAI. 153
Figure 10.2 Feed-forward neural networks for solving the pole balancing problem. 154
Figure 10.3 Class structure that models our logistics planning problem. 162

Figure 11.1 Class structure assumed for the running example. 171

Figure 11.2 Conceptual structure of the MLVM. Adapted from [DK19] and updated in order to reflect recent developments. 174
Figure 11.3 Class definitions. With a recursive definition, e. g., for `Rec`, exhaustive generation of concrete objects does not terminate. 177
Figure 11.4 Example for an object structure that forms a ring. 178
Figure 11.5 Applicable instance types for a given object `A a free` before and after choosing a specific implementation. 183
Figure 11.6 Execution trees created as a result of calling `equals()` on free (Variation 1) or non-free (Variation 2) objects. 187
Figure 11.7 Object-oriented representation of board and queens. 189
Figure 11.8 Representation of graph modification operations in a class structure for the purpose of non-deterministic choice. 190

Figure 12.1 Class diagram for the representation of search trees. 207
Figure 12.2 Different evaluation stages of the search tree corresponding to the search region in Listing 12.2. The constraint of each subtree is noted at the respective edge. 208
Figure 12.3 Comparison of execution times in MLVM with or without explicit search trees, both using depth-first search. Execution times in PAKCS for reference. 212

Figure 13.1 Simplified symbolic execution tree for the program in Listing 13.1 with conditional branching. 223
Figure 13.2 Placement of savepoints within the symbolic execution tree. . . . 225
Figure 13.3 Effects on trail and inverse trail resulting from backtracking a choice point. 232
Figure 13.4 UML class diagram conceptualising the relationships between Java Stream API and the Muli implementation for retrieving individual solutions from encapsulated search. 233
Figure 13.5 Comparison of execution time needed to solve search problems, each averaged over 500 executions. 237

Figure 14.1 Class structure assumed for the running example. 247
Figure 14.2 Applicable instance types for a given object `A a free` before and after choosing a particular subtype. 253

Figure 15.1 Components structure of the Muli runtime environment. 272
Figure 15.2 Effect of backtracking on execution state. Changes highlighted in blue. Operand stack elements with red dashed lines are on the heap, which we omitted for simplicity. 276
Figure 15.3 Unrestricted symbolic execution versus encapsulated symbolic execution. 276

List of Tables

Table 4.1 Types of boolean expression constraints that can be generated as a result of evaluating certain Bytecode instructions. <cond> is substituted with either eq, ne, lt, le, gt, or ge. 48
Table 4.2 Bytecode instructions whose execution potentially results in non-deterministic branching. <cond> is substituted for specific comparisons, e. g., eq for equality. 51
Table 4.3 Trail element types and their respective parameters. 57

Table 5.1 Trail element types and their respective inverses. 64
Table 5.2 Comparison of search strategies using two search regions, w. r. t. the average number of solutions that are found and returned within ten seconds. 73

Table 9.1 Bytecode instructions, resulting choice points, and applicable constraint types. <cond> is one of eq, ne, lt, le, gt, or ge. 129
Table 9.2 Lines of code (LOC) required for implementing the experiments, not counting lines that are empty or comments. 139

Table 10.1 Graph structures generated before the smallest neural network that solves the problem is found. For each network, the time spent on its generation (in milliseconds) and training (in seconds) are indicated as well as its fitness. 160
Table 10.2 Times spent on generating (in milliseconds) and training (in seconds) the first 15 generated large neural networks that were able to solve the problem. 161

Table 12.1 Bytecode instructions that may cause non-deterministic branching upon execution. <cond> is a placeholder for specific comparisons, e. g., eq for equality. 204
Table 12.2 Comparison of search strategies w. r. t. the number of solutions that are returned within ten seconds. 213

Table 13.1 Subset of bytecode instructions that may introduce non-determinism if they involve logic variables. <cond> is one of eq, ne, lt, le, gt, or ge. 224
Table 13.2 Trail elements, representing inverse operations to reverse previous SJVM state changes in trail-based backtracking approaches. 228

Table 15.1 Bytecode instructions, resulting choice points, and applicable constraint types. `<cond>` is one of `eq`, `ne`, `lt`, `le`, `gt`, or `ge`. 275
Table 15.2 Comparison of sample implementations by execution time (in milliseconds). 278

List of Listings

2.1 Specification of facts in a Prolog program. 10
2.2 Rule definition in a Prolog program. 10
2.3 In Curry, overlapping function definitions result in non-deterministic evaluation. 12
2.4 CLP(FD) program that searches for intersecting rectangles. 15

3.1 Valid declarations for free variables that have a primitive type. 25
3.2 A demonstration of how free and regular variables can be used interchangeably. 25
3.3 Abstract search region, demonstrating the specification of constraints and solutions. 26
3.4 Muli code that searches for an integer *e* that can be expressed in two different ways as the sum of two positive integer cubes. 28
3.5 A search region, i. e. a method that formulates a search problem, can be passed to an encapsulated search operator using a method reference. . . 30
3.6 Implementing the modification to the production rules using ExtendJ's parser generator. 32
3.7 Declaring Muli's new AST node types for the ExtendJ framework. 32
3.8 Changing the MLVM execution mode before and after search. 35
3.9 Several encapsulated search operators are convenience methods that anticipate frequent ways of accessing the results of search, processing the stream returned by `Muli.muli()` in predefined ways. 37
3.10 Muli application that returns a solution for the n-Queens problem (here, $n = 8$). 38
3.11 Search region from `NNGenerator` that non-deterministically selects graph operations in order to generate graph structures systematically. 39

4.1 Muli search region example that comprises three solutions and a failure. 54
4.2 JUnit test case that checks whether the MLVM executes a Muli application as expected. 59

5.1 Muli code excerpt that uses non-deterministic evaluation in order to generate all powers of two for non-negative integer exponents. 62
5.2 Navigating upwards in a search tree. 65
5.3 Navigating downwards in a search tree. 66
5.4 Calculation of the closest common ancestor of two nodes. 71
5.5 Search region that theoretically produces a search tree of infinite depth. 72

6.1 Search region that invokes a method on a free object. 76
6.2 Discovering the set of method implementations that are candidates for invocation. 79
6.3 Excerpt from a search region that introduces non-determinism while checking for value equality in two variations. 83
6.4 Implementation of `Rectangle.equals()`. 83

9.1 Muli program that searches the (integer) square root of 5 and prints the result (class header omitted). 110
9.2 Muli program that searches factorials using non-deterministic evaluation and prints the first 100 of them (class header omitted). 110
9.3 Incrementally adding constraints from user input in Muli. 111
9.4 Declaration of additional AST subtypes for the Muli compiler. 115
9.5 Muli program that non-deterministically searches for a solution 2^a where a integer, $a \geq y$ for a given y. 123
9.6 Muli search region implementing the Send More Money Puzzle; class headers and helper functions omitted. 133
9.7 Modification of the factorials search region from Listing 9.2 in order to generate JUnit assertions for testing `fact()`. 136
9.8 Implementation of adding constraints from user input incrementally and of manual backtracking requires more effort in Java (in combination with the JaCoP solver) than in Muli. 137
9.9 Implementation the Send More Money problem in Java (in combination with the JaCoP solver) also requires more effort than in Muli. 138

10.1 Muli search region that calculates the Hardy-Ramanujan number. 151
10.2 Muli search region that systematically generates graph structures by non-deterministic selection of operations. 156
10.3 Processing the solution stream in Muli. 157
10.4 Structure of the Python program as generated by the `NNGenerator` Muli application. Note that the constructor parameters of `ENN` are shown exemplarily; they need to be substituted according to a specific configuration. 158
10.5 Python class `ENN` that creates hidden layers dynamically from the constructor parameters. 159
10.6 Muli code snippet to dispatch orders to trucks with non-deterministic search, modelling weight and volume constraints. 163

11.1 Excerpt from a constraint-logic object-oriented program that invokes a method on a free object. 170
11.2 Example that demonstrates symbolic evaluation of expressions that contain logic variables. 173
11.3 A method that checks whether an object `o` contains a ring structure, such as the one from Figure 11.4. 178
11.4 Using non-deterministic choice for generating ring structures of arbitrary length, such as the one in Figure 11.4. 179

11.5 A search region that branches over the types of a free object and returns the selected classes' names. 180
11.6 Subclasses can hide fields of their supertypes, but fields are never overridden. 184
11.7 Using type operations on a free object. 185
11.8 Example program involving non-determinism in the check for value equality. 186
11.9 n-Queens search region that makes use of object-oriented features for the implementation of a search problem. 188
11.10 Search region that generates directed acyclic graphs using non-deterministic method invocation on a free object. 190

12.1 A simple non-deterministic search region in Muli for the demonstration of constraint-logic object-oriented programming concepts. 200
12.2 Muli search region example that comprises two solutions and a failure. . 200
12.3 Bytecode generated by the Muli compiler for the program in Listing 12.2. 204
12.4 Methods for navigating upwards and downwards in a search tree. 209
12.5 Algorithm for finding the first common ancestor of two nodes. 210
12.6 Muli search region featuring an infinite amount of execution paths. . . . 213

13.1 Muli search region generating 2^y for all integer $y \geq 0$. 220
13.2 Modified `powersOfTwo` method using the new interface. Guaranteed to terminate after computing at most 10 solutions. 235

14.1 A constraint-logic object-oriented program that involves a free object. . 247
14.2 Arithmetic expressions containing bound or unbound variables. 248
14.3 Fields are only hidden, but not overridden. 251
14.4 Declaration of a set of reference type variables. 254

15.1 Muli program that searches the (integer) square root of 5 and prints the result 2 (class header omitted). 267
15.2 Muli program that searches factorials non-deterministically and prints the first 100 of them (class header omitted). 268
15.3 Iterative addition of constraints from user input in Muli. 269
15.4 Muli search region implementing the Send More Money Puzzle; class headers and helper functions omitted. 278
15.5 Modification of the factorials search region from Listing 15.2 to generate JUnit assertions for testing `fact()`. 279
15.6 Implementation of adding constraints from user input incrementally and of manual backtracking requires more effort in Java (using the JaCoP solver) than in Muli. 280

16.1 Non-deterministic computation of the logarithm of a number to the base 2 using (core) Muli. 292
16.2 Demonstration of the limits of constraint propagation using an example in Prolog+CLP(FD). 301

16.3 Minimal example demonstrating that variables may be mutated directly, in contrast to results of their uses: After evaluation, y is 5. 302

List of Acronyms

API Application programming interface
AST Abstract syntax tree

CLOOP Constraint-logic object-oriented programming

DSRM Design Science Research Methodology

FD Finite domain

GPL General Public License

IDE Integrated development environment
IS Information systems

JLS Java Language Specification
JVM Java Virtual Machine
JVMS Java Virtual Machine Specification

MLVM Muli Logic Virtual Machine
Muconst The Münster Constraint Solving Toolkit
Muggl The Münster Generator of Glass-Box Test Cases
Muli The Münster Logic-Imperative Language

PC Program counter

SMT Satisfiability Modulo Theories

VM Virtual machine

WAM Warren Abstract Machine

Part I

Research Overview

1 Exposition

This chapter sets the stage for this dissertation. In the beginning, Section 1.1 motivates the development of an integrated programming language for object-oriented and constraint-logic programming. Subsequently, Section 1.2 states the challenge approached by this research, followed by a description of the research design in Section 1.3. Finally, Section 1.4 outlines the structure of the dissertation.

1.1 Motivation

In contemporary software development, object-oriented programming languages are ubiquitous. Throughout the past years, object-oriented programming languages such as Java, C#, and C++ have dominated in popularity rankings [TIO20; Sta17; Sta18; Sta19]. This is not surprising, seeing that object-oriented programming offers useful features such as inheritance as well as encapsulation of data and behaviour. These features contribute towards structure, maintainability, and re-usability of software artefacts [Lou93]. Moreover, the widespread usage of these programming languages yielded a lot of software components that developers can embed into their own applications, thus making object-oriented programming languages even more useful [Mvn20]. The availability of these software components contributes to the dominant position of object-oriented programming in software development.

Yet, there are application scenarios in which the use of other paradigms would provide relevant benefits. Specifically, business software that occasionally solves search problems, such as route or production planning, can benefit from constraint-logic programming. As a specialization of declarative programming, languages from the constraint-logic paradigm enable the declarative specification of a search problem and rely on solver algorithms of the runtime environment that will implicitly find solutions for the problem [FA03].

For instance, consider a simplified dynamic scheduling problem of a logistics company that transports goods using a set of trucks. A transport order $o_i = (w_i, v_i)$ has a specific weight w_i and volume v_i depending on the goods that are transported. Moreover, every

truck $T_j = (W_j, V_j)$ has a maximum capacity w. r. t. weight W_j and volume V_j. With any given assignment A of orders to trucks, where $A(o_i) = T_j$, the following constraint enforces that the assignment does not exceed the trucks' capacities:

$$\left(\sum_{i \in \{i \mid A(o_i) = T_j\}} w_i\right) \leq W_j \land \left(\sum_{i \in \{i \mid A(o_i) = T_j\}} v_i\right) \leq V_j \quad \forall j$$

A constraint solver can find a solution to this problem, i. e., a feasible assignment. As soon as a solution is found, business software is responsible for putting the solution into effect by communicating loading orders to the respective truck drivers. Developing the search problem in a constraint-logic programming language benefits from the possibility to declaratively specify the problem, whereas the communication of loading orders is more intuitively implemented using an imperative (object-oriented) programming language. Moreover, in the context of a long-running business application, the scheduling problem in this example is a dynamic one as new transport orders arrive at runtime, thus imposing additional constraints that require an adapted solution. Consequently, a software for this scenario would frequently switch between non-deterministic search and deterministic execution. Therefore, this scenario and similar ones would benefit from the possibility to seamlessly interleave object-oriented and constraint-logic application parts.

As a consequence, it is desirable to integrate facilities for constraint-logic search into an object-oriented programming language. There are approaches that intend to achieve such an integration, but they come with limitations. One option is a cross-language integration, e. g., combining a Java business application with a Prolog program that is executed in Prolog using the Java Native Interface [KO08]. However, there are semantic mismatches between the programming languages, making this approach fragile and error-prone [KO08]. Another option is to use one of several constraint-solver libraries, e. g., OptaPlanner [The17], JaCoP [Kuc03], or Choco [PFL16] for Java. However, these libraries do not share a common interface that would make them interchangeable. In the worst case this results in a lock-in effect once a library has been selected. In the past there has been at least one initiative that aimed at standardizing a solver library interface. However, that initiative's efforts have ceased [Fel12].

In summary, even though object-oriented and constraint-logic programming both provide complementary benefits for the development of business software, there is no convenient, integrated, safe, and future-proof way to use constraint-logic features from software that is implemented in an object-oriented language. Ideally, a programming language would provide integrated facilities for the development of such software.

1.2 Problem Statement

Due to the absence of an integrated programming language for the development of software that would benefit from object-orientation as well as from constraint-logic programming, development of such software requires unnecessary effort. An ideal integrated programming language would be primarily object-oriented in order to be appealing as a mainstream programming language for business contexts, while treating constraint-logic features as first-class citizens. Specifically, logic variables should be syntactically similar to regular variables and constraints should be expressed and imposed seamlessly without having to resort to a library. Moreover, the virtual machine (VM) should take care of solving search problems transparently. This includes non-deterministic branching as well as constraint solving, i. e., finding specific values for logic variables given the imposed constraints.

Programming languages that facilitate all this in an integrated way can be subsumed under the term constraint-logic object-oriented programming (CLOOP). The absence of such a programming language constitutes a research gap that this dissertation sets out to fill. In short, the research objective of this dissertation is

Research objective *The design and development of a programming language that provides integrated support for object-oriented programming and constraint-logic programming.*

Three artefacts are developed in order to achieve this objective:

- First and foremost, a *programming language* for CLOOP that integrates constraint-logic features as a first-class citizen into an object-oriented programming language.
- Second, a VM as the *runtime environment* that executes applications that are developed in the programming language. Throughout the execution of an application, the VM provides support for non-deterministic search and employs a constraint solver.
- Third, a *compiler* that transforms source code that is written in the programming language into an intermediate representation that the VM can read and execute.

The work on these research artefacts yields additional contributions. Examples include a non-deterministic operational semantics for the execution of an imperative core language, a structure that encodes non-deterministic execution of (object-oriented and) imperative applications, and the implementation of different strategies for search over

the non-deterministic execution of stateful applications. Moreover, the concept of logic variables is extended beyond primitive types, so that logic variables can be used in lieu of objects.

1.3 Research Methodology

With the objective to design and develop a programming language, this dissertation contributes to computer science research. Ultimately, the results are also intended to benefit businesses, since a novel programming language that integrates facilities for object-oriented programming as well as for constraint-logic search improves the development of business software that frequently resorts to search. Consequently, this dissertation also provides a contribution to the field of information systems (IS) research that is concerned with solving problems that arise where people, organizations, and information technology intersect [SMB95]. To that end, IS research applies knowledge that is rooted in other disciplines (e. g., computer science) [Pef+07].

In IS research, there are two major streams that are distinguished regarding the kind of research objectives that they pursue [Hev+04]. One stream is primarily interested in behavioural research and aims at describing, understanding, explaining, and predicting the effects of technology in general and IS in particular [Gre02]. In contrast, the primary focus of the other stream is towards design-oriented research, viewing IS research as an engineering discipline with the aim of creating and evaluating innovative information-technology artefacts (e. g., software) [Öst+10; GH13].

The stated objective of this research is to design and to develop a programming language. This involves constructive tasks, so that this dissertation pertains to the design-oriented stream. Hevner et al. formulate guidelines that ensure the rigor and relevance of design-oriented research in IS [Hev+04]. The Design Science Research Methodology (DSRM) for IS research is based on these guidelines, proposing an iterative process model that comprises six steps [Pef+07]:

1. *Identify problem and motivate.* The first step of the DSRM requires researchers to state the problem that should be addressed. Moreover, researchers motivate the value of a future solution. For the present work, the motivation is the lack of an integrated programming language that would improve the development of (business) software that requires search.

2. *Define objectives of a solution.* Given that the process of designing solutions is an incremental one, the identified problem might not directly translate into an objective.

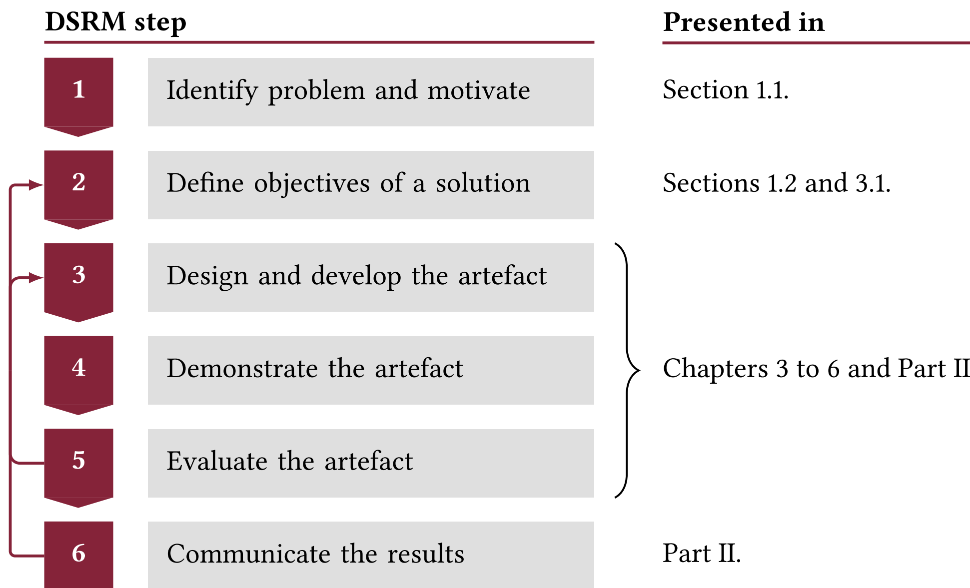


Figure 1.1: Relationships between DSRM steps and the structure of this dissertation.

Therefore, based on the problem specification, researchers infer objectives of the desired solution artefact. The objective of this research is presented in Section 1.2.

3. *Design and develop the artefact.* This step focusses on the creation of the artefact, i. e., of an “object in which a research contribution is embedded in the design” [Pef+07, p. 55]. For the present work, these artefacts are the programming language, the runtime environment, and the compiler.

4. *Demonstrate the artefact.* After the creation of an artefact, researchers apply it to the identified problem in order to demonstrate that it is useful. For instance, this can be achieved by using it in an experiment or in a case study.

5. *Evaluate the artefact.* As an alternative to demonstration, or as a complementary activity, a formal evaluation can prove that the artefact solves the identified problem. The DSRM is defined as an iterative process, enabling researchers to return to step 2 or 3 after the evaluation step.

6. *Communicate the results.* Last but not least, researchers need to publish the outcomes of all steps, including the problem, the motivation, and the artefact, in order to advance the state of research. Moreover, based on the received feedback or on own observations, researchers can make use of the iterative nature of the DSRM by returning to step 2 or 3 in order to continue research.

This dissertation applies the DSRM in order to achieve the stated research objective. Figure 1.1 illustrates how the structure and the contents of this thesis relate to the process steps of the DSRM.

1.4 Dissertation Outline

This dissertation is structured into two parts, with Part I providing an overview of the contributions of the research and Part II comprising the individual publications. In Part I, this exposition is followed by a presentation of the theoretical background of the research in Chapter 2. This lays the ground for the main chapters:

- The Münster Logic-Imperative Language (Muli) is introduced in Chapter 3, comprising a description of the language features, the compiler, and an accompanying runtime library. Moreover, examples demonstrate applications for Muli.
- Afterwards, Chapter 4 presents implementation details of a runtime environment for Muli applications, the Muli Logic Virtual Machine (MLVM). This is accompanied by a description of its main features, namely, symbolic execution, constraint solving, search trees, and trails.
- Based on that, Chapter 5 discusses extensions to CLOOP search. Specifically, it provides insights on how individual solutions can be calculated and retrieved, as opposed to exhaustively exploring the entire search space before returning all solutions to the invoking application at once. Moreover, it describes the implementation of different search strategies for CLOOP applications.
- Another extension to the previous concepts is the introduction of logic variables that represent objects. This extension and its implications regarding non-deterministic execution are presented in Chapter 6.
- Chapter 7 concludes Part I by summarizing the contributions of this research, also stating its limitations. Moreover, perspectives for future research are outlined.

Last but not least, Part II reproduces the eight academic publications that form part of this thesis.

2

Foundations

This chapter presents and explains the theoretical background of this thesis. Initially, logic programming is explained in Section 2.1. Afterwards, Section 2.2 provides an introduction to constraint-logic programming. Moving towards imperative programming languages, Section 2.3 presents Java bytecode in the context of the Java Virtual Machine (JVM) and explains how certain approaches facilitate symbolic execution of Java bytecode. Section 2.4 discusses existing support for constraint solving in contemporary imperative and object-oriented languages, with a focus on Java. Moreover, other concepts from declarative programming that have found their way into mainstream programming languages are outlined in Section 2.5.

2.1 Logic Programming and Non-Deterministic Execution

In logic programming, programs consist of declarative descriptions of a problem domain [CM03, Section 10.7]. A program specifies a set of propositions and relations between propositions, thus defining properties of elements from the program's problem domain as well as relationships between these elements. The program can then solve a logic problem by inferring implicit propositions about the problem domain from the ones in the explicit specification. In contrast to programming with typical imperative programming languages, logic programming does not require developers to explicitly define a sequence of operations that must be performed in order to solve a logic problem. Instead, a runtime environment that executes the logic program draws conclusions using a resolution algorithm (cf. [CM03, Section 10.4; Doe94]).

A prominent logic programming language is Prolog [CM03]. Prolog programs can be executed using an appropriate runtime environment, such as SWI-Prolog [Wie03]. The Warren Abstract Machine (WAM) describes an abstract machine for the execution of Prolog programs [War83]. In Prolog, a program consists of facts and rules regarding the problem domain [CM03, Section 1]. A question about the problem domain is specified as a goal, causing the runtime environment to infer whether the statement in the question is

correct. Alternatively, if the goal contains at least one variable, the runtime environment will attempt to find values for that variable so that the goal is satisfied.

In an excerpt from a recipe database in Prolog, Listing 2.1 presents facts for a predicate `ingredient(X, Y)`. A human reader would interpret the predicate as “`Y` is an ingredient of `X`”. Note that, in Prolog, names of variables begin with an uppercase letter, whereas names of constants start in lowercase.

```
1 ingredient(bolognese, spaghetti).
2 ingredient(bolognese, sauce).
3 ingredient(sauce, oil).
4 ingredient(sauce, tomatoes).
5 ingredient(sauce, oregano).
```

Listing 2.1: Specification of facts in a Prolog program.

A program can already formulate a goal with a query against these facts, such as

`?- ingredient(bolognese, oil).`,

to which the runtime environment responds with `false.` because the statement in the goal cannot be inferred from the given facts. Alternatively, facts can be used in rules. In Listing 2.2, the rule `contains(X, Y)` describes that `X` contains `Y`, either directly or indirectly, using two clauses. The Prolog syntax for a clause `A :- B.` is read as the implication `B` ⇒ `A`, i. e., `A` is true if `B` is true.

```
1 contains(X, Y) :- ingredient(X, Y).
2 contains(X, Y) :- contains(X, Z), ingredient(Z, Y).
```

Listing 2.2: Rule definition in a Prolog program.

Note that the rule definitions in Listing 2.2 overlap: While the left-hand sides match the same pattern, the clauses differ w. r. t. their right-hand sides. The implication is that `contains(X, Y)` is true if at least one of the clauses that define it can be inferred to be true. As a consequence, this rule considers direct ingredient relations as well as transitive ones via a recursive definition. With the rule definitions from Listing 2.2, the previous goal can be modified to

`?- contains(bolognese, oil).`

and would evaluate to `true.` because the relation `contains(bolognese, oil)` can be inferred from the specified facts and rules.

Moreover, goals can contain variables such as

```
?- contains(bolognese, Q).,
```

causing the runtime to find bindings for Q, i. e., specific values for which the goal is satisfied. In this example, there are five alternative bindings for Q that render the statement contains(bolognese, Q) true.

A Prolog runtime environment performs non-deterministic search in order to find all alternative bindings. In the non-deterministic execution of Prolog applications, choices are made regarding the bindings of variables as well as regarding the selection of clauses. The runtime environment creates a choice point when making a non-deterministic choice. Once an alternative has been evaluated in full (i. e., found to result either in a true or in a false statement), the runtime environment performs backtracking to the latest choice point, undoing the choice that was made in order to evaluate further alternatives. For instance, in order to infer bindings for Q in the goal

```
?- contains(bolognese, Q).,
```

a runtime environment could choose to consider the first clause of contains(X, Y) first, which results in the first choice point as well as in the evaluation of ingredient(bolognese, Q). With the next choice point, the runtime environment binds Q = spaghetti, which results in a true statement. Afterwards, backtracking occurs in order to evaluate another binding, Q = sauce. The runtime environment continues non-deterministic search in this way until all alternatives have been considered, or until execution is interrupted externally, e. g., from user interaction.

Concepts from logic programming, particularly w. r. t. non-deterministic execution, have found their way into other programming languages. For example, Curry is a functional-logic programming language [HKM95] whose syntax is largely based on that of Haskell [Jon03]. In Curry, a logic variable is declared using the free keyword. For example, in the Curry progam

```
main = one && two where one, two free,
```

both one and two are boolean variables that are not bound (and, therefore, “free”). Evaluating the above expression already results in non-deterministic execution, branching over the possible bindings of one (True and False). Furthermore, expressions can explicitly be formulated as non-deterministic using the choice operator ?. For example, the expression

```
main = 1 ? 2,
```

can evaluate to 1 as well as to 2. Moreover, non-determinism is implicitly introduced when there are overlapping function definitions. For example, both left-hand side definitions of nonDeterministicFun in Listing 2.3 are identical. Therefore, the runtime environment non-deterministically chooses one of the right-hand sides during evaluation. As in Prolog,

```
1 nonDeterministicFun :: a -> a -> a
2 nonDeterministicFun first second = first
3 nonDeterministicFun first second = second
```

Listing 2.3: In Curry, overlapping function definitions result in non-deterministic evaluation.

the runtime environment will eventually consider all alternatives unless only one solution is required.

Curry internally represents the non-deterministic evaluation of expressions using a search tree that is defined as (cf. [BHH04])

```
data SearchTree a = Value a
                  | Fail
                  | Or [SearchTree a].
```

In that representation, `Or` is the non-deterministic choice that maintains a list of alternative expressions. Leaves of the search tree are either of the form `Value a` that holds the value of an expression of a type that matches the type variable `a`, or `Fail` that represents an unsuccessful computation. For example, evaluating the Curry expression

```
nonDeterministicFun 1 2
```

with the function definitions from Listing 2.3 results in the search tree

```
Or [Value 1, Value 2].
```

These search tree representations are used, for example, in the context of encapsulated search [BHH04]. In a recent KiCS2 implementation of Curry (KiCS2 2.1.0), the function

```
someSearchTree :: a -> SearchTree a
```

can be used in order to obtain the above search tree [HPR16]. In combination with the KiCS2 function

```
allValuesDFS :: SearchTree a -> [a],
```

the expression

```
allValuesDFS $ someSearchTree $ nonDeterministicFun 1 2
```

encapsulates non-deterministic search. The results are collected in a list, returning `[1, 2]` for this example. `allValuesDFS` uses a depth-first search strategy to traverse the search tree of the encapsulated search. Analogously, further functions offer breadth-first search and an iterative-deepening strategy as alternatives for traversing the search tree.

There are several implementations of Curry that make Curry code executable on different platforms, including PAKCS that creates Prolog code from Curry programs [Han+19], KiCS2 that transpiles to Haskell code [Bra+11], or the Münster Curry Compiler

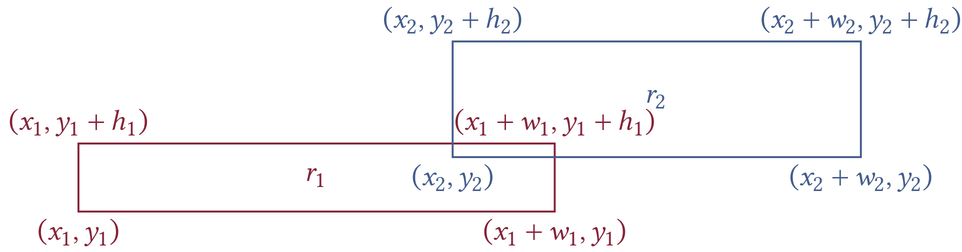


Figure 2.1: Rectangles r_1 and r_2 must intersect.

that compiles a Curry program for a custom Curry runtime system implemented in C [LK99; Lux99].

2.2 Constraint-Logic Programming

Like logic programming, constraint programming and constraint-logic programming both rely on declarative descriptions for the specification of problems. The introduction to constraint-logic programming first requires a short presentation of constraint programming concepts. A constraint program specifies a constraint problem as a set of variables, combined with (arithmetic and non-arithmetic) constraints that describe the relationships between the problem's variables [MS98, Section 1.2]. In contrast to an imperative implementation, a constraint program does not specify the steps for solving the problem. Instead, the program serves as the input for a constraint solver that collects the constraints and provides algorithms that are executed in order to solve the constraints [MS98, Section 1.3].

Constraint solvers differ regarding their supported theories, i. e., by the kinds of constraint problems that they can solve [FA03, Section 8.3]. For example, consider the following constraint problem. It specifies that two rectangles r_1 and r_2 must intersect (each represented by a tuple $r_i = (x_i, y_i, w_i, h_i)$, with coordinates growing from left to right and from bottom to top as illustrated in Figure 2.1):

$$(x_2 + w_2 \geq x_1) \wedge (x_1 + w_1 \geq x_2) \wedge (y_1 + h_1 \geq y_2) \wedge (y_2 + h_2 \geq y_1)$$

Depending on the assumed domain of the variables, solvers for different theories are adequate. For example, assuming the variables are real numbers, i. e.,

$$x_1, y_1, w_1, h_1, x_2, y_2, w_2, h_2 \in \mathbb{R},$$

the problem can be solved using algorithms for the linear arithmetic theory [FA03]. Alternatively, under the assumption that all variables are integers from a finite domain, e. g.,

$$x_1, y_1, w_1, h_1, x_2, y_2, w_2, h_2 \in \{1, 2, \dots, 5000\},$$

the problem can be solved by a constraint solver that implements algorithms for the finite domain (FD) theory. Non-numeric theories are also possible, for example in the context of solvers for string constraints (e. g., [Kie+09; Kri+20]). Satisfiability Modulo Theories (SMT) solvers implement algorithms with support for constraints from several theories [Bar+09].

After a constraint is imposed, the relationships that is formulated by the constraint must hold for all the values that are substituted for the involved variables. During execution of a constraint program, a constraint solver collects the constraints that are imposed incrementally, thus creating a constraint system that is the conjunction of all imposed constraints [FA03]. Among other tasks such as simplification [MS98, Section 8.3], the constraint solver uses the constraint system in order to perform the following tasks that are particularly relevant to this work and constraint-logic object-oriented programming:

Determining satisfiability The solver decides whether there is a possible substitution for every variable such that the conjunction of all constraints holds [MS98, Section 8.3]. For example, a constraint system of $x = y \land x \neq y$ is not satisfiable because of the contradiction, whereas checking $x = 1 \land x = y$ for satisfiability succeeds because there is a substitution for x and y that satisfies the constraint system. Satisfiability is used synonymously with consistency, i. e., a constraint system that is satisfiable is also consistent.

Solving A solution is a substitution for all (or all relevant) variables that satisfies all constraints of a constraint problem [Apt09, Section 2.1]. The way this is achieved depends on the underlying theory or theories. For example, in the context of FD solving (and for discrete domains in general), a solver might use non-deterministic search to try out possible values from the variables' domains for each variable, using backtracking to try other values if a substitution renders a constraint system inconsistent [FA03, Section 8.5]. In literature, such an enumerative approach is also referred to as labelling. Constraint systems consisting of linear arithmetic constraints can be solved using simplex-based algorithms [DD06]. Solvers can also combine labelling with other operations or algorithms [FA03, Section 8.5].

```
1 :- use_module(library(clpfd)).
2 must_intersect([X1, W1, Y1, H1, X2, W2, Y2, H2]) :-
3                X2 + W2 #>= X1, X1 + W1 #>= X2, Y2 + H2 #>= Y1, Y1 + H1 #>= Y2.
4 ?- Coords = [X1,W1,Y1,H1,X2,W2,Y2,H2], Coords ins 1..5000,
  ↪  must_intersect(Coords), label(Coords).
```

Listing 2.4: CLP(FD) program that searches for intersecting rectangles.

In the worst case, a labelling approach will enumerate all possible bindings for the variables of a constraint problem, so that the effort that is required for search depends on the combinatorial complexity. For instance, consider a constraint problem with two variables x and y, where each variable is an unsigned integer that is stored in 32 bits. Therefore, the variables initially have a finite domain with 2^{32} possible values each. Given a constraint system $c_1 \wedge c_2$ with the constraints c_1: $x > y$ and c_2: $x = 0$, labelling tries 2^{32} values for y in the worst case until finding out that there is no substitution that satisfies the constraint system. As a mitigation, constraint propagation algorithms can reduce the combinatorial complexity [Apt09, Section 5]. In this example, constraint propagation leverages the facts that, first, c_2 reduces the domain of x to the single value 0 and that, second, c_1 involves both x and y, in order to reduce the domain of y. Therefore, immediately as soon as both constraints are imposed, a constraint propagation algorithm will reduce the domain of y to contain only values $y < 0$, resulting in an empty domain. This reduces the required effort for solving as well as for determining satisfiability. During solving, labelling tries only values from the domain of a variable. As the domain is empty, the solver can easily determine that there is no substitution. This fact also helps when satisfiability is determined, as a constraint system is not satisfiable if the domain of at least one variable is empty. For details on specific constraint propagation algorithms, refer to the presentation by Apt [Apt09, Section 7].

Constraint-logic programming incorporates concepts from constraint programming in logic programming. In the example of Prolog, it is possible to use constraint-logic programming by importing a library, thus allowing to define and solve constraints as part of Prolog rules. For example, the CLP(FD) library adds support for FD constraints to Prolog [Tri12], whereas CLP(B) provides support for constraints over boolean variables [Tri18].

For demonstration, consider the implementation of the rectangle intersection problem using Prolog with CLP(FD). The first line of Listing 2.4 loads the CLP(FD) library, thus enabling the definition and use of FD constraints in the program. In CLP(FD) programs,

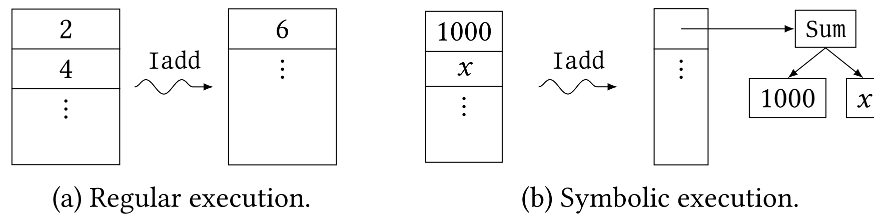
arithmetic constraints are prefixed by `#`, e. g., `#=` for equality. The `must_intersect` rule uses `#>=` to specify the greater-than constraints. Conjunctions are expressed with a comma (`,`) between individual terms, thus using regular Prolog syntax. Furthermore, the last line shows the goal that is used to search for concrete instances of rectangles. `Coords` is defined as the list of the variables that specify the rectangles' bounds. The domain of the variables is restricted to {1..5000} using `ins`. After the constraints of `must_intersect` are applied, `label` is used in order to find specific coordinates.

2.3 Symbolic Execution of Java Bytecode

Instead of operating on the human-written source code files of Java applications, a JVM parses and executes a binary intermediate representation of the application that is produced by a Java compiler [Lin+15, § 1.2]. For each class of the compiled application, this intermediate representation contains a single file that describes all aspects of the class. The intermediate representation defines the behaviour of the class's methods using so-called bytecode, i. e., instructions for the VM [Lin+15, § 6.5]. Executing a bytecode operation has an immediate effect on the execution state that the JVM maintains during the execution of an application. The JVM represents execution state using a combination of [Lin+15, §§2.5 f.]

- the *frame stack* that contains one element per executed frame, i. e., for every method that has been invoked and that has not terminated yet, representing the part of the execution state that is local to a method, such as values of local variables;
- one *operand stack* per frame, containing intermediate calculations on which most bytecode instructions operate;
- a *program counter (PC)* that references a specific instruction from the method that belongs to the topmost frame on the frame stack, thus defining which instruction is currently executed; and
- a *heap* that stores objects and arrays.

In standard Java, a variable can only be used after a value has been assigned to it, i. e., a variable always has a value [Gos+15, § 16]. As a result, the value that can be obtained from using a variable is always a constant, even though the value of non-final variables may be changed by a subsequent assignment (thus changing the value that will be obtained in later uses of the variable). In non-standard contexts, such as for the purpose of test-case generation, it may be desirable to initialize a variable with a logic variable instead of a constant [Kin76; MLK04]. For instance, consider an application that intends to derive



(a) Regular execution. (b) Symbolic execution.

Figure 2.2: Comparison of the effects of non-branching instructions on the operand stack.

test cases for a given method under test `boolean m(int a)`. Assume that the behaviour of `m()` depends on the value that is passed as a parameter. For the purpose of finding all distinct behaviours, it is desirable to invoke and execute `m()`, passing a logic variable instead of a constant value, in order to observe how the behaviour of the method depends on the value. Symbolic execution facilitates the use of logic variables in the execution of applications by changing the execution semantics for computations that involve logic variables [Kin76]. Therefore, the runtime environment has to provide support for the representation of logic variables and other symbolic expressions.

For a comparison, consider how the regular execution semantics of the `Iadd` bytecode instruction in Java differs from that in symbolic execution. In order to add two integer variables from the current operand stack, `Iadd` takes the two topmost elements from the stack, adds them, and pushes the result back to the stack. In regular execution semantics, the elements that are originally on the stack are all constants, so that the addition produces another constant that is pushed as a result. This is illustrated in Figure 2.2a. In contrast, if at least one element on the stack is a symbolic expression, such as a variable, the result of addition is a symbolic expression instead of a constant. The symbolic expression maintains the relationship between the result of the computation and the logic variable(s) that the computation depends on [Kin76]. For instance, executing `Iadd` on a constant and a logic variable yields a symbolic expression that represents their addition (Figure 2.2b).

The difference in execution semantics becomes even more interesting for instructions that branch the control flow of execution. For example, `If_icmpeq` jumps to a specific bytecode instruction if a condition is fulfilled, e. g., $1000 + x > 0$. If that condition is a symbolic expression, it is possible that the condition evaluates to `true` for specific substitutions of the involved logic variables whereas, for other substitutions, the condition evaluates to `false`. If this is the case, the control flow could continue either way, so that symbolic execution performs a non-deterministic choice [Kin76].

As a consequence, if all possible execution behaviours are of interest, symbolic execution has to take all possible decisions of non-deterministic choices into consideration. The execution paths of an application can be represented in a symbolic execution tree as

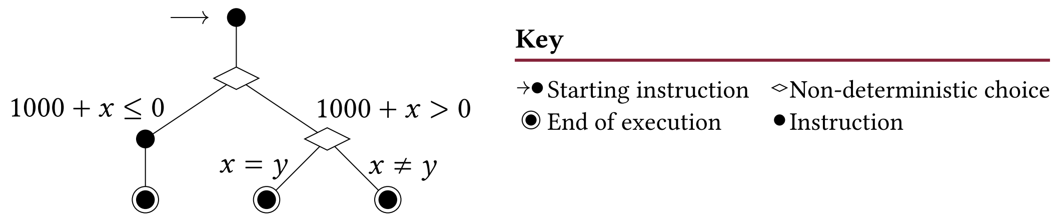


Figure 2.3: A symbolic execution tree, with branch constraints denoted at the edges.

illustrated in Figure 2.3, in which each inner node represents a non-deterministic choice with one subtree per decision alternative [Kin76].

Taking a decision at a non-deterministic choice results in a constraint that describes the prerequisite for taking that decision. This constraint is called a branch constraint. Branch constraints are indicated by the edge labels of the symbolic execution tree in Figure 2.3. For instance, if execution branches at an `If_icmpeq` instruction, the condition of the original `if` statement becomes the branch constraint of the `true` branch. As a consequence, the `else` branch for which the condition evaluates to `false` uses the negated condition as its constraint. The leaves of the symbolic execution tree are the outcomes of the program. For example, if the purpose of symbolic execution is to generate test cases for a method under test, there will be one leaf per value that is returned (or exception that is thrown) after taking a specific sequence of decisions at choices. Therefore, a single path from the root to a leaf corresponds to a specific execution of the method under test that can only happen if all branch constraints on the path hold. The conjunction of all branch constraints for a given path is called the path constraint [God+08]. Symbolic execution approaches can leverage constraint solvers for checking whether path constraints are satisfiable. Moreover, constraint solvers can be used to solve the constraint system in order to generate concrete values, e. g., as input data for generated executable test cases [Lem+04].

The symbolic execution of Java applications requires a custom JVM. Prominent examples of symbolic JVMs that have been developed for the purpose of test-case generation are IBIS [DM03], GlassTT [MLK04], and the Münster Generator of Glass-Box Test Cases (Muggl) [MK09]. Symbolic execution is also used in the context of languages other than Java [Meu01; TS06; TH08; CDE08; Cad+11]. Several implementations of symbolic execution only support single-threaded applications and disregard multi-threading [KPV03; MLK04]. In the presence of more than one application thread, a decision for the subsequently executed thread has to be taken after the execution of every single instruction. As a result, non-deterministic choices are made at branching instructions as well as after every single instruction [DK19a]. Therefore, support for multi-threading in combination

with non-deterministic search inflates the search tree, thus increasing the complexity of search. As a consequence, proper support for multi-threading in symbolic execution would require sophisticated techniques that reduce the search space.

2.4 Constraint Solving in Object-Oriented Languages

Object-oriented applications can embed constraint solving and search by implementing a custom solver or by importing a constraint solver library. For example, the automated test-case generator GlassTT implements the Münster Constraint Solving Toolkit (Muconst). Muconst is a custom constraint solver that is specialized for solving constraints in order to generate test data [Lem+04; EMK12]. However, most application scenarios do not require the additional effort that results from developing and maintaining a custom solver. A plethora of library implementations exist for several mainstream programming languages [Ceb20].

For the development of Java applications, contemporary constraint solver libraries include, e. g., Choco [PFL16], OptaPlanner [The17], and JaCoP [Kuc03]. These libraries are implemented in Java. Moreover, constraint solvers implemented in other programming languages offer bindings for Java, such as the Z3 solver [DB08].

In the context of this dissertation, Muconst and JaCoP are used as constraint solvers (see Section 4.3). The first solver, Muconst, is an SMT solver specifically geared towards the generation of test data after symbolically executing a Java method under test [Lem+04; EMK12]. It offers SAT solving for propositional logic as well as support for non-linear and linear arithmetic theories. Moreover, it is aware of potential rounding errors in floating-point solutions and ensures that rounding is correct w. r. t. the active constraint system. However, since Muconst relies on the Simplex algorithm for problems that involve numeric variables [EMK12], it can only decide the satisfiability of a constraint system by solving the entire problem using the Simplex algorithm. Therefore, checking the consistency of a branch's constraint system is computationally intensive. The second solver, JaCoP, is a library with a specific focus on FD constraints [Kuc03] that also includes experimental support for constraints involving floating-point variables. It leverages constraint propagation techniques in order to reduce the domains of finite-domain variables early, and uses labelling in order to find solutions to the imposed constraints [Kuc03].

A problem of using constraint solver libraries, as well as of developing a custom solver, is that there are no standardized interfaces. For instance, even though both the Choco

solver and JaCoP are libraries for Java, their facilities for defining constraints as well as for starting search differ [Kuc03; PFL16]. The Java Specification Request 331 describes an intention towards standardization [Fel12]. However, the latest activity on that request was recorded in 2012, and the efforts have not resulted in a uniform interface. As a consequence, switching the code of an application to a different solver library results in a tremendous effort. This results in problems once a used solver library is no longer maintained, or if the application requires novel kinds of constraints that the used library does not support.

2.5 Concepts from Declarative Programming in Imperative Languages

There are several approaches that integrate declarative programming and imperative programming, either as additions to existing languages or in the form of new, integrated programming languages. The selection that is presented is far from exhaustive, but it demonstrates that there is a continuous effort regarding the integration of useful concepts from multiple paradigms into a single programming language, in both research and practice. These additions and integrations increase the versatility of programming languages and reduce the need to develop manual integrations.

Alma-0 integrates non-deterministic choice into the imperative language Modula-2 [Apt+98; Wir85]. Non-deterministic choice is added to Alma-0 using special syntax. For example, the statement `EITHER` s_1 `ORELSE` s_2 `END` will cause the runtime environment to branch, first evaluating the statements in the s_1 branch and, after backtracking, those in s_2. It is possible to specify more than one `ORELSE` in order to create additional branches at the choice. Other statements add support for non-deterministic choice to loops and facilitate controlling the behaviour of search (e. g., statements for cutting alternative branches of execution). In an extension, the authors of Alma-0 also discuss the addition of constraints to the language, but leave the implementation of a solver to future work [AS99].

Logic Java attempts to integrate constraint-logic programming with object-oriented programming [MK11a]. The approach is based on Java and uses the symbolic JVM of Muggl for symbolic execution and constraint definition. In Logic Java, the `@Search` annotation is added to methods that the runtime environment should execute using non-deterministic search. That annotation is parameterized in order to allow specifying whether depth-first search or iterative deepening should be used as the search strategy.

Moreover, developers annotate variables that should be treated as logic variables with the `@LogicVariable` annotation. Even though the annotation-based approach allows writing valid Java code that compiles with a standard compiler, the use of annotations has a disadvantage. The Java compiler as well as the Java Virtual Machine Specification (JVMS) both limit support for variable annotations to field variables. Therefore, logic variables cannot be declared as local to a method and must be fields of a class instead. As a result, the structure of Logic Java code deteriorates in comparison to regular Java code.

PROLOG++ integrates object-orientation into Prolog [Mos94; KJ00]. This adds the ability to specify rules in the context of instantiable types, thus facilitating the specification of behaviour that depends on instance values. In contrast to Logic Java, Alma-0, and Muli, PROLOG++ tackles the integration from the opposite direction, adding the concept of objects to a declarative language. Visual Prolog has taken a similar approach that differs from PROLOG++ in its concrete syntax [Sco10]. Additionally, Visual Prolog includes facilities for creating graphical user interfaces. The programming language Oz is an additional example for an integration from the opposite direction. Oz integrates features from object-orientation into a constraint-logic programming language [Van+03]. In contrast to the other presented languages, Oz does not use a Prolog-based syntax.

The CAPJa approach sees benefits in the integration of object-oriented and logic programs [Ost15]. The integration consists of a so-called connector architecture that allows Java applications to call Prolog code as well as using Java objects from within Prolog code. The CAPJa approach is a useful addition to Java-based applications that heavily rely on search using Prolog applications. However, even though CAPJa syntax seamlessly integrates into the respective programming languages, application parts for search and application parts that comprise business logic are still implemented using different programming languages. Therefore, these parts are kept separate from each other, making this approach slightly less seamless than a single multi-paradigm programming language.

Further concepts from declarative programming have been introduced into popular imperative (and object-oriented) programming languages. With LINQ for C# [MBB06] and the Java Stream API for Java [UFM14; Ora20c], two mainstream programming languages prominently facilitate the use of concepts from functional programming in object-oriented programming languages, such as higher-order functions, lambda expressions, and non-strict evaluation. Recent C++ standards [Cuk17] as well as Python and JavaScript also introduce concepts from functional programming into imperative programming languages.

Moreover, programming language platforms that provide a single runtime environment for multiple programming languages contribute to the integration of programming languages from different paradigms. For example, the JVM serves as a platform for multiple languages besides Java [LWS13]. All programming languages for the JVM are compiled to the same bytecode, so that applications from different programming languages that run on the same JVM are interoperable. For example, Scala is a language for functional programming on the JVM that, since it compiles to the same bytecode, is able to invoke methods from classes developed in Java (and vice versa) [Hun18]. Similarly, all .NET programming languages compile to a common so-called Intermediate Language that is executed on the .NET Common Language Runtime [BS03]. With F#, .NET also offers a functional programming language that can be used in combination with C# or other imperative languages that are executed on the .NET Common Language Runtime [PS09].

3 The Münster Logic-Imperative Language as a Multi-Paradigm Language

This chapter introduces the Münster Logic-Imperative Language (Muli) that constitutes the principal research artefact of this thesis. Section 3.1 describes the principles guiding the design of Muli, including the relationship of Muli to its base language Java. This is followed by an outline of the main features of the Muli programming language in Section 3.2. Syntactic additions require the implementation of a compiler as presented in Section 3.3. Other language features do not require syntactic changes and are therefore implemented as a runtime library (Section 3.4). Section 3.5 demonstrates the Muli programming language using some exemplary applications. Finally, Section 3.6 summarizes the chapter and provides pointers to relevant publications from Part II.

3.1 Design Principles

Muli is designed as an extension to the object-oriented programming language Java, with Java 8 as the base version (cf. [Gos+15]). Consequently, Muli is a superset of Java, so that every Java program can be compiled and executed by Muli as well. Several benefits come from using Java as a starting point for the development of a new programming language. Firstly, the novel programming language might appeal to a large audience that is already familiar with Java, given the ubiquity of Java in contemporary software development (cf. e. g. [Sta19]). Secondly, a multitude of libraries is available for Java which will also be usable from the new programming language, thus yielding a highly useful and flexible language right away. Thirdly, there is extensive documentation on the syntax of the programming language [Gos+15] as well as on the Java bytecode format and its execution semantics [Lin+15] which, in combination, facilitate the implementation of a derivative language.

Muli is developed with a set of design principles in mind [DK18a]:

Transparent problem solving The programming language can be used to develop object-oriented programs, with a means to formulate search problems that a specialized VM solves transparently.

Encapsulated search Search, and therefore non-deterministic execution, is encapsulated and returns solutions as well as a representation of the search space. Non-deterministic execution is only performed if it is required explicitly by the program. In contrast, remaining parts of the program are deterministic in execution. Moreover, encapsulation allows application developers to influence the behaviour of search.

Minimal syntax extension The syntax additions compared to Java are kept to a minimum. This includes the definition of constraints that can be expressed using the relational operators that exist in Java [Gos+15], as opposed to adding special constraint definition operators.

No lazy evaluation As opposed to integrations of (constraint) logic programming with other paradigms (cf. e. g. Curry [Han97]), Muli applications are not executed lazily. For instance, the assignment of an expression to a variable causes the expression to be evaluated, regardless of whether the result of the assignment is actually used by subsequent instructions. This is done so that the execution semantics of bytecode instructions in Muli remains consistent with that in Java.

Maintain Java backwards compatibility Compatibility with Java is maintained as much as possible, so that most pre-existing Java programs and libraries can be used in combination with Muli programs. This implies that Muli extends Java syntax, but that existing syntax is not removed. Another implication is that, in deterministic execution, bytecode instructions are evaluated according to the official JVMS (cf. [Lin+15]), i. e., in deterministic execution contexts there is no semantic difference between bytecode of Muli programs and bytecode of Java programs.

As a notable exception to the effort of maintaining backwards compatibility, Muli currently assumes single-threaded applications and therefore does not support multi-threading for the reasons outlined in Section 2.3. In particular, the `Monitorenter` and `Monitorexit` bytecode instructions for thread synchronization are ignored, and there is no implementation for the native methods in `java.lang.Thread` that would otherwise start new threads [Lin+15]. Therefore, the development of multi-threading facilities for Muli is left to future work.

3.2 Language Features of Muli

A core feature of Muli is the possibility to declare logic variables. In contrast to a regular variable, a logic variable is not initially bound to a specific value. For that reason, logic variables in Muli are also called *free variables*; the terms are used interchangeably. A free variable is declared by adding the `free` keyword to a variable declaration, e. g., `int i free;`. Any variable of a primitive type can be a logic variable, therefore all declarations shown in Listing 3.1 are valid declarations. Both class fields and local variables can be free variables.

```
1 boolean z free;
2 byte b free;
3 short s free;
4 int i free;
5 long j free;
6 char c free;
7 float f free;
8 double d free;
```

Listing 3.1: Valid declarations for free variables that have a primitive type.

In a Muli program, free variables and regular variables can be used interchangeably. Consequently, a free variable can be employed as part of any expression in the same way as a regular variable, as long as the types are otherwise compatible. For instance, consider the declarations in Listing 3.2. An invocation of `plusOne(i1)` is trivially valid because both the parameter and `i1` are of type `int`. Since free variables can be used in lieu of regular ones, `plusOne(i2)` is valid in Muli as well. In contrast, `plusOne(d1)` cannot compile, even with a standard Java compiler, because an implicit cast of a `double` variable to `int` results in potential loss of information. For the same reason, `plusOne(d2)` is also not allowed in Muli.

```
1 int plusOne(int arg) { return arg + 1; }
2 int i1 = 1000;
3 int i2 free;
4 double d1 = 1000.0;
5 double d2 free;
```

Listing 3.2: A demonstration of how free and regular variables can be used interchangeably.

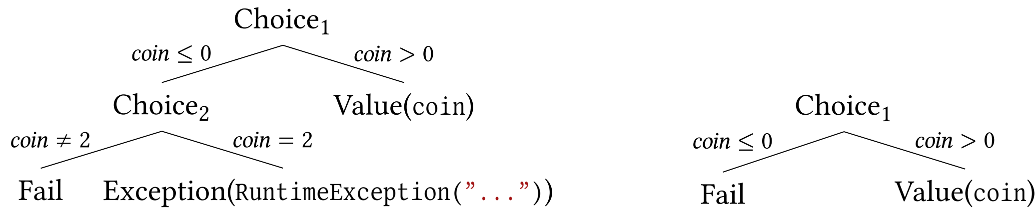
```
1  int exampleSearchRegion() {
2    int coin free;
3    if (coin > 0) { // Branch A.
4      return coin;
5    } else { // Branch B.
6      if (coin == 2) { // Branch B-1.
7        throw new RuntimeException("Exceptions are solutions, too.");
8      } else { // Branch B-2.
9        throw Muli.fail(); } } }
```

Listing 3.3: Abstract search region, demonstrating the specification of constraints and solutions.

Moreover, variables of a class or interface type, i. e. objects, can be free variables as well. As their interpretation in an object-oriented program is more complex than that of primitive types, details are provided in Chapter 6.

A Muli application specifies a search problem in a method. Such methods are called *search regions*. Search region methods can contain arbitrary code, i. e., they can invoke other methods and use recursion, create and interact with objects, or perform other non-search-related tasks. *Solutions* to the search problem can be values or exceptions and are determined based on how the execution of the search region ends. Specifically, a solution is either the return value of the search region, or an uncaught exception that is thrown inside the search region. Consider the abstract example search region in Listing 3.3. Branch A returns the (constrained, see below) free variable `coin` as a value. Branch B-1 throws a runtime exception that is also considered as a solution. Alternatively, `throw Muli.fail()` throws a special exception in Branch B-2. This is not a solution; instead, invoking `Muli.fail()` expresses an explicit *failure* in order to stop execution of the current branch and to exempt it from the final solutions.

Muli offers *non-deterministic search* by making a non-deterministic choice whenever the execution flow is branched in two or more ways that are equally possible. For instance, consider the equivalent of flipping a coin, shown in Listing 3.3: As the `coin` variable is not bound, it is equally feasible to evaluate the first `if` condition either to `true` or `false`. Therefore, initially, both Branch A and Branch B are candidates for further execution and the runtime environment (e. g., the MLVM) has to take a decision [DK18b]. The resulting execution paths can be conceptualized as a search tree that is a symbolic execution tree whose leaves represent either the solutions to the search problem or failures. Figure 3.1a



(a) Conceptual search tree, disregarding the satisfiability of the path constraints.

(b) Actual search tree that is known after the search region has been executed in full.

Figure 3.1: Search trees that represent non-deterministic search for the search region specified in Listing 3.3.

illustrates the search tree for the example search region from Listing 3.3, depicting the non-deterministic choices as well as the solutions.

Taking a decision at a choice causes the runtime environment to *impose a constraint* that represents and enforces the decision. That way, applications can impose constraints dynamically at runtime in order to specify a search problem. Constraints are derived from the condition [DK18b]. In the first `if` condition of Listing 3.3, two alternative constraints are derived, namely, $\gamma_1 = \texttt{coin} > 0$ and $\gamma_2 = \neg\gamma_1 = \texttt{coin} \leq 0$. As a result of deriving constraints from branch conditions, Muli does not add or require any special syntax for constraint specification. Instead, the runtime environment imposes the derived constraint in order to commit to a branch [DK18b]. For example, after imposing γ_1 the condition uniquely evaluates to `true`, so that execution can continue with Branch A, returning the constrained `coin` variable as a solution. Alternatively, Branch B is executed after imposing γ_2. Unless only a single solution is required, the runtime environment is expected to evaluate all available alternatives non-deterministically so that all alternatives are considered eventually [DK19a]. For a core imperative language, a non-deterministic operational semantics that describes non-deterministic branching is presented in [DK18b]. Moreover, Chapter 4 describes the implementation of non-deterministic search with backtracking in the MLVM.

Taking several non-deterministic choices in sequence imposes a *constraint system* that comprises the conjunction of all imposed constraints, i. e., the path constraint [DK18b]. The constraint system ensures that future evaluations of the involved variables are consistent with the assumptions that were made at the time of branching. In the example from Listing 3.3, this implies that `coin` in Branch A cannot assume any value less than or equal to 0, because that would violate the assumption that was made to enter that branch. For Branch B, the constraint is negated, so that `coin` may only assume values less than or equal to 0. The condition of the second `if` in Listing 3.3 violates this constraint, so that

```
class HardyRamanujan {
  int search() {
    int a free, b free, c free, d free, e free;
    positiveDomain(a, b, c, d, e);
    if (a != c && a != d && cube(a) + cube(b) == e && cube(c) + cube(d) == e)
      return e;
    else throw Muli.fail(); }

  int cube(int x) { return x * x * x; }

  void positiveDomain(int... vars) {
    for (int v : vars)
      if (v <= 0) throw Muli.fail(); } }
```

Extended from [DK20a]

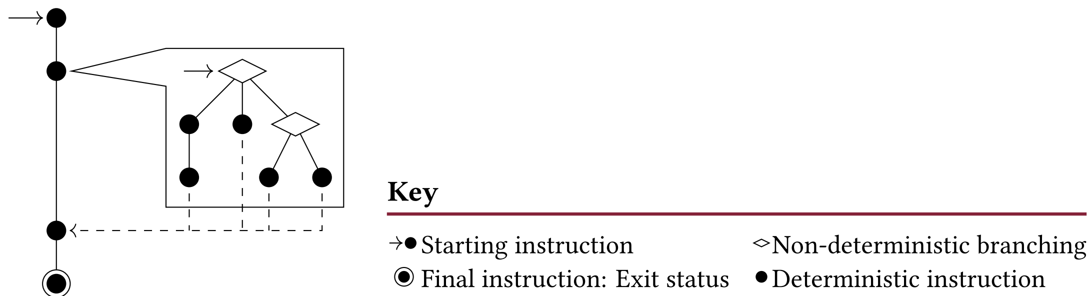
Listing 3.4: Muli code that searches for an integer *e* that can be expressed in two different ways as the sum of two positive integer cubes.

Branch B-1 cannot be executed. Therefore, the exception in Branch B-1 is actually not a solution because the constraint system $coin \leq 0 \wedge coin = 2$ that would be required to reach that solution is not satisfiable due to the contradictory constraints. As a consequence, evaluating the second `if` does not result in non-deterministic branching at all. This is reflected in the updated search tree in Figure 3.1b, replacing Choice_2 with the failure from Branch B-2.

As a less abstract search problem, consider an example that searches for an integer *e* that can be expressed in two different ways as the sum of two positive integer cubes (ideally the smallest integer; i. e., the Hardy-Ramanujan number $e = 1729$). The corresponding constraint is [DK20a]:

$$
\begin{aligned}
& e = a^3 + b^3 = c^3 + d^3 \\
\wedge\ & a \neq c \wedge a \neq d \\
\wedge\ & a, b, c, d, e \in \mathbb{N} - \{0\}
\end{aligned}
$$

These constraints can be expressed by the Muli search region `search()` shown in Listing 3.4. This example also demonstrates the use of explicit failure in Muli: A solution in which the constraints are not fulfilled is not interesting w. r. t. the problem, so the explicit failure is used to exclude the branch from the found solutions.



Adapted from [DK18a]

Figure 3.2: Deterministic main execution control flow of a Muli application. Parts that are executed non-deterministically are restricted to encapsulated search. Results are collected and returned to the surrounding application.

In Muli, search is *encapsulated*. Non-deterministic branching is limited to encapsulated search and is disallowed outside of search. After all, contemporary operating systems require a single exit status of an application, therefore it is not possible to have one exit status per leaf of the symbolic execution tree. As illustrated in Figure 3.2, the main execution control flow of a Muli application is deterministic and ends in a single exit status, but the control flow may break out into non-deterministically executed program parts using encapsulated search. The surrounding main execution control flow ensures that the results of encapsulated search are collected (indicated in Figure 3.2 using dashed lines), thus joining the branches of non-deterministic execution before continuing with deterministic execution. Encapsulated search returns solutions, including representations of the search space that correspond to each solution.

Muli provides *encapsulated search operators* that accept a search region as a parameter, causing the runtime environment to start non-deterministic search for the search region, while collecting the found solutions in order to return them to the surrounding application. Search regions can be formulated either in a lambda expression (cf. [Lin+15, § 15.27]) or, facilitating reuse, in a method that is passed to the operator using a method reference (cf. [Lin+15, § 15.13]). A static class offers multiple encapsulated search operators that differ in the number of returned solutions and in the way that they are returned to the surrounding application. Details on the static class and its encapsulated search operators are presented in Section 3.4. Listing 3.5 shows exemplarily how the `Muli.getAllSolutions()` operator is used with a method reference that points to the search region from Listing 3.4.

```
1 Solution<Integer>[] integers = Muli.getAllSolutions(HardyRamanujan::search);
```

Listing 3.5: A search region, i. e. a method that formulates a search problem, can be passed to an encapsulated search operator using a method reference.

In summary, Muli programs can declare logic variables, allowing their use in symbolic expressions in any part of the program. Non-deterministic search is encapsulated, whereas execution is deterministic outside encapsulated search. The tight integration of imperative (object-oriented) programming with non-deterministic search facilitates the development of applications that interleave constraint definition and search with imperative statements, instead of having to maintain imperative program parts separate from search. Compared to Java, the only modification of the syntax is the addition of the `free` keyword. The remaining features either re-use existing syntax (e. g., for constraint definition) or can be implemented as methods of a static class (e. g., search operators). The static class is made available as a runtime library (cf. Section 3.4). Nevertheless, adding a keyword and encoding additional information about free variables into the bytecode motivate the development of a custom compiler for Muli as presented in Section 3.3.

3.3 The Muli Compiler

Muli requires a custom compiler in order to add support for the `free` keyword for declarations of fields and variables. The Java Language Specification (JLS) defines the productions for field declarations as follows (definitions adapted from [Gos+15, § 8.3]):

FieldDeclaration ::= FieldModifier Type VariableDeclarator (, VariableDeclarator)* ;*
FieldModifier ::= Annotation | `public` | `protected` | `private` | `static` | `final` | `transient` | `volatile`
Type ::= ReferenceType | `boolean` | `byte` | `short` | `int` | `long` | `char` | `float` | `double`
VariableDeclarator ::= VariableDeclaratorId (= VariableInitializer)?

In these productions, *Annotation* stands for an `@`-annotation, such as `@NotNull`. Furthermore, *ReferenceType* is a type variable or a class, interface, or array type. Both *Annotation* and *ReferenceType* are dynamic non-terminal symbols that depend on the types that are available from the class path at runtime, so neither rule can be reproduced here. Further-

more, *VariableDeclaratorId* is a valid identifier name by which the field can be referenced after its declaration,[1] and *VariableInitializer* is an expression or an array initialization.

Moreover, for local variables, Java uses the following productions (definitions adapted from [Gos+15, § 14.4]):

LocalVariableDeclaration ::= VariableModifier Type VariableDeclarator (, VariableDeclarator)* ;*

VariableModifier ::= Annotation | `final`

FieldDeclaration and *LocalVariableDeclaration* reference the same *VariableDeclarator*. Therefore, free fields and variables are both added to the syntax by modifying a single non-terminal symbol. The *VariableDeclarator* non-terminal is changed so that the `free` keyword becomes an alternative to initialization [DK19a]:

VariableDeclarator ::= VariableDeclaratorId (`free` | *(* `=` *VariableInitializer))?*

In comparison to the existing Java syntax (cf. [Gos+15, § 19]), the extension is very small. The ExtendJ framework (formerly known as JastAddJ) is a compiler for Java 8 that offers full support of the syntax and bytecode, while providing the opportunity to add incremental modifications [EH07]. ExtendJ enforces a layer structure of language modifications: Its compiler for Java 8 is a layer of incremental modifications of the ExtendJ compiler for Java 7 that, in turn, is a modification layer on top of the ExtendJ compiler for Java 6 (and so on, with Java 1.4 as the base language) [EH07]. As opposed to implementing a compiler for Muli from scratch with full support for Java 8, leveraging ExtendJ facilitates a simple compiler implementation by adding a layer of Muli modifications.

In the parser generator of ExtendJ, existing parser rules can be modified incrementally. For example, in Listing 3.6, line 1 adds the `free_variable_declarator` as an alternative to the `variable_declarator` production rule that the ExtendJ framework defines. The parser generator implicitly adds repeatedly defined rules as alternatives to previous definitions, so that it is not necessary to reproduce the existing definition of `variable_declarator`. The ExtendJ syntax is peculiar: `FREE` is a reference to the terminal symbol for the `free` keyword, whereas the syntax `rule.result` (as for instance in `free_variable_declarator_id.v`) signifies that the production rule `rule` should be applied, storing the result in a variable `result`. Additional Java code that will be applied is specified between `{:` and `:}`. Usually this additional code involves creating, modifying, or returning abstract syntax tree (AST) nodes. An optional rule application is marked with a question mark `?`, such as in the optional specification of array dimensions (`dims.s?`). For instance, in

[1] This is simplified for the purpose of explaining the modification. According to the JLS, *VariableDeclaratorId* also optionally accepts further annotations and array dimensions for individual fields [Gos+15, § 8.3], but that feature is not relevant in this context.

Listing 3.6, `free_variable_declarator` returns the `FreeVariableDeclarator` AST object that is instantiated as a result of applying the `free_variable_declarator_id` rule. In combination, the rules in Listing 3.6 modify the allowed syntax, implementing free variables as an alternative to regular variables. The `field_declarator` rule that is also pre-defined in the framework is augmented analogously.

```
1  VariableDeclarator variable_declarator = free_variable_declarator;
2  FreeVariableDeclarator free_variable_declarator =
3    free_variable_declarator_id.v FREE {: return v; :};
4  FreeVariableDeclarator free_variable_declarator_id =
5    IDENTIFIER dims.d? {: return new FreeVariableDeclarator(IDENTIFIER, d, new
     ↪  Opt()); :};
6
7  FieldDeclarator field_declarator = free_field_declarator;
8  FreeFieldDeclarator free_field_declarator =
9    free_field_declarator_id.v FREE {:    return v; :};
10 FreeFieldDeclarator free_field_declarator_id =
11   IDENTIFIER dims.d? {: return new FreeFieldDeclarator(IDENTIFIER, d, new
     ↪  Opt()); :};
```

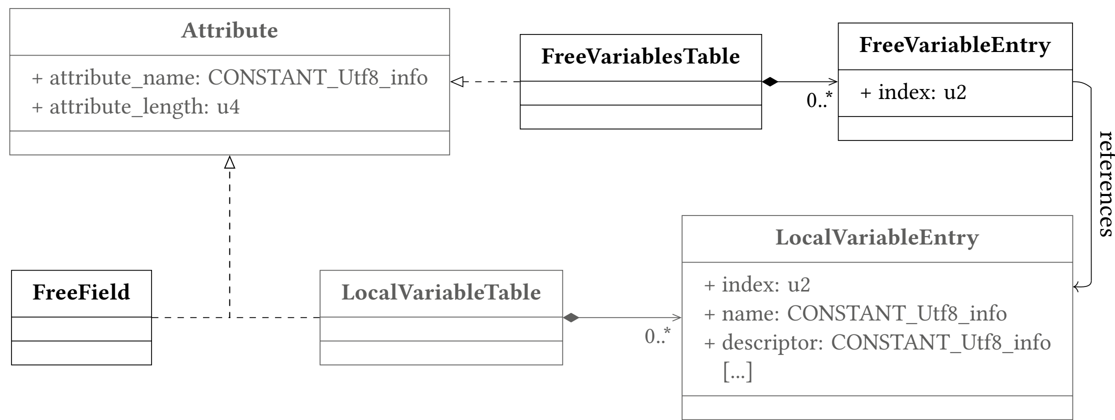
Listing 3.6: Implementing the modification to the production rules using ExtendJ's parser generator.

As a result, the productions generate `FreeVariableDeclarator` and `FreeFieldDeclarator` nodes for the AST. Free variable declarators are a specialization of (regular) variable declarators and free field declarators specialize (regular) field declarators. This relationship is defined by the specification shown in Listing 3.7 that ExtendJ adds to its pre-defined AST declarations.

```
1  FreeVariableDeclarator : VariableDeclarator;
2  FreeFieldDeclarator : FieldDeclarator;
```

Listing 3.7: Declaring Muli's new AST node types for the ExtendJ framework.

Based on the generated AST, the Muli compiler generates bytecode structures that encode information about free variables and fields in the compiled `.class` files. The bytecode format of the `.class` files conforms to the JVMS (see [Lin+15]), so that existing Java bytecode parsers are able to read files compiled by the Muli compiler. Additional information needed for free variables is encoded in so-called attribute structures that



Adapted from [DK19a]

Figure 3.3: Attribute structures that are generated into the compiled bytecode. Types used in this figure follow the JVMS [Lin+15], where u*n* signifies an *n*-byte unsigned integer and `CONSTANT_Utf8_info` is a string constant.

allow the addition of custom attribute types [Lin+15]. Muli leverages this flexibility for the representation of free variables and fields in bytecode, adding the three custom attributes that are depicted in the class diagram in Figure 3.3: `FreeField`, `FreeVariablesTable`, and `FreeVariableEntry`. In contrast, attributes depicted in gray are standard Java attributes. While parsing a bytecode file, parsers that do not know about the custom Muli attributes are able to skip over them, thus ignoring them, because attributes are required to specify their own length in their header (`attribute_length` in the specification of Attribute, see Figure 3.3).

Per method, in the `method_info` attribute structure, compiled bytecode contains a `LocalVariableTable` that holds one `LocalVariableEntry` for each declared local variable [Lin+15, § 4.7.13]. It is not possible to extend the `LocalVariableEntry` definition with a flag that signifies a free variable without breaking bytecode compatibility. Therefore, the custom `FreeVariablesTable` attribute mimicks the structure, containing a `FreeVariableEntry` per declared free variable. The `FreeVariableEntry` only contains the index of a local variable, which is used to reference the corresponding `LocalVariableEntry` that holds information about the type (encoded in the descriptor) and the name of the free variable.

Representing free fields is simpler, as compiled bytecode contains individual `field_info` structures for every field. In this structure, the custom `FreeField` attribute serves as a marker: A field is a free field if its `field_info` structure contains a `FreeField` attribute. Therefore, the attribute does not need to store any additional data.

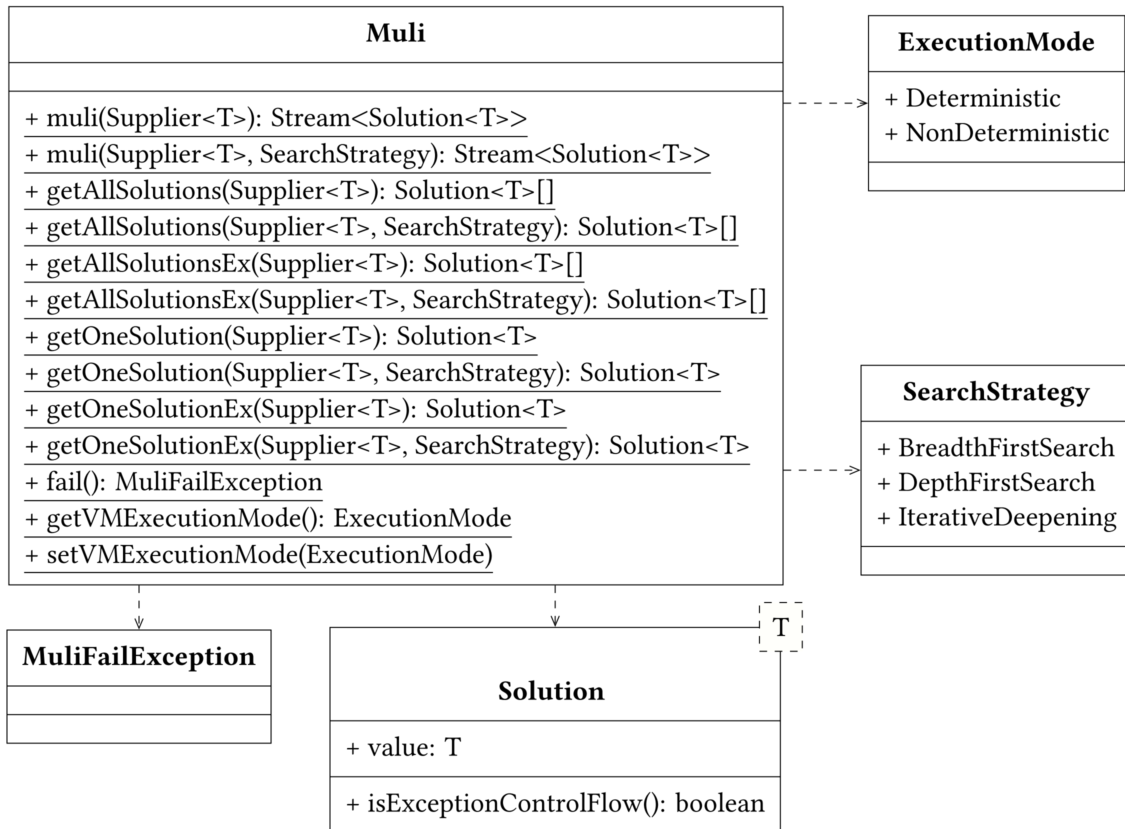


Figure 3.4: Public API offered by the Muli runtime library.

3.4 The Muli Runtime Library

Language features that do not require changes to the syntax, such as explicit failures and encapsulated search operators, are made available to Muli programs in a runtime library. This library serves as an application programming interface (API) and is always loaded on the class path of Muli applications. As a consequence, Muli applications can use classes from the library without having to load it explicitly. This approach is similar to that of the Java Platform SE API, which defines the standard set of classes that are always available on the class path of Java applications without explicitly loading them [Ora20a].

Figure 3.4 gives an overview of the classes in the public API. The central class of the Muli runtime library is `de.wwu.muli.Muli` (subsequently abbreviated to `Muli`). Muli applications do not need to instantiate this class; instead, all features are offered as static methods. The `Muli` class offers a method that applications can use to express the explicit failure, methods that alter the execution behaviour of the MLVM, and encapsulated search operators. These features are explained in the following.

The method `Muli.fail()` is offered to Muli applications for expressing an explicit failure in non-deterministic execution, thus exempting it from the found solutions. Even though its full definition is

`public static native MuliFailException fail();`,

the `MuliFailException` cannot be caught by a surrounding program, because invoking this method causes the MLVM to immediately perform backtracking to the latest choice that offers another alternative. Therefore, the `MuliFailException` class is never instantiated and serves merely as a syntactic detail that allows developers to write

`throw Muli.fail();`

in their Muli applications, which helps static analyses of Java integrated development environments (IDEs) to determine that a `Muli.fail()` validly ends the execution of a method. Otherwise, contemporary IDEs would complain about the absence of a `throw` or `return` statement in branches that specify an explicit failure. Furthermore, the `native` keyword in the declaration specifies that the method is not actually implemented in the library. Instead, an invocation of this method is picked up by the MLVM that provides an appropriate implementation.

Another method that is declared `native` is `Muli.setVMExecutionMode()`. It accepts one of two values: `ExecutionMode.Deterministic` and `ExecutionMode.NonDeterministic`, where the enumeration type `ExecutionMode` is provided as part of the Muli runtime library as well. Invoking this method instructs the MLVM to switch execution modes, thus changing the interpretation of bytecode instructions that have potentially non-deterministic behaviour (e. g., `If_icmpeq` that compares two integers for equality). The method is only meant to be used by the encapsulated search operators, not by Muli applications. Instead, encapsulated search operators use `Muli.setVMExecutionMode()` to begin and end encapsulated search. In combination with `Muli.getVMExecutionMode()` that obtains the current mode of execution from the MLVM, an encapsulated search operator changes MLVM roughly as demonstrated in Listing 3.8. Obtaining the previous execution mode is important because it facilitates nesting of search regions.

```
1 ExecutionMode previousExecutionMode = Muli.getVMExecutionMode();
2 Muli.setVMExecutionMode(ExecutionMode.NonDeterministic);
3 // <Perform search using non-deterministic execution.>
4 Muli.setVMExecutionMode(previousExecutionMode);
5 // <Wrap and return solution.>
```

Listing 3.8: Changing the MLVM execution mode before and after search.

Last but not least, the Muli runtime library provides encapsulated search operators that accept a search region in the form of an object of a type that implements the functional interface `java.util.function.Supplier<T>`. Typically, instances of the `Supplier` type are transparently created by the JVM (and, therefore, by the MLVM) at runtime, serving as a substitute for a method reference or a lambda expression from the original source code. Regardless of whether the search region is formulated as a method reference or as a lambda expression, a prerequisite for both is that they do not have any parameters and that they have a return value, as per the definition of `Supplier<T>` [Ora20b]. Search operators return a solution, an array of solutions, or a stream of solutions. Every found solution is wrapped in a `Solution<T>` object, where `T` is identical with `T` of the `Supplier<T>` parameter. Furthermore, all search operators are overloaded, offering an additional parameter that specifies the search strategy used by the MLVM for processing the search region. Developers use the enumeration type `SearchStrategy` for choosing whether a given search region should be explored using depth-first search, breadth-first search, or an iterative-deepening strategy. If that parameter is not provided, depth-first search is currently used as the default strategy. Details on the search strategies and their implementations are presented in Section 5.2.

`Muli.muli()` is the main encapsulated search operator. The invocation of `Muli.muli()` returns a stream of solutions. Since `Muli.muli()` returns a stream object and Java streams are non-strict, invoking `Muli.muli()` does not cause the MLVM to search for solutions immediately. Instead, the stream object can be used as the source in a so-called stream pipeline that processes objects from a source by composing stream operations from the Java Stream API [Ora20c]. In a stream pipeline, a source is followed by intermediate operations that modify a stream (e. g., `map()`, `filter()`, or `limit()`). A terminal operation concludes a pipeline, e. g., `forEach()` or `count()`. The solution stream of Muli causes the MLVM to find individual solutions as soon as they are demanded by a terminal operation invoked on the solution stream or pipeline. Section 5.1 provides details on the implementation of the solution stream.

The remaining encapsulated search operators are convenience methods that process the solution stream returned by `Muli.muli()`. Like `Muli.muli()`, they are overloaded and can be invoked with only a search region, or with a search region and a strategy. `Muli.getAllSolutions()` processes the entire search region and returns an array with all solutions that were collected from the evaluation of `return` statements, therefore omitting any exceptions thrown during the evaluation of the search region. Listing 3.9 shows how this convenience method that anticipates the need for processing all solutions is implemented on the basis of `Muli.muli()`. `Muli.getAllSolutionsEx()` is similar, but the returned

```
public static <T> Solution<T>[] getAllSolutions(Supplier<T> searchRegion,
  ↪ SearchStrategy strategy) {
  Stream<Solution<T>> search = Muli.muli(searchRegion, strategy);
  return (Solution<T>[]) search
    .filter(x -> !x.isExceptionControlFlow())
    .toArray(size -> new Solution[size]); }
public static <T> Solution<T> getOneSolution(Supplier<T> searchRegion,
  ↪ SearchStrategy strategy) {
  Stream<Solution<T>> search = Muli.muli(searchRegion, strategy);
  return (Solution<T>) search
    .filter(x -> !x.isExceptionControlFlow())
    .findFirst().get(); }
```

Listing 3.9: Several encapsulated search operators are convenience methods that anticipate frequent ways of accessing the results of search, processing the stream returned by `Muli.muli()` in predefined ways.

array includes thrown exceptions as well. Therefore, the only difference in the implementation is that the `filter()` operation is omitted. Note that `Muli.getAllSolutions()` and `Muli.getAllSolutionsEx()` do not return a result if infinite non-deterministic branching occurs in the search region, because the `collect()` terminal operation unsuccessfully waits for the solution stream to finish. Furthermore, `Muli.getOneSolution()` and `Muli.getOneSolutionEx()` process the solution stream as well, where `Muli.getOneSolutionEx()` is indifferent to whether the first solution of a search region is a thrown exception or a returned value. Exemplarily, the implementation of `Muli.getOneSolution()` is also depicted in Listing 3.9. The `findFirst()` terminal operation only requests a single element from the solution stream so that, even for search regions with infinite non-deterministic branching, `Muli.getOneSolution()` and `Muli.getOneSolutionEx()` are able to return a solution.

3.5 Applications of Muli

Muli's language features facilitate the implementation of various search applications using an object-oriented syntax. Muli is useful for solving constraints in several applications. Moreover, non-deterministic execution can be exploited for the purpose of enumerating alternatives. The following presents examples for both cases.

A classic search problem is the n-Queens problem (for details refer to, e. g., [Apt09, Section 2.2]). The goal is to find locations for n queens on an $n \times n$ board, $n \geq 3$, such that no queen is able to attack any other queen according to the queen's movement rules of

```
1  public class NQueens {
2    public static void main(String[] args) {
3      Solution<Queen[]> solution = Muli.getOneSolution(() -> {
4        final int n = 8;
5        Board board = new Board(n); Queen[] qs = new Queen[n];
6        for (int i = 0; i < n; i++) {
7          Queen q free; qs[i] = q; }
8        for (int i = 0; i < n; i++) {
9          if (!board.isOnBoard(qs[i]))
10           Muli.fail();
11         for (int j = i+1; j < n; j++)
12           if (board.threatens(qs[i], qs[j]))
13             Muli.fail(); }
14       return qs; });
15
16     for (Queen q: solution.value)
17       System.out.println("(" + q.x + "," + q.y + ")"); } }
```

[DK20b]

Listing 3.10: Muli application that returns a solution for the n-Queens problem (here, $n = 8$).

chess. Listing 3.10 presents a solution approach that uses CLOOP and Muli, in which the search region is formulated using a lambda expression. The search region first initializes representations of the board and of the n queens. Methods that impose constraints are implemented in the `Board` class, namely `isOnBoard()` that restricts the positions of queens on the board to $0 < x \leq n$ and $0 < y \leq n$ (where x and y are coordinates on the board), and `threatens()` that, for two queens that are passed as arguments, imposes that the two queens may not be on the same column, row, or diagonal. `Muli.fail()` excludes wrong solutions, and solutions that satisfy all constraints are returned using `return qs;`.

As another application example, Muli has been successfully used for the generation of hidden layers of feed-forward neural networks [DK20a]. The `NNGenerator` application does not perform constraint solving; instead, it leverages non-deterministic branching to systematically generate directed acyclic graphs. The goal is to find the smallest neural network that is able to solve a given problem. The smallest hidden layer is an empty one, so that all the input nodes are directly connected to all output nodes. Starting from the hidden layer, two operations that modify the graph are possible: Adding a layer, or adding a node to one of the existing layers. By using as little graph operations as possible, Listing 3.11 shows how `NNGenerator` uses a free integer variable to branch over

```
Network generateNetwork() {
  return generateNetwork( new Network(4, 2) ); }

Network generateNetwork(Network network) {
  int operation free;
  switch (operation) {
    case 0: // Return current network.
      return network;
    case 1: // Add layer.
      network.addLayer();
      return generateNetwork(network);
    default: // Add node. But where?
      if (network.numberOfLayers > 0) {
        int toLayer free;
        for (int layer = 0; layer < network.numberOfLayers; layer++) {
          if (layer == toLayer) {
            network.addNode(layer);
            return generateNetwork(network);
          } else {
            // Add at a different layer!
          } }
        throw Muli.fail();
      } else {
        throw Muli.fail(); } } }
```

[DK20a]

Listing 3.11: Search region from `NNGenerator` that non-deterministically selects graph operations in order to generate graph structures systematically.

the available operations, adding a third (technical) operation that returns the generated graph. Moreover, adding a node requires additional non-deterministic branching in order to decide to which layer the node is added. In that example, the overloaded method `generateNetwork()` without parameters is the search region that instantiates a network with four input and two output layers. That initial network is passed to the other overloading of `generateNetwork()` that recursively performs the graph operations.

`NNGenerator` fetches the generated graphs from a stream by invoking `Stream<Solution<Network>> solutions = Muli.muli(NNGenerator::generateNetwork);`, passing the search region using a method reference. Obtaining solutions individually from the stream is essential, because `Muli.getAllSolutions()` would not terminate for the lack of a termination criterion in the search region. Graphs are fetched from the stream and transformed

into a Python script that uses PyTorch (cf. [Pas+19]) for implementing, training, and using a neural network according to the generated graph. The neural network is trained and used for solving a specific problem, such as the pole balancing problem, where the aim is to balance a single pole on a movable cart (refer to [BSA83] for details on the problem and to [DK20a] for its implementation). If the performance of a neural network is above a certain threshold, `NNGenerator` terminates search and returns that neural network implementation.

3.6 Summary

The language features presented in this chapter form the programming language Muli. The Muli compiler generates JVMS-compatible bytecode that can be parsed by standard Java bytecode readers and executed by specialized VMs such as the MLVM that is presented in Chapter 4. Language features that do not require a compiler are provided in the Muli runtime library, implemented in Muli, that is on the class path at compile time and execution time. Based on Java, Muli serves as a prototypical reference language for the novel integrated paradigm of constraint-logic object-oriented programming. By integrating non-deterministic search with imperative (object-oriented) programming seamlessly in search regions, Muli and CLOOP facilitate the development of applications that interleave constraint definition and search with imperative statements. Consequently, developers are relieved from the task of maintaining imperative program parts separate from search.

For its backwards compatibility with Java, existing Java applications can be transformed to Muli applications by executing them in a Muli runtime environment. This facilitates augmenting an existing Java application with search using Muli language features.

The Muli compiler[2] and the Muli runtime library[3] are published at GitHub.com under the General Public License (GPL) v3.0. This facilitates future modifications and extensions as well as collaborations with researchers from other groups.

Part II contains publications with more details on the contributions presented in this section, including some evaluations. Specifically, these publications provide additional insights:

1. Chapter 9: Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of*

[2] `muli-lang` at `https://github.com/wwu-pi/muli/`.
[3] `muli-classpath` at `https://github.com/wwu-pi/muli/`.

Computer Languages 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001

2. Chapter 10: Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems.* Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review

3. Chapter 15: Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing.* ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260

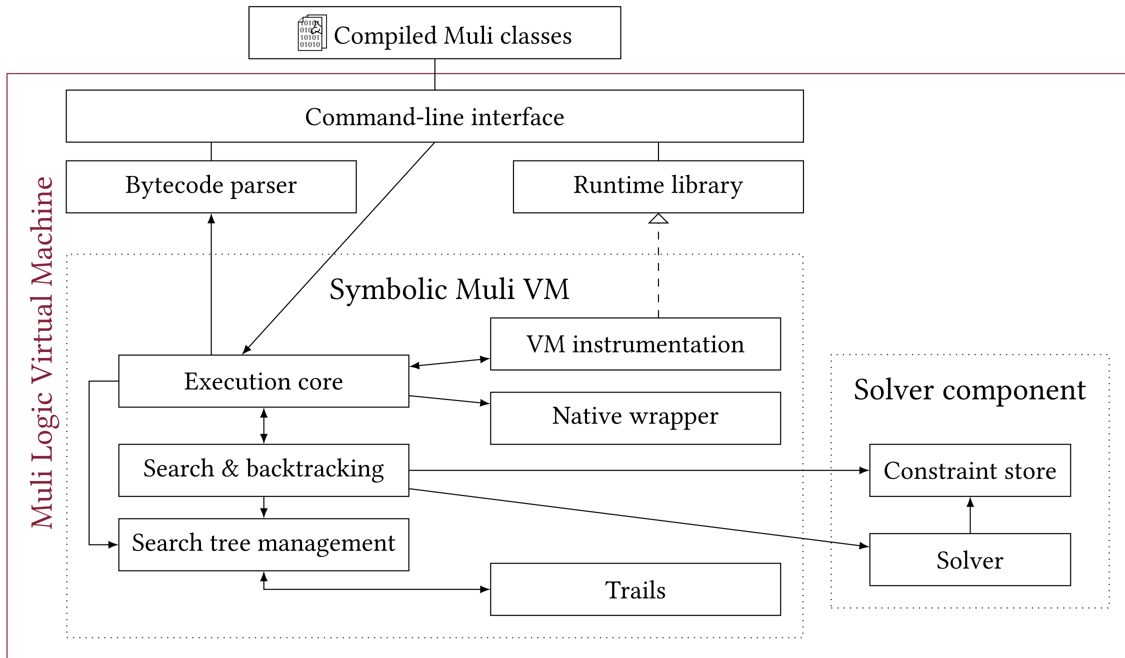
4. Chapter 16: Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5

4 Non-Deterministic Execution of Constraint-Logic Object-Oriented Applications

The Muli Logic Virtual Machine (MLVM) is a runtime environment for the execution of Muli applications. As a customization of a VM for Java applications, it features symbolic execution and non-deterministic search in search regions. Section 4.1 introduces the general concepts of the MLVM and its main components. The representation of symbolic expressions at runtime is explained in Section 4.2, followed by a description of the solver component that derives constraints from symbolic expressions (Section 4.3). Furthermore, Section 4.4 describes the implementation of non-deterministic execution of search regions in the MLVM, including the explicit search tree representation that represents alternative execution paths. Afterwards, Section 4.5 provides details on how the MLVM is able to reverse side effects that result from the execution of (object-oriented) imperative applications, which is a prerequisite for backtracking during search. The behaviour of the MLVM is continuously tested using an automated set of JUnit tests. The test suite is presented in Section 4.6. Last but not least, the contributions of this chapter are summarized in Section 4.7.

4.1 A Virtual Machine for Constraint-Logic Object-Oriented Programs

The MLVM is a runtime environment for the execution of CLOOP applications that are developed in Muli. Since Muli is based on Java, the implementation of the MLVM closely follows the JVMS (cf. [Lin+15]) and adds customizations for the support of the Muli-specific features described in Chapter 3. The MLVM is derived from the symbolic JVM of Muggl (cf. [MK11b]), generalizing it from a domain-specific JVM for test case generation to a general-purpose one for the non-deterministic execution of arbitrary applications [DK18a].



[DK20b]

Figure 4.1: Components of the MLVM.

Muggl's symbolic JVM and, therefore, the MLVM are implemented in Java. This makes the MLVM and Muli just as platform-independent as Java. Moreover, the implementation is able to leverage the plethora of libraries that are available on the JVM, where constraint solvers and libraries for parsing and writing bytecode are particularly useful. In addition to that, running the MLVM inside an actual JVM ensures that the set of basic Java classes from the Java Platform SE API that most Java applications rely on is available as well.

Figure 4.1 illustrates the main components of the MLVM. The MLVM is started using a command-line interface, passing the name of an executable Muli class (i. e., one with a method matching the `public static void main(String[] args)` descriptor known from Java). On start-up, the name of the initial class is passed to the *execution core* that is responsible for the evaluation of Java bytecode. It relies on the bytecode parser component to read the initial class as well as the classes from the Muli runtime library. Moreover, the bytecode parser will be used throughout execution when additional classes are used, such as ones from the Java Platform SE API or classes that belong to the executed application.

In order to maintain the execution state of an application at runtime, the MLVM implements all data structures that the JVMS prescribes for the execution of Java applications [Lin+15, §§2.5 f.]. Therefore, execution state for deterministic and non-deterministic

execution is represented in the MLVM using a combination of a *PC*, a *heap*, *frame stack*, and an *operand stack* for every frame (see Section 2.3). The standard data structures for execution state are all part of the execution core component [DK18a]. Moreover, support for non-deterministic execution requires additional data structures: The *constraint store* maintains the active constraint system, which is the conjunction of all imposed constraints [DK18a]. Moreover, *trails* record all changes that are made to the execution state as a result of side effects, which is a prerequisite for being able to restore a specific execution state during search [DK19b] (Section 4.5 elaborates on this). Finally, using a representation of *search trees*, in combination with a pointer to the currently active tree node, the search tree management component keeps track of the non-deterministic execution of search regions [DT20]. In such a search tree, which is essentially a symbolic execution tree (cf. [Kin76]), inner nodes represent non-deterministic choices and leaves correspond to solutions or explicit failures.

The execution core is responsible for interpreting bytecode instructions at runtime and updates the execution state after the execution of every bytecode instruction. Within search regions, the execution of a bytecode instruction that has non-deterministic behaviour results in the creation of an object representation of the choice, describing the alternatives that it offers (details on this are presented in Section 4.4). The created choice object is then passed to the *search tree management* component, thus updating the search tree that corresponds to the search region that is currently being executed. Afterwards, the execution core delegates the decision for an alternative to the *search & backtracking* component, which adds the corresponding constraint to the *constraint store* and checks whether the resulting constraint system is still satisfiable using the *solver*. If it is not, the search & backtracking component performs backtracking in order to try the next available alternative with a satisfiable constraint system. Once a (satisfiable) alternative is selected, the execution core continues execution on the chosen path.

Methods that are declared `native` do not provide a Muli/Java method body. Therefore, they need to be handled by the MLVM. The *VM instrumentation* methods (in particular, `Muli.fail()`, `Muli.getVMExecutionMode()`, and `Muli.setVMExecutionMode()` from Section 3.4) change the behaviour of the execution core. The *native wrapper* component forwards the remaining native methods to the JVM in which the MLVM is executed.

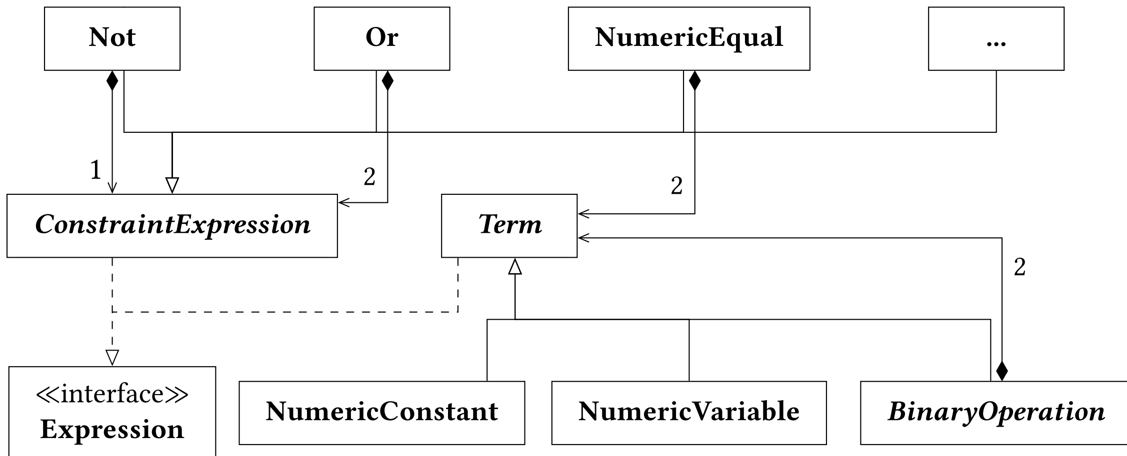
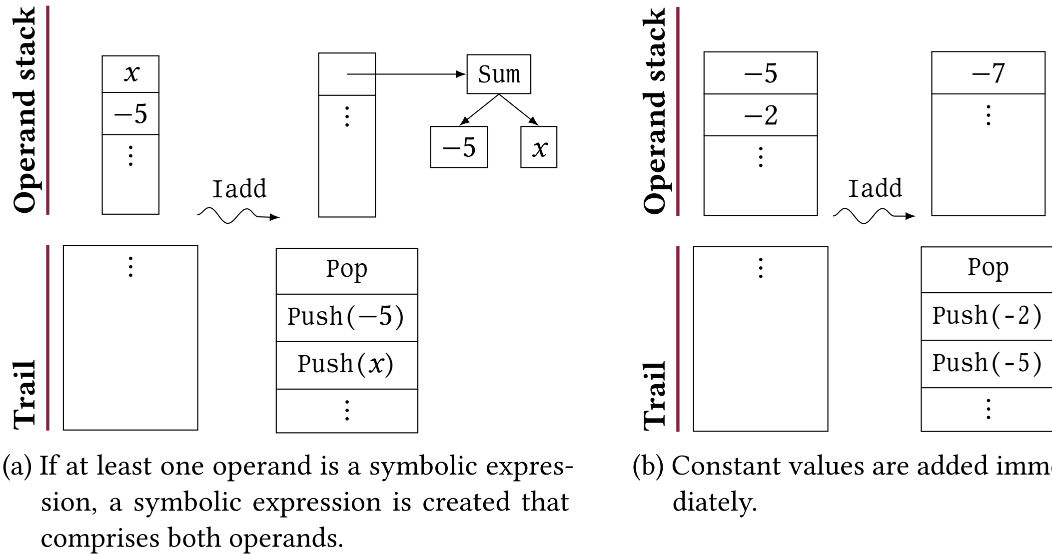


Figure 4.2: Class structure for the representation of symbolic expressions in the MLVM.

4.2 Representation of Symbolic Expressions

In addition to primitive types and reference types of the Java, Muli adds symbolic types for the representation of free variables and for expressions that involve free variables [DK18a]. The symbolic representation of an array uses an instance of the `Arrayref` class that maintains information about the intended type of the array and its elements. Similarly, the symbolic representation of an object is an instance of `Objectref`, that maintains information about field values instead of array elements. Variables of primitive types are represented by `NumericVariable` instances or, for boolean variables specifically, by instances of the `BooleanVariable` class. In `NumericVariable` objects, a flag indicates the particular primitive type. It does not hold a value; instead, its possible values are restricted using constraints in the constraint store.

Figure 4.2 illustrates the class structure for the representation of symbolic expressions involving numeric types, i. e., constants, variables, arithmetic, and boolean operations. Arithmetic expressions (including numeric constants and variables) are represented by subtypes of the abstract class `Term`. Expressions from arithmetic operations on (symbolic) expressions are represented using subtypes of `BinaryOperation`, e. g., `Sum` for an addition of two subterms. The `Term` representations of the subterms are linked to the representation of the operation using the composite design pattern (cf. [Gam+94]). For instance, executing the `Iadd` bytecode instruction for the addition of two integers pops the two topmost elements from the operand stack, adds them, and pushes the result onto the stack afterwards. If one or both popped elements are instances of a subtype of `Term`, the MLVM implementation of `Iadd` constructs a symbolic expression representing the addition using an instance of `Sum`, as illustrated in Figure 4.3a. For the sake of completeness, the effect



(a) If at least one operand is a symbolic expression, a symbolic expression is created that comprises both operands.

(b) Constant values are added immediately.

Figure 4.3: Effects of the `Iadd` instruction on the execution state.

of executing `Iadd` in the case that both operands are constants is depicted exemplarily in Figure 4.3b.

The evaluation of boolean expressions, such as in conditions, is similar. Symbolic expressions will make use of subtypes of `ConstraintExpression`, for instance, the evaluation of an `If_icmpeq` instruction creates an instance of `NumericEqual` for an equality constraint. Constraint expressions created by symbolic execution are not necessarily added to the constraint store. They are only added to the constraint store if the constraint expression is relevant for branching at a choice. The instructions that can potentially create a choice are listed in Table 4.1. If a constraint expression is relevant for branching and is therefore imposed as the constraint for one branch, it is likely that the constraint of the other branch is derived from the original constraint expression. For example, the constraint expression that is created from the condition of an `If_icmpeq` instruction will be used for the first branch and its negation for the second branch.

Note that the presented class structure from Figure 4.2 is only used for internal representations in the MLVM. From a Muli application's point of view, variables of symbolic types can be used interchangeably with variables of regular types as described in Section 3.2. The implementations of bytecode instructions take care that this is actually the case at runtime. For instance, a symbolic expression created as a result of the `Iadd` operation is compatible with the `int` type. Similarly, if the evaluation of boolean expressions involves symbolic expressions (both, terms or constraint expressions), the resulting symbolic expression is compatible with the `boolean` type.

Bytecode instruction	Type of choice	Constraint types
If<cond>, If_icmp<cond>	if instruction, integer comp.	$=, \neq, <, \leq, >, \geq$
Idiv	integer division	$=, \neq$
FCmpg, FCmpl, DCmpg, DCmpl	floating point comparison	$=, <, >$
LCmp	long comparison	$=, <, >$
Lookupswitch, Tableswitch	switch instruction	$=, \notin$

Extended from [DK18a]

Table 4.1: Types of boolean expression constraints that can be generated as a result of evaluating certain Bytecode instructions. <cond> is substituted with either eq, ne, lt, le, gt, or ge.

4.3 The Solver Component

Constraints are added incrementally at runtime when decisions for a choice are taken and, to revoke the decisions during backtracking, need to be removed from the constraint system in reverse order. Therefore, the active constraint system is represented as a stack of constraints. The solver component is responsible for maintaining this constraint stack. Moreover, in order to find solutions for variables that satisfy the constraint system, the solver component leverages the functionality of a solver library. The library that is used for constraint solving is configured during start-up of the MLVM.

Figure 4.4 illustrates the class structure of the solver component. ConstraintExpression is the representation of constraints containing symbolic expressions from Section 4.2. The central interface that the MLVM uses is SolverManager. The MLVM calls hasSolution() in order to determine whether the current constraint system is consistent and, when actual values for the variables are required, the MLVM invokes getSolution() in order to solve the constraint system. Moreover, addConstraint() accepts a new ConstraintExpression that is added to the constraint store. In contrast, removeConstraint() always removes the most recently added constraint. Therefore, the combination of addConstraint() and removeConstraint() realizes the stack structure of the constraint system. That way, Muli is constraint solver agnostic, relying only on the SolverManager interface. This facilitates the integration of additional constraint solvers in the future, since the classes that implement the interface serve as adapters (cf. [Gam+94]) to the actual constraint solvers. The adapters manage the addition and removal of constraints to their respective solver's store and transform Muli-specific ConstraintExpressions to library-specific constraint representations.

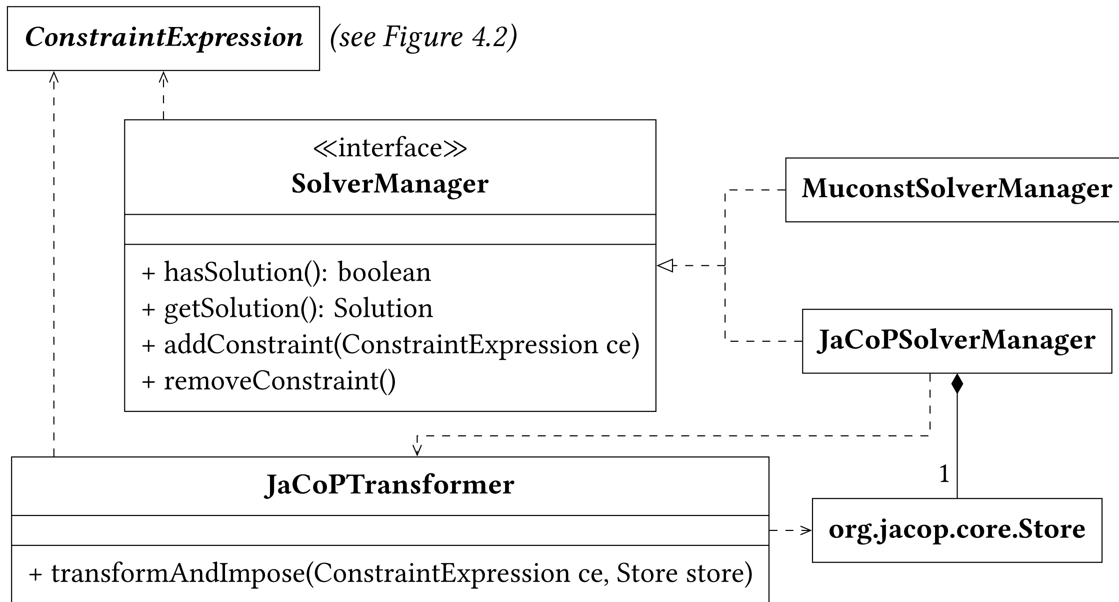


Figure 4.4: Interface and implementations of solver managers that integrate libraries in the solver component.

At the time of writing, the MLVM solver component integrates the two constraint solvers that are introduced in Section 2.4 [DK18a]. First, Muconst is integrated as it is the original solver of Muggl, relying mostly on the Simplex algorithm [EMK12]. Second, the FD solver JaCoP is used because, after adding a new constraint, its use of constraint propagation facilitates the computationally inexpensive detection of whether (FD) constraint systems are rendered unsatisfiable. As a consequence, many infeasible execution branches can be cut off with little effort, as opposed to Muconst's requirement of performing a full run of the Simplex algorithm in order to determine whether an execution branch is infeasible. Therefore, for many applications that mostly use constraints involving finite-domain numeric variables, selecting JaCoP as the underlying solver results in quicker execution. However, the "best" solver depends on the specific application and its constraints. Therefore, it needs to be determined experimentally for every individual application. For that reason, the MLVM offers the facility to configure the underlying solver before it is started, instead of imposing a specific solver on the user.

As illustrated in Figure 4.4, `MuconstSolverManager` and `JaCoPSolverManager` implement the `SolverManager` interface and integrate the respective solvers. The solver manager implementation for JaCoP can serve as an example for future integrations of third-party solver libraries. Particularly relevant is the implementation of a transformer class similar to the `JaCoPTransformer` that is called by the implementation of `addConstraint()` in `JaCoPSolverManager`. The method `transformAndImpose()` accepts constraints that are

represented using Muli's `ConstraintExpressions` and converts them into a representation that uses JaCoP's classes. The JaCoP representation is then immediately added to the active constraint store, which is an instance of the `Store` class provided by the JaCoP library.

4.4 A Structure that Encodes Non-Deterministic Execution Paths

The bytecode instructions that are presented in Table 4.2 exhibit potentially non-deterministic execution behaviour (if they are executed within search regions, see Section 3.2). For instance, the `If_icmpeq` instruction performs a jump by setting the PC to a specified bytecode instruction within the current method if the two topmost values or symbolic expressions (subsequently, e_1 and e_2) on the operand stack are equal [Lin+15, § 6.5]. Otherwise, the PC is moved to the subsequent instruction as usual. Within search regions and if at least one of e_1, e_2 is a symbolic expression, both cases are potentially possible, but not at the same time. Specifically, they can be made equal by imposing an equality constraint $e_1 = e_2$, or made not equal by imposing the inverse $e_1 \neq e_2$. This scenario constitutes a *choice* with two alternatives. Once one of the constraints is imposed, execution can continue accordingly. At a later time, execution of the second alternative can follow, provided that

- all changes to the execution state that happened *after* committing to the first alternative are undone, and
- the constraint of the first alternative is removed from the constraint store.

The implication is that, before selecting the second alternative, the execution state must be exactly the same as it was before the first alternative was chosen.[4] The only exception to that are the search trees that permanently reflect whether choices and their respective execution alternatives have been evaluated yet. Taking decisions at every choice conceptually produces a search tree in which the inner nodes are choices and the leaves represent alternative ends of execution paths, i. e., a symbolic execution tree [Kin76; DT20]. As a consequence, the search tree represents all alternative execution paths.

For every executed search region, the MLVM stores an explicit representation of the corresponding search tree as part of its execution state [DT20]. The explicit representation

[4] Section 4.5 presents details on how a specific previous execution state is achieved.

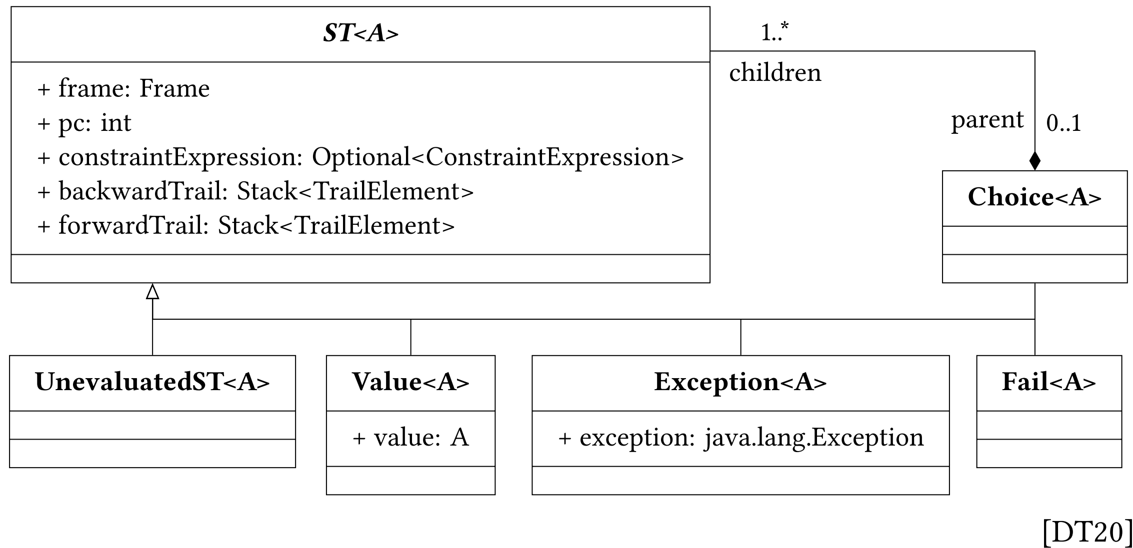
Bytecode instruction	Type of choice	No. of alternatives
`If<cond>`, `If_icmp<cond>`	`if` instruction, integer comp.	2
`Idiv`	integer division	2
`FCmpg`, `FCmpl`, `DCmpg`, `DCmpl`	floating point comparison	2
`LCmp`	long comparison	3
`Lookupswitch`, `Tableswitch`	`switch` instruction	1 per case + 1 default
`Checkcast`, `Instanceof`	instance type check	2
`Invokevirtual`, `Invokeinterface`	method invocation	depends on class path

Adapted and extended from [DT20]

Table 4.2: Bytecode instructions whose execution potentially results in non-deterministic branching. `<cond>` is substituted for specific comparisons, e. g., `eq` for equality.

is useful for debugging non-deterministic execution behaviour, as it allows developers of the MLVM to introspect the generated choices in the search tree at runtime by setting a breakpoint in the MLVM code. However, the search tree structure is primarily used to support search algorithms, keeping track of which decision alternatives have not been evaluated yet.

Inspired by the search tree structure of Curry (see Section 2.1), the search tree of the MLVM distinguishes five types of nodes [DT20]: exceptions that ended execution, values that were returned, choices, failed computations (implicit or explicit), and (as placeholders) subtrees that are not evaluated yet. These types and their relationships are illustrated with the class diagram in Figure 4.5, with `ST` as the abstract supertype of all node types. Two node types describe solutions of a search region, namely a `Value` node holds the value returned by a computation, whereas thrown exceptions are encapsulated in an `Exception` node. Explicit failures or branches whose constraint system is inconsistent are represented by a `Fail` node which, even though it is a leaf node, does not hold a value because it is not a solution. Applying the composite design pattern (cf. [Gam+94]), a `Choice` node stores a list of its subtrees, each of which is of type `UnevaluatedST` at first. In turn, each of the subtrees point to the parent choice node, thus facilitating direct navigation towards the root node. The `parent` attribute of the root node equals `null`. Last but not least, `UnevaluatedST` is a placeholder for subtrees that are not evaluated yet, in order to be able to represent partial search trees. This aids in the construction of the search tree while the search region is executed, instead of having to wait for the exhaustive evaluation of a search region before the tree can be constructed. The intention is to evaluate an instance of `UnevaluatedST` and then immediately replace it



[DT20]

Figure 4.5: MLVM class structure for the representation of search trees.

with an adequate instance of one of the other types. Note that neither of these node types implement their own behaviour, such as decision taking in the `Choice` type. Instead, the intention of these types is to reflect the execution state, so that information about the execution state can be made available, for instance, to search strategies implemented in the MLVM.

With the fields of the abstract supertype `ST`, all nodes of a search tree store data that prepares for later execution starting from that node. Using the `frame` and `pc` fields, the MLVM stores the (mutable) stack frame as well as the PC from the execution state at which the node has been created, thus pointing to the specific instruction whose execution resulted in the node. Optionally, a node stores a constraint expression from non-deterministic branching. That expression must be satisfied in order to evaluate the node. Moreover, a node stores two fields for trails. Trails and the meaning of these fields are described in Section 4.5 and Section 5.1.

The representation of a search tree is created throughout search, i. e., during non-deterministic execution of a search region. A search strategy evaluates `UnevaluatedST` nodes as long as there are such nodes left and the encapsulating program demands additional solutions. The specific order by which unevaluated search tree nodes are selected and, consequently, replaced by their results, depends on which search strategy a Muli application selects (see Sections 3.4 and 5.2).

Generally, the MLVM first chooses an `UnevaluatedST` node. Second, it imposes the node's constraint. If this new constraint renders the constraint store inconsistent, the chosen node is replaced with a `Fail` node immediately. Otherwise, the MLVM executes

the search region starting from the PC that is stored with the node. Execution continues until either

- the search region method body uses a `return` statement to return a value,
- an exception is thrown but not caught,
- the code invokes `Muli.fail()` in order to explicitly fail a branch, or
- an instruction from Table 4.2 is executed with non-deterministic behaviour.

Regardless of the encountered situation, the `UnevaluatedST` node in the search tree is replaced with its evaluated counterpart, i. e., with an appropriate instance of either `Value`, `Exception`, `Fail`, or `Choice`. A fresh `Choice` node initializes all its children with instances of `UnevaluatedST`.

When the execution of a search region begins, the corresponding search tree is unknown. Therefore, the initial search tree is just a single `UnevaluatedST` node that points to PC `0` of a frame for the search region. Moreover, the (optional) constraint expression is empty as well as the trails.

In order to illustrate the construction of a search tree during search, consider the search region with four free variables that is provided in Listing 4.1. For that search region, Figure 4.6 depicts the three representations of the same search tree. The representations have been evaluated to different degrees, corresponding to (intermediate) evaluation stages under the assumption of a depth-first search strategy. If a different search strategy is used, other intermediate stages are possible.

With this explicit search tree representation, the MLVM is better at structuring non-deterministic execution than the symbolic JVM of Muggl. As a structure for non-deterministic execution, Muggl's symbolic JVM uses a stack of objects that represent the choices encountered along a single execution path [MK11a]. As a result, Muggl's support for search strategies is limited to depth-first strategies, and iterative-deepening depth-first search requires search to start from scratch if the search depth is increased, thus re-calculating known solutions. In contrast, the MLVM's search tree represents all execution paths, thus facilitating the implementation of arbitrary search strategies as presented in Section 5.2. Moreover, Muggl mixes the responsibilities regarding search, in that the instruction implementations already take the decision to select the first alternative when they are executed, while creating a representation of the choice. Effectively, this enforces depth-first search. The choice representations in Muggl are only responsible for selecting subsequent alternatives. As a result, the process of taking decisions is implemented redundantly: Once in the implementations of bytecode instructions and once in the choice representations. In contrast, the search-tree based MLVM uses the search

```
String commentOnPasta() {
  boolean spaghetti free, tomatoes free, cheese free, zucchini free;
  if (!spaghetti)
    throw Muli.fail();
  else if (!tomatoes) {
    if (!cheese)
      return "boring";
    else
      return "unhealthy";
  } else if (!cheese) {
    if (!zucchini)
      return "too simple";
    else
      return "vegan&tasty";
  } else
    return "vegetarian&tasty"; }
```

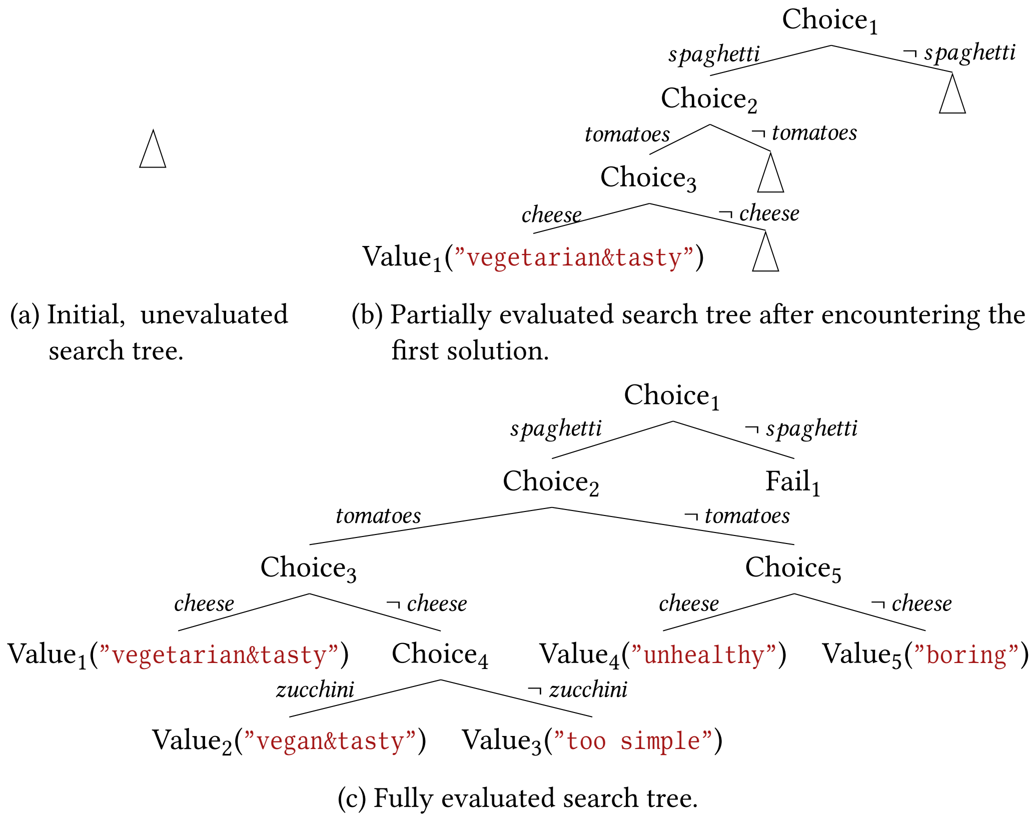
Listing 4.1: Muli search region example that comprises three solutions and a failure.

tree purely as a data structure whereas the selection of branches is only implemented once in the search algorithms. As a consequence, the implementation is cleaner and less redundant.

These benefits do not come at the cost of decreased performance. Experimental results indicate that the execution times of the depth-first search implementation that uses the explicit search tree representation is comparable, if not slightly faster, than those of the original Muggl implementation that relies on the choice stack [DT20]. At the same time, storing the search tree instead of just the current execution path results in higher memory requirements. However, this has not resulted in any problems in experiments yet. In case of problems in the future, an optimization can be to remove subtrees that have been evaluated exhaustively.

4.5 Making Side Effects of Imperative Execution Reversible

An execution path ends when a solution is encountered or when an implicit or explicit failure occurs. As soon as that happens, the MLVM attempts the evaluation of another solution, provided that the current search tree offers a feasible decision alternative that has not been evaluated yet. To that end, the MLVM performs backtracking to the latest



(a) Initial, unevaluated search tree.

(b) Partially evaluated search tree after encountering the first solution.

(c) Fully evaluated search tree.

Figure 4.6: Intermediate stages of the construction of the search tree for the search region from Listing 4.1, assuming a depth-first search strategy. An edge label specifies the constraint of the corresponding subtree.

choice with a feasible alternative. As a prerequisite for taking the alternative decision and following its execution path, the execution state (with the exception of search trees) has to be identical with the state in which the original decision was taken. Otherwise, side effects from the previous execution path would remain in place, thus yielding an inconsistent execution state. If no feasible alternative is left, i. e., if the search region is evaluated exhaustively, the MLVM backtracks until the root of the search tree, thus removing all side effects from encapsulated search.

For the purpose of restoring a previous execution state, the MLVM stores *trail* structures in search tree nodes, as seen in Figure 4.5. The trails in the MLVM are inspired by the trail structure of the WAM. The WAM trail is responsible for recording variable bindings so that they can be unbound on backtracking. Analogously, the MLVM trails record all side effects that have been applied, so that these side effects can be reverted in order to achieve specific previous execution states. However, the MLVM's ability of reverting previous side effects is limited to modifying the MLVM-internal state, i. e., the PC, operand stacks,

frame stacks, constraint store, and the heap. External side effects cannot be undone safely, since other applications that run concurrently are able to make conflicting modifications to the same external state. Invoking an external application is another kind of external side effect that cannot be reversed, as it would require the cooperation of the other application. As a consequence, when a previous execution state is reinstated by changing the core data structures of the MLVM, external side effects will remain in place, such as modifications of files or console output of an application. Nevertheless, performing operations that result in external side effects in search regions is allowed intentionally as there are useful scenarios for irreversible side effects during non-deterministic execution, for instance for logging purposes [DK19a].

The backward trail of a node *n* records all (internal) side effects that were applied after a decision was taken in the parent of *n* until the node *n* was created during execution, thus providing information about state changes that is required for backtracking towards the parent of *n*. As the counterpart of the backward trail, the forward trail of *n* stores state change information that facilitate reaching *n* from the parent of *n*. This facilitates the continuation of search in arbitrary unevaluated subtrees, instead of limiting search to depth-first search. For instance, the backward tree of Choice_1 from Figure 4.6b contains information for undoing changes made to the execution state changes since the beginning of the search region, until the choice was reached. In contrast, the forward trail prepares for re-applying the same state changes for the same part of the execution path. Details on using the forward trail are presented in Section 5.1. In this section, the focus is on backtracking using the backward trail for the purpose of explaining the concept of trails in the MLVM.

The MLVM trails are stacks of trail elements. Executing a bytecode instruction with a side effect results in the application of the side effect as well as in the creation of a trail element that contains partial information of the state prior to the application of the side effect. The created trail element is pushed to the stack of the current backward trail. As a result, later backtracking can use the information stored in the trail element parameters for reverting the field modification. Table 4.3 presents the types of trail elements. Exemplarily, a `FieldPut` trail element is created during the execution of `putfield` and `putstatic` bytecode instructions that modify the value of an instance field, or a static field, respectively. The parameters of `FieldPut` specify the target instance (or class), the modified field, and the value *prior* to the change. As another example, consider the `Pop` and `Push` elements that describe manipulations of the operand stack using Figure 4.3 from earlier. Evaluating the `Iadd` instruction pops two elements from the stack and pushes the result. At the same time, this results in creating two `Push` elements for every popped

Trail element type	Affects execution state	Parameters
`PCChange`	PC, frame	pc
`Restore`	Frame	variable index, value
`FrameChange`	Frame stack	frame
`VmPop`	Frame stack	
`VmPush`	Frame stack	value
`Pop`	Operand stack	
`Push`	Operand stack	value
`PopFromFrame`	Operand stack	
`PushToFrame`	Operand stack	frame, value
`FieldPut`	Heap	instance/class, field, value
`ArrayRestore`	Heap	array, index, value

Adapted from [DK19b]

Table 4.3: Trail element types and their respective parameters.

stack element, containing the previous value in the parameter; and a `Pop` element that will later remove the pushed result.

Assuming a depth-first search strategy and that a solution node v was reached, the MLVM can take a new decision at the most recent choice c_1 that offers a feasible, unevaluated alternative. Under the depth-first search assumption, c_1 is directly or transitively a parent of v. The execution state that was valid during the creation of c_1 can then be restored by using the backward trail of v, popping trail elements and applying the corresponding state changes until the trail is empty. If c_1 is the direct parent of v, the execution state is correct in order to continue with the execution of the alternative. Otherwise, the backward trail of the direct parent c_n is used in the same way, thus achieving the execution state of c_n's parent c_{n-1}. This is repeated until reaching c_1. In any case, the backward trail of c_1 is left untouched and, starting from c_1, the next decision is taken and implemented.

An alternative to using a trail for restoring specific previous execution states is to store snapshots of the entire execution state at every choice by copying the state [DK19b]. However, storing a trail that describes incremental changes is (depending on the application) more memory-efficient [MK11b]. Moreover, it is more reliable since the JVM (and, therefore, the MLVM) does not provide any immutability guarantees, so that application code is able to accidentally modify snapshots after their creation [DK19b]. An ideal compromise would be to maintain the entire execution state, including operand stacks and the heap, in copy-on-write data structures. This would be a prerequisite to storing incremental snapshots of the execution state at every choice. However, classes of the JVM

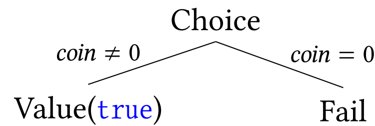


Figure 4.7: Expected search tree for `FailCoin`, as checked by the JUnit test in Listing 4.2.

as well as of arbitrary Muli applications do not reliably provide copy-on-write structures, so that snapshots are not immutable.

4.6 Continuous Testing and Integration of the MLVM

A test suite ensures that the MLVM works as expected, particularly after making changes to its implementation. In fact, several Muli application examples from previous publications now serve as high-level test applications in JUnit tests. Every time a change to the source code is created and pushed to the source code repository at GitHub, these JUnit tests are executed by the Travis CI service.[5] This continuous testing process facilitates the early detection of regressions during development. Therefore, the process ensures that changes to the MLVM that are integrated to the `master` branch on GitHub are stable.

For testing purposes, the MLVM provides a small testing framework. The class `TestableMuliRunner` is essential in this framework. This class is not immediately executed on the command line. Instead, it provides a static method `TestableMuliRunner.runApplication()` that should be called from JUnit tests. `TestableMuliRunner.runApplication()` is invoked with the name of a class that implements a `public static void main(String[] args)` method, i. e., a Muli application. That causes the specified application to be executed. After execution finishes, the method returns the final state of all search trees that were generated during the execution of the application's search regions. In addition, the output of applications started with the `TestableMuliRunner` is redirected to a special string buffer, `TestablePrintStreamWrapper`. Tests can use the string buffer to obtain the output of an application and can check whether the output matches the assertions.

Exemplarily, a JUnit test case of the MLVM is provided in Listing 4.2. It executes the application that is implemented in the class `FailCoin`. When the execution finishes, the test checks whether there is just one search tree, corresponding to the single search region that `FailCoin` implements. Moreover, the test checks that the tree looks like the illustration in Figure 4.7, i. e., that it only has a single choice and two leaves, and that one leaf is a `Value` node, whereas the other one is a `Fail` node.

[5] `https://travis-ci.org/github/wwu-pi/muli`.

```
1  @Test public final void test_FailCoin() {
2    ST[] searchTrees =
3      TestableMuliRunner.runApplication("applications.muliST.FailCoin");
4    assertEquals(1, foundTrees.length);
5    Object[] leaves = foundTrees[0].toArray();
6    assertEquals(2, leaves.length);
7    assertEquals(1, Arrays.stream(leaves)
8      .filter(x -> x instanceof Value).count());
9    assertEquals(1, Arrays.stream(leaves)
10     .filter(x -> x instanceof Fail).count()); }
```

Listing 4.2: JUnit test case that checks whether the MLVM executes a Muli application as expected.

4.7 Summary

The MLVM is a custom VM that is based on the specification for Java VMs and customized in order to support CLOOP features. The MLVM features WAM-inspired trail structures for the purpose of restoring previous execution states, an explicit representation of the structure of non-deterministic execution, and uses solver libraries for finding solutions to constraints that are derived from symbolic execution.

Muli refrains from adding Muli-specific or solver-specific syntax for the specification of constraints and uses Java syntax instead. Consequently, the solver component is responsible for transforming the constraints into a form that is appropriate for the solver that is selected for the execution of a Muli application. As a result, the solver component makes Muli applications solver-agnostic, avoiding a lock-in effect with specific solver libraries.

The explicit structure of non-deterministic execution encodes the intermediate and final states of the execution of a search region. This structure is useful for developing different search strategies for encapsulated search, see Section 5.2. Moreover, it aids in debugging the MLVM as it facilitates introspection of execution states, for example when a debugging breakpoint is hit (thus making intermediate stages of the search tree visible) or after execution of a Muli application ends (thus investigating the final search tree).

Like the other components of Muli, the MLVM is published at GitHub.com under the GPL v3.0.[6] That repository also contains the modifications that are presented in subsequent chapters.

[6] muli-env at https://github.com/wwu-pi/muli/.

Publications that present details on the concepts and components of the MLVM, as well as their implementations, are provided in Part II. The following publications are particularly relevant in the context of this chapter.

1. Chapter 12: Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13

2. Chapter 13: Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing.* Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912

3. Chapter 15: Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing.* ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260

4. Chapter 16: Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5

5 Search in a Constraint-Logic Object-Oriented Language

In the context of object-oriented programming, search by non-deterministic execution is particularly challenging. As opposed to declarative languages, such as Curry [HKM95] or Prolog [War83], the execution of object-oriented programs results in side effects and is, therefore, stateful. Consequently, search in CLOOP has to consider the execution state carefully. Subsequently, Section 5.1 describes how the MLVM trails are used to interrupt and resume encapsulated search, thus facilitating the retrieval of individual solutions even from search regions with a theoretically infinite number of choices. Moreover, the combination of trails and the explicit search tree structure allow the implementation of arbitrary search strategies as described in Section 5.2. Lastly, Section 5.3 provides a chapter summary.

5.1 Interrupting and Resuming Search

There are search regions that create an infinite number of non-deterministic choices. In theory, the non-deterministic execution of such search regions in full would not terminate; in practice, it would terminate as soon as the main memory is exhausted [DK19b]. Therefore, calculating all solutions before returning them to the surrounding, deterministic application is not always feasible. As an example, consider the Muli methods in Listing 5.1 that branch over the free variable `coin`. The methods do not use a termination criterion. As a consequence, the corresponding search tree (theoretically) has an infinite depth and an infinite number of solutions.

It is desirable that applications that formulate such search regions can be executed, regardless of the theoretically infinite number of choices. After all, the example from Listing 5.1 only uses ten solutions eventually. Moreover, even for problems with a finite but large number of solutions it may be impractical to explore the search space exhaustively. Therefore, Muli needs a mechanism that calculates individual solutions

```
1  void powersOfTwo() {
2    Stream<Solution<Integer>> powers = Muli.muli(() -> {
3      return twoPower(0); });
4    powers.limit(10).forEach(System.out::println); }
5
6  int twoPower(int y) {
7    boolean coin free;
8    if (coin) return Math.pow(2, y);
9    else return twoPower(y+1); }
```

Adapted from [DK19b]

Listing 5.1: Muli code excerpt that uses non-deterministic evaluation in order to generate all powers of two for non-negative integer exponents.

from encapsulated search, resuming search for further solutions on demand, while at the same time retaining deterministic execution behaviour of the surrounding program parts.

Using a single trail per choice would be sufficient in order to implement local (or partial) backtracking to the most recent choice that offers another alternative. However, that would only cater to scenarios that use depth-first search only and that are able to exhaustively evaluate all solutions before returning to the main program. Alternatively, instead of evaluating all solutions, it is simple to end search after n solutions if n is known before search is started, but resuming search for additional solutions is hard. Neither of these scenarios is acceptable in Muli applications. For this reason, the MLVM maintains two complementary trails per choice, backward trail and forward trail, thus preparing for arbitrary jumps between execution states instead of just backtracking. This is an integral part of providing a way to stop search immediately after a solution is found, while maintaining the ability to resume it on demand.

5.1.1 Using Dual Trails for the Retrieval of Individual Solutions

Recall that a trail is a stack of trail elements, each of which describes an operation on the execution state. The operation described by a trail element is the inverse of an original change to the execution state, and the trail element exists for the specific purpose of reverting that original change. Refer to Section 4.5 for a description of how the backward trails of search tree nodes are built during the execution of a search region. In contrast, the forward trail of a search tree node is the inverse of that node's backward trail, meaning that the elements on the forward trails are inverses to the elements on the backward trails. Moreover, since both trails are stack structures, the order of the elements on the

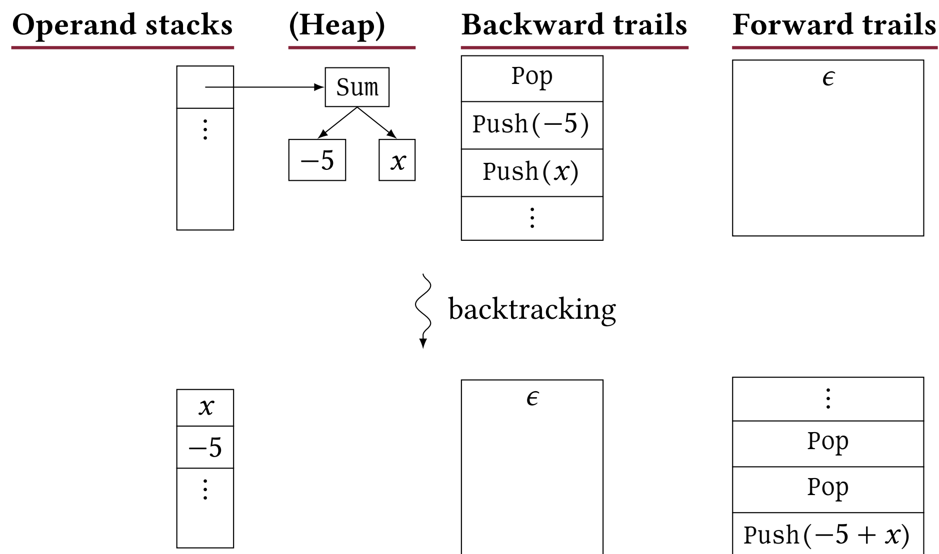


Figure 5.1: Processing the backward trail in order to establish a former execution state while creating the corresponding forward trail.

forward trail is reversed compared to their inverse counterparts on the backward trail. Exemplarily, Figure 5.1 illustrates how processing the backward trail pops the symbolic expression $-5 + x$ from the operand stack, while creating a corresponding `Push` trail element on the forward trail in order to be able to restore the symbolic expression on the operand stack at a later stage. Table 5.1 specifies the inverse trail element types for every element type, thus showing that there are inverses for all trail element types.

The two trails of a search tree node are complementary in that at least one of the two is always empty (unless the trails are being processed, in which case both trails hold elements temporarily). While the elements on the backward trail are (at first) generated during execution of a search region, elements for the forwards trail are created while the backward trail is processed. Subsequently, processing the forward trail also results in re-creating the backward trail. Using a forward trail in that direction is useful, because otherwise the bytecode instructions would have to be executed again. This is not desirable, because re-executing the same bytecode instructions does not necessarily result in the exact previous state. Consider, for example, instructions that rely on state outside the MLVM, such as an instruction that invokes a method which reads content from a file whose contents have changed in the meantime.

With the two trails in place, the MLVM can perform full backtracking after finding a solution, i. e., revert the execution state to the state from *before* starting encapsulated search by processing all backward trails from the most recent solution node and of its parents, until (and including) that of the root of a search tree. Recall the example search

Trail element type	Affects execution state	Inverse type
`PCChange`	PC, frame	`PCChange`
`Restore`	Frame	`Restore`
`FrameChange`	Frame stack	`FrameChange`
`VmPop`	Frame stack	`VmPush`
`VmPush`	Frame stack	`VmPop`
`Pop`	Operand stack	`Push`
`Push`	Operand stack	`Pop`
`PopFromFrame`	Operand stack	`PushToFrame`
`PushToFrame`	Operand stack	`PopFromFrame`
`FieldPut`	Heap	`FieldPut`
`ArrayRestore`	Heap	`ArrayRestore`

Extended from [DK19b]

Table 5.1: Trail element types and their respective inverses.

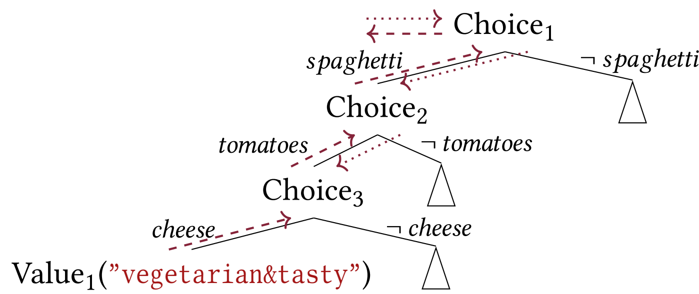


Figure 5.2: Using backward trails (dashed) and forward trails (dotted) in order to achieve specific execution states.

region from Listing 4.1. After the first solution has been found, the backward trails (each indicated with a dashed line in Figure 5.2) are processed in order to achieve the execution state from before search. This leaves the application in an execution state in which it can accept the found solution, and in which it can trigger the MLVM to resume search in the search region. Once another solution is requested, the MLVM will then resume search by processing the newly created forward trails (dotted lines in Figure 5.2) in order to establish the execution state of the choice that provides the next feasible alternative.

By using the combination of backward and forward trails at every search tree node, as well as the stored PC and frame (cf. Section 4.4), the MLVM is able to navigate in the search tree arbitrarily while ensuring a consistent execution state [DT20]. As a result, the MLVM is no longer restricted to performing backtracking only. Therefore, the term *navigating upwards* is used as a synonym for backtracking, i. e., a navigation in the direction from a leaf or inner node towards the root of the tree, whereas *navigating downwards* describes

```
1  void navigateUpwards(ST from, Choice to) {
2    while (from != to) {
3      if (from.constraintExpression.isPresent())
4        vm.constraintStack.pop();
5      vm.processTrail(from.backwardTrail, from.forwardTrail);
6      vm.setFrame(from.frame);
7      vm.setPc(from.pc);
8      from = from.parent; } }
```

[DT20]

Listing 5.2: Navigating upwards in a search tree.

the opposite direction. Certain search strategies, such as breadth-first search, will use a combination of both directions (see Section 5.2). In upwards direction, the target of navigation is always a choice node, since inner nodes of the search tree are choices. In downwards direction, the target is also a choice, because navigating downwards is only sensible for the purpose of reaching an unevaluated subtree that is the child of the target choice (and then evaluating the subtree afterwards).

The MLVM navigates upwards from a search tree node `from` to a target node `to` navigation via the node's parent references, as outlined in Listing 5.2. The parameter `to` must be set to `null` in order to revert all effects of encapsulated search because `null` is the parent of the root node. Alternatively, `to` may be any node that is a (transitive) parent of `from`. While moving towards the target node, the execution state is reverted using the information stored at each visited node: If the evaluation of a node has required a constraint, that constraint is removed from the solver manager's constraint stack. Moreover, `processTrail()` uses the elements on the node's backward trail in order to revert the effects on the execution state. Afterwards, frame and PC are explicitly set to the state in which the visited node was created. Note that `processTrail()` has two parameters. The first parameter is the trail that contains information on how to revert execution state. While processing the elements on that trail (and changing execution state in the process), `processTrail()` creates appropriate inverse elements and pushes them to the trail that is passed in the second parameter.

Downwards navigation requires the same steps, but in reverse order. Additional effort is needed initially in order to find the path from the source to the target node, as the search tree describes the path in reverse order, i. e., from target to source. To that end, the downwards navigation method first records the nodes along the path on a stack (see Listing 5.3). Afterwards, the stack is used to visit nodes from the path and to restore

```
1  void navigateDownwards(Choice from, Choice to) {
2    Stack<ST> nodes = new Stack<>();
3    while (to != from) {
4      nodes.put(to);
5      to = to.parent; }
6    while (!nodes.empty()) {
7      to = nodes.pop();
8      vm.setFrame(to.frame);
9      vm.setPc(to.pc);
10     vm.processTrail(to.forwardTrail, to.backwardTrail);
11     if (to.constraintExpression.isPresent())
12       vm.constraintStack.push(to.constraintExpression.get()); } }
```

[DT20]

Listing 5.3: Navigating downwards in a search tree.

execution state according to the data stored in the nodes. Specifically, frame and PC are set first, followed by using `processTrail()` to process the forward trail (while re-creating the backward trail in the process). Finally, if a node requires a constraint, that constraint is imposed by adding it to the constraint stack.

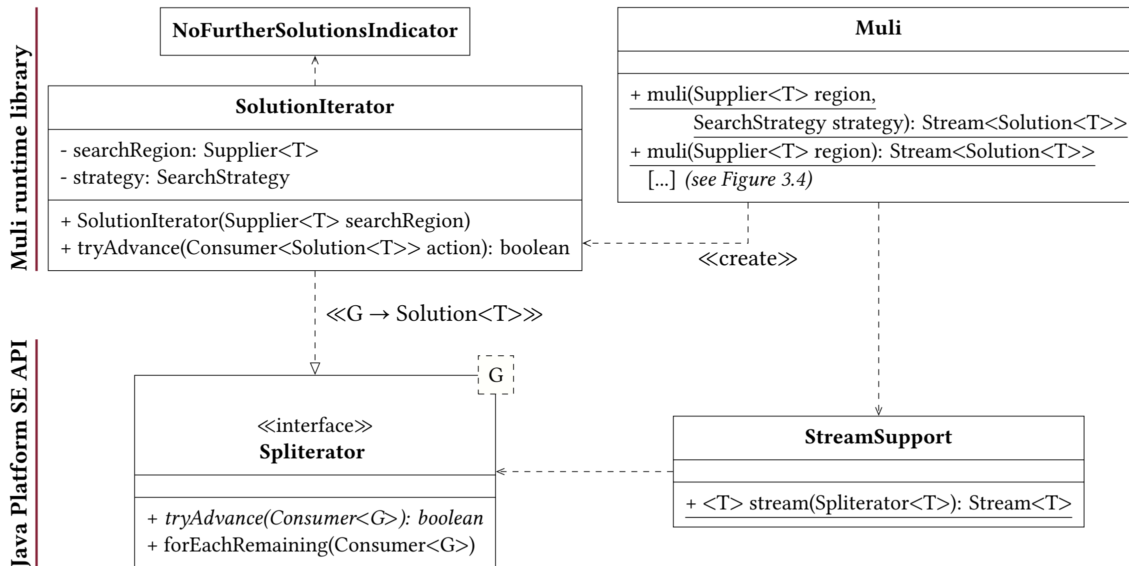
5.1.2 Obtaining Individual Solutions in Muli Applications

For the purpose of retrieving individual solutions and passing them to the surrounding application, the MLVM internally uses `navigateUpwards()` to navigate to the root once a solution leaf node is encountered, while memorizing the most recent choice along the path that offers another alternative. Accordingly, when the application requests another solution of a search region, the MLVM uses `navigateDownwards()` in order to navigate from the root to the choice that was memorized earlier.

Externally, i. e. as the API for Muli applications, the Muli runtime library offers the `Muli.muli()` operator that returns a stream from which individual solutions can be obtained. According to the Stream API documentation, streams are sequences of elements [Ora20c]

- that are of (potentially) infinite length, and
- whose elements are not evaluated unless they are consumed individually.

Moreover, a consumer of a stream can consume every element from that stream only exactly once. This has two consequences [DK19b]. First, a stream is an appropriate representation for returning solutions of Muli search regions, as there is an unknown and



Adapted from [DK19b]

Figure 5.3: Relationships between the stream-related classes of the Java Platform SE API and the encapsulated search operator `Muli.muli()`.

potentially infinite number of solutions. Second, as soon as a solution has been returned, the MLVM does not need to produce it again. Furthermore, since the Stream API requires stream implementations to follow the Iterator behavioural pattern (cf. [Gam+94]), it does not require the MLVM to produce solutions in a certain way or order.

In the Muli runtime library, the main encapsulated search operator `Muli.muli()` accepts a search region as a parameter and returns a stream of `Solution` objects. As a second parameter the search strategy can be specified. This second parameter is optional by overloading. Figure 5.3 illustrates the relationship between the public API provided by `Muli.muli()` and its supporting classes that come from either the Java Platform SE API or from the Muli runtime library.

In the creation of the stream object, `Muli.muli()` relies on the helper method `StreamSupport.stream()` that accepts an implementation of an iterator. That iterator class must implement the interface `Spliterator<G>` from the Java Platform SE API, where the type parameter `G` is the type the objects returned by the iterator. In this case, `G` is specialized to `Solution<T>`, because the MLVM will wrap returned solutions in objects of that type.

The Muli runtime library implements the `Spliterator` interface in the class `Solution-Iterator<T>`. That class is responsible for controlling the MLVM execution behaviour regarding encapsulated search. The method `tryAdvance()` is called (directly or indirectly)

from the terminal operation of the stream pipeline. Since the number of elements in the stream is unknown, that method either returns `false` if no additional element could be computed, or `true` otherwise, passing the computed element to the `action` consumer that has been passed to the `tryAdvance()` method. `action` is a consumer that corresponds to the next operation in the stream pipeline.

In order to search for a solution of the search region, `tryAdvance()` instructs the MLVM to switch to non-deterministic execution as described in Section 3.4 and Listing 3.8. When the first element is requested, `tryAdvance()` invokes `searchRegion.get()`, i. e., the search region's functional interface method. For subsequent elements, the MLVM is instructed to change the execution state to that of the next choice with a feasible alternative by using `navigateDownwards()`, and to continue execution from there. In either case, a found solution is passed to the `action` consumer, after using `navigateUpwards()` to remove all effects from search and setting the MLVM execution mode to the previous mode. If search ends without finding a solution, the MLVM creates an object of the special `NoFurtherSolutionsIndicator` type from the Muli runtime library. This object serves as an indicator that is interpreted by the `tryAdvance()`, thus causing it to return `false`.

5.2 Search Strategy Selection at Runtime

The dual trails at every search tree node are also useful to implement arbitrary search strategies for the traversal of the search tree of a search region. The search strategies also leverage the `navigateDownwards()` and `navigateUpwards()` methods presented in Section 5.1 [DT20]. Muli applications can choose a search strategy that will be used for a search region by passing an appropriate second parameter to any of the encapsulated search operators. Allowed values for the second parameter are instances of the enumeration type `SearchStrategy` from the Muli runtime library (see Figure 3.4); specifically, `SearchStrategy.DepthFirstSearch`, `SearchStrategy.BreadthFirstSearch`, or `SearchStrategy.IterativeDeepening`. These three strategies are currently implemented in the MLVM.

Since the search strategy can be selected on a per-search-region basis, the MLVM instantiates a search strategy class for every search region, indicated with the composition relationship between `LogicVirtualMachine` and `AbstractSearchAlgorithm` in Figure 5.4. The search strategy instance is then responsible for managing search-related execution state; namely, the search tree and the tree node that is currently under evaluation. The `AbstractSearchAlgorithm` supertype implements functionality that is shared across all

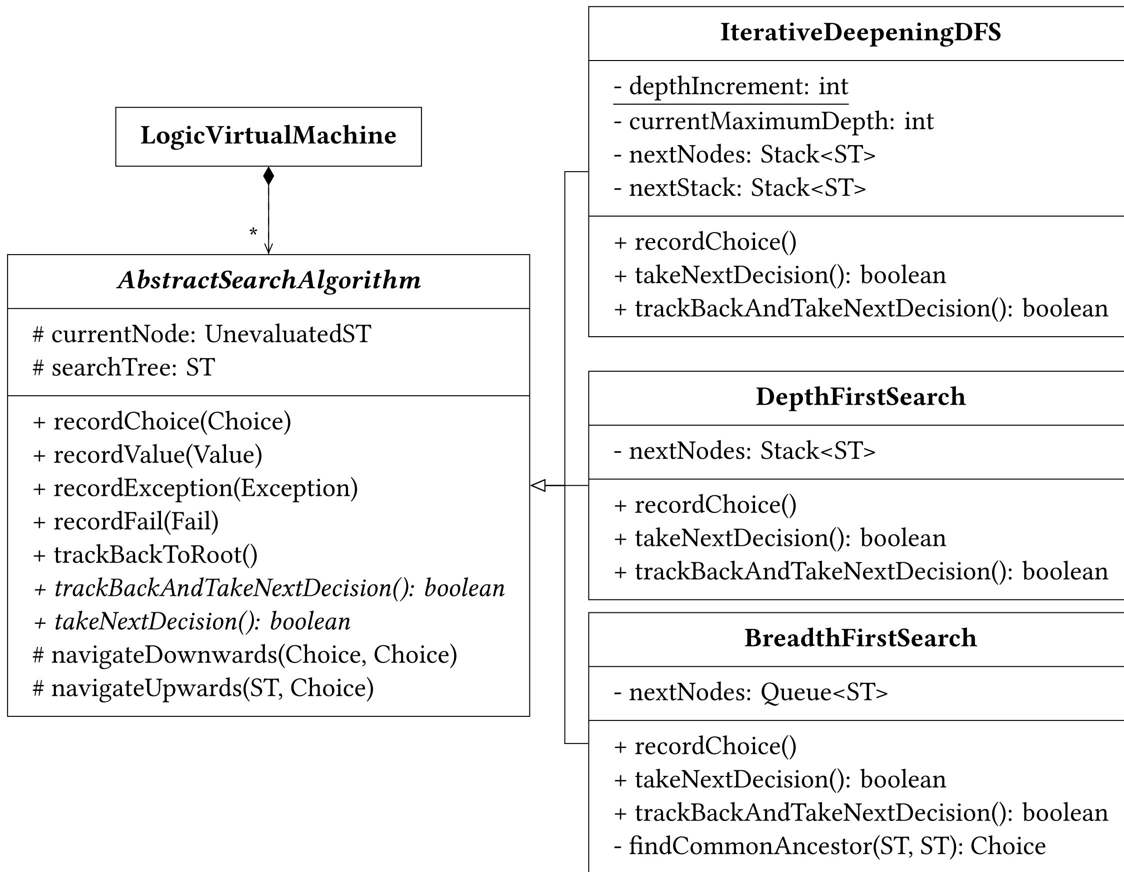


Figure 5.4: Simplified class structure showing the search strategy implementations and their relation to the VM class.

strategies. In particular, these are fields that represent the current evaluation state of the search tree as well as the current node, and methods that update the search tree by replacing the current node with a found node (`record<Nodetype>()`, where `Nodetype` indicates the type of the found node). Furthermore, `AbstractSearchAlgorithm` implements the navigation methods from Section 5.1 so that they are available to the strategy implementations, as well as a `trackBackToRoot()` method that the MLVM uses when a solution is found.

Two methods are declared as abstract methods because they will be invoked from the MLVM, but their implementation depends on the specific strategy. The method `trackBackAndTakeNextDecision()` is used when a failure is encountered *during* search, so that the strategy immediately tries to evaluate the next feasible alternative. In contrast, `takeNextDecision()` is invoked when search is *resumed*. Both methods return `false` if no further alternative is available and `true` otherwise. The strategy implementations are presented subsequently.

Depth-first search In order to perform depth-first search, the strategy implementation in the class `DepthFirstSearch` stores unevaluated subtrees for future evaluation on the `nextNodes` stack (see Figure 5.4). When the strategy class is instantiated, the instance of `UnevaluatedST` that represents the entire search tree is pushed to the stack [DT20]. Moreover, `recordChoice()` is overridden so that, in addition to replacing the node under evaluation with the found choice, all subtrees corresponding to decision alternatives are pushed to the `nextNodes` stack. In order to resume search in `takeNextDecision()`, the implementation pops the next unevaluated subtree from the `nextNodes` stack and uses `navigateDownwards()` to provide the appropriate execution state. `trackBackAndTakeNextDecision()` also pops the next subtree from the stack, but since it is invoked during search when a leaf is encountered it uses `navigateUpwards()` to go to the next parent that offers another alternative (if any).

Breadth-first search Breadth-first search is novel in the context of (non-deterministic) execution of imperative or object-oriented programs [DT20]. The `BreadthFirst-Search` class uses a first-in-first-out queue of unevaluated subtrees. When search is started or resumed, the next subtree is taken from the head of the queue, whereas the decision alternatives of encountered `Choice` nodes are appended at the end of the queue. The `recordChoice()` method is overridden in order to add the decision alternatives to the queue, but also to take the next decision according to the strategy. As opposed to depth-first search, navigation is not strictly downwards or upwards only: When a failure is encountered or when a choice is encountered, `trackBackAndTakeNextDecision()` may navigate to a subtree that is not the direct subtree of a parent node, but to a subtree that is within an entirely different subtree. Even though navigation via the root is possible, it may be more efficient to find a shorter path via a closer common ancestor node as illustrated in Figure 5.5. In the illustration, the current node Choice_4 has just been encountered and the next node for evaluation is the subtree of Choice_5 that has not been evaluated yet. Since the execution state has to remain consistent, switching from the current node to the next one has to be performed using the backward and forward trails. An inefficient implementation could navigate upwards until the root using the backward trails, followed by navigation downwards towards the new node. However, in the example, using the backward trail to revert the execution state to that of Choice_1 only to immediately change it back to that of Choice_2, is unnecessarily inefficient (red sequence of arrows). Finding a shorter path (blue sequence) avoids such redundant usage of trails. The simple method in Listing 5.4 is used to

```
Choice findCommonAncestor(ST a, ST b) {
  Set<ST> set = {};
  while (b != null) {
    set.add(b);
    b = b.parent; }
  while (!set.contains(a))
    a = a.parent;
  return a; }
```

Adapted from [DT20]

Listing 5.4: Calculation of the closest common ancestor of two nodes.

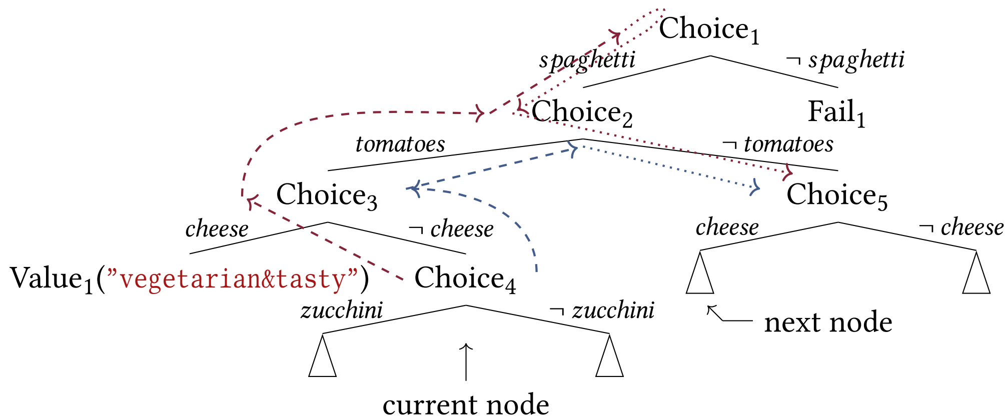


Figure 5.5: Efficient navigation between two nodes in arbitrary subtrees is possible via the closest common ancestor (blue path) instead of navigating via the root node (red path).

find the first common ancestor of two nodes in the search tree. It never returns `null` since the root node can always be an ancestor. Once that ancestor is found, `trackBackAndTakeNextDecision()` combines `navigateUpwards()` to set the execution state to that of the found ancestor with `navigateDownwards()` in order to prepare the execution state for the next subtree.

Iterative deepening depth-first search Iterative deepening bounds depth-first search at a maximum tree depth that can be increased if necessary [Kor85]. In a different form, iterative deepening depth-first search was already applied to the (non-deterministic) execution of imperative or object-oriented programs, specifically, in test case generation with Muggl [MK11b]. However, that implementation requires starting search over at the root if the maximum depth is increased, thus recomputing solutions that have already been found. In contrast, the explicit representation

```
1  boolean nonTerminatingCoin() {
2    int coin free;
3    if (coin == 0)
4      return true;
5    else
6      return nonTerminatingCoin(); }
```

Adapted from [DT20]

Listing 5.5: Search region that theoretically produces a search tree of infinite depth.

of the search tree, with trails and `UnevaluatedST` as placeholders for unevaluated subtrees, facilitates a more efficient implementation that does not have to recompute known solutions [DT20]. In the `IterativeDeepeningDFS` implementation, two stacks are maintained: `nextNodes` is used as described for depth-first search, but contains only unevaluated subtrees with a depth that is less or equal to the current maximum depth stored at `currentMaximumDepth`. In contrast, subtrees with a greater depth are pushed to the `nextStack` stack. Once the `nextNodes` stack is depleted, `currentMaximumDepth` is increased by the `depthIncrement` constant. Moreover, the stack in `nextStack` becomes the new `nextNodes` stack, thus providing new candidates for evaluation. The strategy implementation overrides the `recordChoice()` method in order to implement this behaviour.

The ability to select a strategy from Muli application code facilitates the execution of search regions for which depth-first search is counterproductive. For example, consider a search region as depicted in Listing 5.5 that branches over a variable and that uses recursion in one of the branches. The result is infinite recursion and, conceptually, an infinite search tree. If the depth-first search strategy is implemented to always evaluate the `else` branches first, the MLVM can never return a single solution as that is the branch that invokes the search region recursively. However, developers are able to determine manually that this is a potential problem and can choose to employ a different search strategy accordingly. As a result, a Muli application that uses this search region is able to obtain solutions from search. In order to prove this, an experiment was conducted with two search regions for which depth-first search is deficient [DT20]. The results are reproduced in Table 5.2, demonstrating that the breadth-first and iterative-deepening strategies are able to produce solutions when depth-first search is not.

	Depth-first	Breadth-first	Iterative deepening
Simple infinite recursion	0	1469.7	1555.2
Water jug problem	0	29.5	34.4

Adapted from [DT20]

Table 5.2: Comparison of search strategies using two search regions, w. r. t. the average number of solutions that are found and returned within ten seconds.

5.3 Summary

With the extension of the trail concept from Section 4.5 towards a set of dual trails per search tree node, trails provide benefits for CLOOP in at least two ways. On the one hand, the dual trails are a prerequisite for stopping and resuming encapsulated search in a stateful execution environment. On the other hand, they can be combined with the explicit search tree structure presented in Section 4.4, serving as a basis for the implementation of arbitrary search strategies.

In contrast to the depth-first search strategy that was already implemented in Muggl and could therefore be re-used in Muli (but has been rewritten to use the explicit search tree representation, see Section 4.4), the other search strategies are novel for CLOOP in general and Muli in particular. Breadth-first search can only be implemented with the described data structures in place, whereas iterative deepening depth-first search is novel in the efficiency of its implementation since it does not recompute previously known solutions after increasing the maximum search depth. As an outlook, the data structures allow the implementation of further search strategies. For instance, an interactive strategy could interrupt the execution of a Muli search region whenever a search tree node is encountered, thus allowing the user to choose which subtree to descend into. This kind of interactivity could be useful for debugging during the development of Muli applications.

Presumably, these results can be generalized for future implementations of other CLOOP languages. The combination of an explicit search tree representation that encodes the state of search, and trails that encode differences in execution state between search tree nodes, serves as a bluebrint for the development of VMs for other CLOOP languages. As a result, these languages would also benefit from stopping and resuming encapsulated search, thus facilitating the retrieval of individual solutions, and could offer arbitrary search strategies to application developers as well.

The following publications from Part II go into more detail regarding search in CLOOP and Muli, thus providing additional insights that complement this chapter:

1. Chapter 10: Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems.* Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review

2. Chapter 12: Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13

3. Chapter 13: Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing.* Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912

6 Free Objects

Syntactically, a logic variable can be of an arbitrary type. However, variables of reference types allow more kinds of interactions than variables of primitive types. Consider, for example, the possibility of invoking a method on an object. As a consequence, the introduction of *logic* variables of reference types into a programming language requires a careful definition of how interactions with such variables are treated at runtime. Section 6.1 discusses the different kinds of reference types that are defined by the JVMS. Afterwards, Section 6.2 names and defines specific interactions with free objects and discusses their treatment and implementation in the MLVM. Finally, the chapter is concluded with a summary in Section 6.3.

6.1 Reference Types

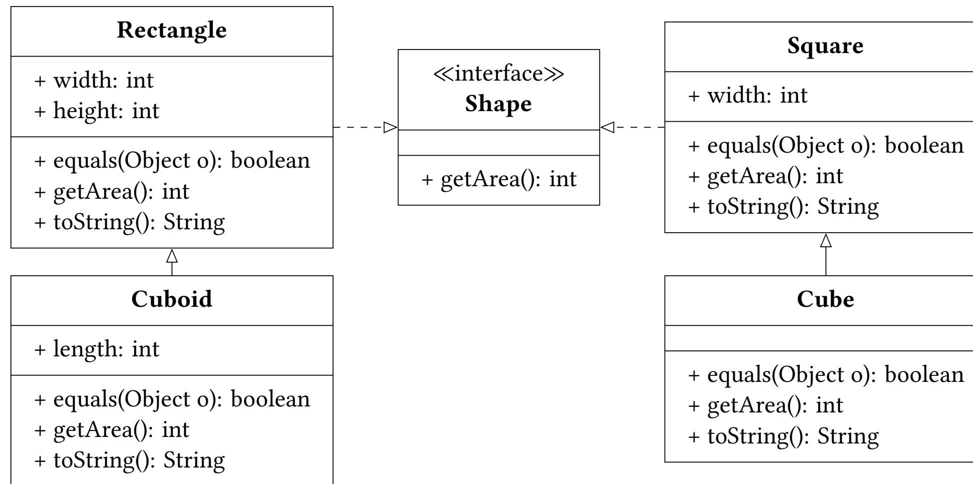
According to the JLS, reference types are distinguished into [Gos+15, § 4.3]

- class or interface types,
- array types, or
- type variables.

As Muli is based on Java, Muli follows this distinction.

Type variables are fundamentally different from the other reference types [Dag19]. A type variable is the result of a type parameter in the declaration of generic types, e. g., `E` in `ArrayList<E>` [Gos+15, § 4.4]. Therefore, at runtime, type variables never correspond to an object on the heap at runtime, whereas variables of the other reference types are either `null` or point to an object on the heap. By excluding type variables from further considerations regarding reference types, the remaining definition of reference types is congruent with that of C# [Mic20]. Therefore, the findings on reference-type logic variables presented in this chapter could be transferable to a future CLOOP language based on C#.

This chapter describes the implementation of support for class-type or interface-type logic variables, or short: *free objects*. Adding support for array-type logic variables (or free arrays) is left for future work.



[DK20b]

Figure 6.1: Application class structure comprising an interface and four implementations.

```
1 String search() {
2   Shape s free;
3   if (s.getArea() == 16)
4     return s.toString();
5   else
6     throw Muli.fail(); }
```

Adapted from [DK20b]

Listing 6.1: Search region that invokes a method on a free object.

6.2 Types of Interaction with Free Objects

By facilitating the use of objects as logic variables, i. e. free objects, it is possible to specify constraints over entire objects, for example regarding their types or field values. Furthermore, it is important to keep in mind that objects in object-oriented programming languages encapsulate data *and* behaviour. For example, consider an application's class structure as illustrated in Figure 6.1 in combination with the search region in Listing 6.1. The variable `s` is a free object of type `Shape`, but `Shape` is an interface type. Therefore, the actual type of `s` can be any subtype of `Shape`, each of which implements `getArea()` according to its respective needs. In general terms, the actual object behind a declaration `C o free;` can be of any type that is type-compatible with `C`, i. e., `C` itself, or classes or interfaces that are subtypes of `C`. As a consequence, the declaration of a free object `o` only provides partial information about the actual type of specific objects that would meet the constraints over `o`.

The possible types that a free object may assume also depend on the types that are available to an application at runtime. In Java and Muli, types may be available even though they are not (yet) loaded into main memory, because the class loader usually loads the classes from the filesystem (specifically, from the class path) when they are first used [Lin+15, § 5.3.5]. Therefore, in order to make use of the full type hierarchy in interactions with free objects, the MLVM has to discover the classes that are available on the class path prior to their first explicit use [DK20b].

Subsequently, various specific kinds of interactions with free objects are detailed, namely,

- their instantiation (and initialization) in Subsection 6.2.1,
- accesses to their fields of a free object in Subsection 6.2.2,
- method invocations in Subsection 6.2.3,
- type operations in Subsection 6.2.4, and
- finally, in Subsection 6.2.5, equality checks on free (and regular) objects.

6.2.1 Instantiation and Initialization

Consider a free object declaration `C o free;`. Subsequently, `o` is available as a symbolic object that the MLVM internally represents using the `Objectref` type (see Section 4.2). As a consequence, the free object `o` is instantiated as a fresh object instance that can be used in all expressions that syntactically allow the use of objects [DK20b]. A specific object, however, is not yet known; i. e., the free object does not hold any data, its actual type is unknown, and no constructors are invoked.

Fields that occur in the definition of `C` are initialized with free variables (according to their respective type). For instance, against the background of the class structure in Figure 6.1, a declaration of `Rectangle rect free;` would result in a symbolic object with two fields, `width` and `height`, that both are free integer variables. An important exception are fields that the type definition of `C` declares `static`: In order to maintain consistency with regular objects of the type `C`, static fields are not initialized with free variables. Instead, they share their static value with other objects of the same type. Consequently, if the free object is the first of its type to be initialized at runtime, the MLVM has to execute the static initializer that the type definition implements (`<clinit>()`, see [Lin+15, § 2.9]), thus preparing the shared fields for future free and regular objects of the same type.

Afterwards, the application code that has declared a free object `o` can refine it by interacting with it. For instance, it can invoke a method that pertains to `o` or use any of its fields in a constraint, thus specifying aspects of `o` as needed.

The alternative to treating the free object symbolically would be to generate specific object instances that fit the declaration `C o free;` at the time of instantiation. Afterwards, the MLVM could branch non-deterministically over the specific instances. However, this results in large or infinite sets of alternatives even for simple class structures [DK20b].

6.2.2 Field Access

Given a free object that is declared `Rectangle rect free;` and no additional constraints, its fields are accessible via `rect.width` or `rect.height`, respectively. Java and Muli do not offer runtime polymorphism for fields: Even if the actual type of `rect` is a subtype of `Rectangle` that provides its own field definition for `width`, the original field is not overridden [Gos+15, §§ 8.3 and 9.3]. Instead, such an instance would in fact store two fields of the same name; and accesses through the variable `rect` would result in accessing the original field because of the declaration as `Rectangle rect`.

This has two implications [DK20b]. First, a field access operation is always deterministic and depends only on the type of the variable through which the field is accessed, regardless of whether subtypes re-define the accessed field. Second, accessing the field of a freshly instantiated free object returns a free variable to be used as part of symbolic expressions. Depending on its type, that free variable can be a free object as well.

6.2.3 Method Invocation

The behaviour that objects encapsulate via methods may, in fact, change along the implementation hierarchy due to overriding [Dag19]. Since Muli inherits its support for runtime polymorphism from Java, invoking a method on a free object results in non-deterministic branching over the available implementations, based on the implementation hierarchy of the object's type. For instance, consider the example from Listing 6.1 that invokes `getArea()` on an unconstrained free object of the `Shape` type. Since there are four different implementations for `getArea()` (cf. Figure 6.1), that invocation has to result in a choice with four alternatives, one per implementation [DK20b].

Each of the generated choice alternatives is accompanied by an appropriate constraint that restricts the type of the free object accordingly [DK20b]. This ensures that the type assumption made for the purpose of invocation is enforced for future interactions with the same free object. The example from Listing 6.1 invokes a second method on the same free object, namely, `toString()`, which is implicitly available in the `Shape` interface per the definition in the JLS (see [Gos+15, § 4.3.2]). Again, there are four distinct

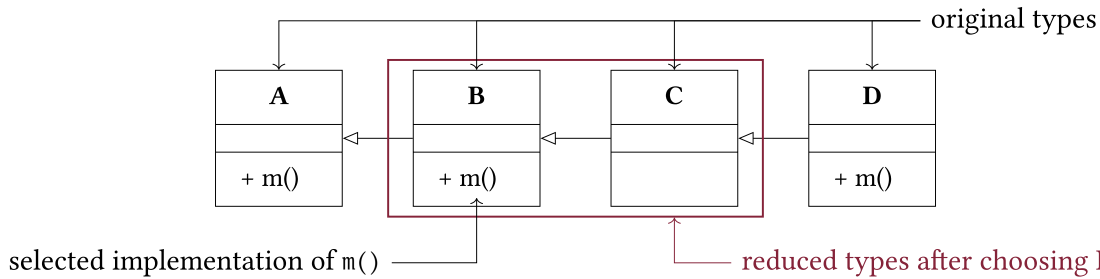
```
1  Set<Method> implementations(Object target, Method m) {
2    Set<Method> impls := {};
3    Method mostSpecificFromSupertypes := jvmsLookup(m, target.getClass());
4    if (mostSpecificFromSupertypes != null) {
5      impls += mostSpecificFromSupertypes; }
6    Set<Type> types := target.getPossibleTypes();
7    foreach (type ∈ types) {
8      Method implementation := type.getMethod(m);
9      if (implementation != null && !implementation.isAccAbstract()) {
10       if (type.isAccAbstract() || type.isAccInterface()) {
11         Set<Type> subtypes := type.getImmediateInstantiableSubtypes();
12         foreach (subtype ∈ subtypes) {
13           impls += subtype.getMethod(m); }
14       } else {
15         impls += implementation; } } }
16   return impls; }
```

Adapted from [DK20b]

Listing 6.2: Discovering the set of method implementations that are candidates for invocation.

implementation candidates for `toString()`. However, since branching over `getArea()` already makes assumptions over the free object's type, the implementation that is selected for `toString()` has to be consistent with that assumption. Therefore, in this example, the invocation of `toString()` is deterministic, because the invocation of `getArea()` imposes a constraint that specifies the free object's type uniquely (and, therefore, the selected implementation for `toString()`).

Listing 6.2 presents the pseudocode of a method `implementations()` that the MLVM uses to find implementation candidates when a method `m()` is invoked on a free object `target`. If the `implementations()` returns more than one element, invocation will result in a choice that branches over the returned implementations. First, the MLVM looks up the implementation that will be invoked if the free object assumes its supertype (`target.getClass()`). That is the most specific implementation upwards along the class hierarchy, and lookup is performed using the default deterministic JVMS lookup mechanism as specified for the *Invokeinterface* and *Invokevirtual* instructions [Lin+15, § 6.5]. This default mechanism is implemented in `jvmsLookup()`. Afterwards, implementations are obtained using `getMethod()` from every type that the free object may assume (`target.getPossibleTypes()`). These implementations are added to the result of `implementations()`, unless they belong to a type that is abstract. If the type that imple-



Adapted from [Dag19]

Figure 6.2: Types that a free object with the declaration `A a free` may assume, before and after choosing a method implementation for invocation.

ments a method is abstract, it cannot be instantiated. For such cases, the implementations from their immediate, instantiable subtypes are added; i. e. those from direct non-abstract subtypes and, for abstract subtypes, implementations from their immediate, instantiable subtypes. Finally, all found implementations are returned, which will result in the creation of an appropriate choice node that offers the found candidates as alternatives.

When the MLVM selects a branch from the choice, thus committing to an implementation and, therefore, to a type that provides the desired implementation, it imposes an appropriate constraint on the free object that restricts its possible types in accordance with the selected method implementation. However, this does not necessarily mean that the type of the free object is now known. Consider the artificial implementation hierarchy that is illustrated by Figure 6.2. Classes `A`, `B`, and `D` provide their own implementations of `m()`, thus exhibiting distinct behaviour. In contrast, `C` does not implement `m()` and therefore inherits the behaviour of `B`. Therefore, with a free object `A a free;`, invoking `a.m()` creates a choice with three branches, one per distinct implementation. While the unconstrained free object may assume either of the four types from Figure 6.2, every branch of the choice has a constraint restricting the type of `a`. After selecting a method implementation, `a` may assume the type that provides the selected implementation. Alternatively, it may assume any of its subtypes, unless a subtype provides its own implementation, because selecting a subtype with an implementation would require a contradictory constraint. For the example, this means that selecting the implementation provided by class `B` still allows the free object to assume the type of `B` or `C`, but not `D` because `D` overrides the implementation of `B`. The created choice, and constraints for all branches, are illustrated as a partial search tree in Figure 6.3. *types*(`a`) is the type constraint that specifies the exact allowed types for a free object `a`.

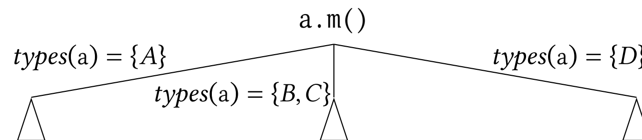


Figure 6.3: Partial search tree created from the invocation of `a.m()` in the example from Figure 6.2.

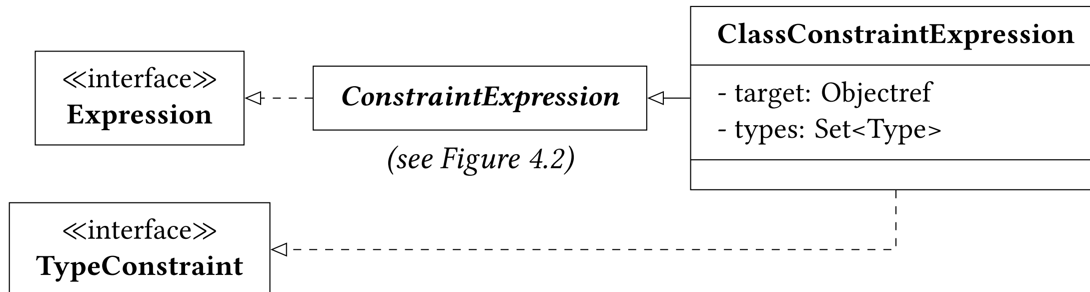


Figure 6.4: Representation of type constraints in the solver component of the MLVM.

Implementing the type constraint *types*() over free objects requires an extension of the solver component of the MLVM. After all, the solver libraries that the solver component uses (see Section 4.3) can solve constraints involving arithmetic and boolean expressions, but they have no notion of Muli and/or Java types. The MLVM represents type constraints in the class `ClassConstraintExpression` that has two fields, as illustrated in Figure 6.4. First, `target` holds the target `Objectref` that the constraint refers to. Second, the set `types` exhaustively describes all types that the target may assume, i. e., it may become any type that is explicitly specified in the set. The class `ClassConstraintExpression` extends from `ConstraintExpression` so that it can be added to the active constraint system by passing it to `addConstraint()` in the solver manager. Furthermore, it implements a new marker interface class `TypeConstraint`, symbolizing that the solver manager cannot transform and impose the constraint with the used solver library. Instead, a type constraint needs to be handled by the MLVM itself. Specifically, imposing a type constraint results in changing the allowed types that are specified in the target class `Objectref`, and removing the constraint reverts that change. Since type constraints are handled by the MLVM instead of the solver library, they work regardless of the selected solver.

6.2.4 Type Check and Type Cast

There are two bytecode operations that check the validity of type operations at runtime [Lin+15, § 6.5]. `Checkcast` is the bytecode equivalent of the Muli statement `(T)o` which attempts to cast an object `o` to the type `T`, whereas the bytecode instruction `Instanceof`

is the compiled result of the boolean expression `o instanceof T`, checking whether the object `o` is type-compatible with the type `T`, i. e., the type of `o` is `T` or a subtype of `T`. The two instructions differ in their behaviour, but not in their criterion. Both pop the topmost element `o` from the operand stack and check whether it is type-compatible with the intended type `T`. `Checkcast` throws an exception if that is not the case and does nothing otherwise, whereas `Instanceof` pushes the result of the check (i. e., `true` or `false`) to the operand stack.

For regular objects, these bytecode instructions are deterministic. In contrast, for free objects, evaluation of these operation results in the creation of a non-deterministic choice if the object's possible types allow either result for the type-compatibility check [DK20b]. However, they are evaluated deterministically if the type constraints are sufficiently specific. In order to determine whether the evaluation of `(T)o` or `o instanceof T` is deterministic or non-deterministic for a free object `Object o free;` with a type constraint *types*(o), two sets are created. The first set consists of those possible types in *types*(o) that would render the operation successful:

$$SuccessfulTypes_{o,\mathtt{T}} = \{t | t \in types(o), t \preceq \mathtt{T}\}$$

For the opposite case, a second set comprises the types that `o` may assume but that would result in an unsuccessful type-compatibility check.

$$AdverseTypes_{o,\mathtt{T}} = \{t | t \in types(o), t \npreceq \mathtt{T}\}$$

In these equations, $a \preceq b$ signifies that a is type-compatible with b. Since *types*(o) is never empty, at least one of the sets holds at least one type. If one of the sets is empty, execution is deterministic and its outcome depends solely on the instruction-specific behaviour. Otherwise, execution is non-deterministic and a choice node with two branches is created. The branch representing the successful operation has a constraint $types_1(o) = SuccessfulTypes_{o,\mathtt{T}}$, whereas the constraint of the other branch is $types_2(o) = AdverseTypes_{o,\mathtt{T}}$. An example for the resulting branch constraints is illustrated in Figure 6.5b.

6.2.5 Equality

Muli follows Java in the distinction of *reference equality* from *value equality* [Dag19; Gos+15, § 15.21.3]. Neither kind of equality needs special handling w. r. t. their (potentially

```
1 Shape s free;
2 Rectangle r = new Rectangle();
3 r.width = 100; r.height = 101;
4 return s.equals(r); // Variation 1,
5 // or...
6 return r.equals(s); // Variation 2.
```

[DK20b]

Listing 6.3: Excerpt from a search region that introduces non-determinism while checking for value equality in two variations.

```
1 public boolean equals(Object o) {
2   if (!(o instanceof Rectangle)) return false;
3   return this.width == ((Rectangle)o).width
4       && this.height == ((Rectangle)o).height; }
```

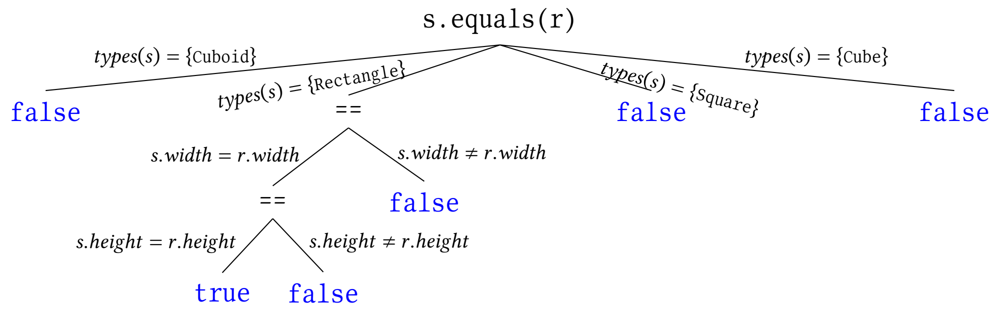
Listing 6.4: Implementation of `Rectangle.equals()`.

non-deterministic) evaluation, but the consequences of comparing equality between (potentially free) objects are interesting regardless.

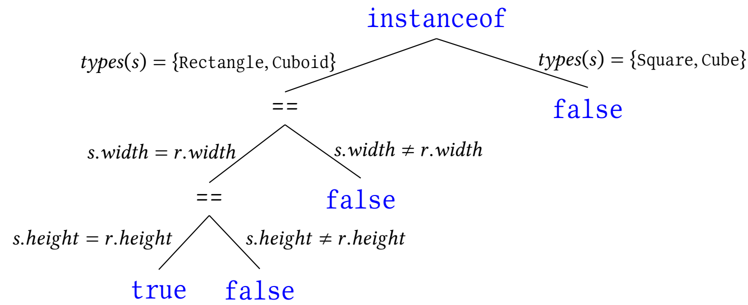
Reference equality is checked using the `==` or `!=` operators that resort to a comparison of the addresses of two objects on the heap, thus effectively checking whether two variables point to the same object. Here, the presence of free variables does not make a difference. Therefore, reference equality is always deterministic.

Value equality is slightly more complicated, because it is checked by invoking the `equals()` method on a (potentially free) object, passing another object for comparison as a parameter. As a consequence, value equality is not commutative. Moreover, the behaviour depends on the implementation of the invoked `equals()` method as well as on the object that the `equals()` method is invoked on, thus potentially adding non-determinism if that object is a free object. For example, Listing 6.3 instantiates a free and a regular object. The example invokes `equals()` to compare the two objects in two variations in order to demonstrate that the operation is not commutative, and to illustrate the different outcomes. For reference, the used implementation of `Rectangle.equals()` is provided in Listing 6.4. The other implementations of `equals()` in the class structure from Figure 6.1 are similar.

In the first variation, the `equals()` method is invoked on the free object, resulting in a choice with one per subtype that provides an implementation (see Figure 6.5a). The second variation invokes a specific method on a regular object, so the invocation in itself



(a) Variation 1: Invocation on the free object.



(b) Variation 2: Invocation on the regular object.

Adapted from [DK20b]

Figure 6.5: Search trees created from the invocation of `equals()` on free or regular objects.

is deterministic. However, the implementation of `Rectangle.equals()` first checks the type of the passed object using `instanceof`. As the type is not sufficiently constrained yet, `instanceof` computes two sets of types that would render the operation either successful or failing, respectively, as follows:

$$
\begin{aligned}
\mathit{SuccessfulTypes}_{s,\texttt{Rectangle}} &= \{\texttt{Rectangle}, \texttt{Cuboid}\} \\
\mathit{AdverseTypes}_{s,\texttt{Rectangle}} &= \{\texttt{Square}, \texttt{Cube}\}
\end{aligned}
$$

Since both sets contain types, a choice with two branches is created, using the sets of types for the branch constraints as illustrated in Figure 6.5b. Note that, once the types are sufficiently constrained, the remaining subtrees of the two search trees look identical, adding constraints over the two fields of `Rectangle`.

As a final note, keep in mind that value equality depends on the specific implementations of `equals()` that are invoked. In the presented example, `equals()` compares types and field values for equality. Other classes can use other criteria that need not necessarily

result in non-deterministic execution, even if a free object is passed as a parameter. For example, consider an implementation for `equals()` that always returns `true`.

6.3 Summary

As a result of the considerations in this chapter, Muli offers logic variables for objects. This gives application developers the ability to perform the same interactions with these logic variables as those that are allowed for regular objects. With respect to non-deterministic execution, specific measures are necessary for handling invocation and type operations on free objects. Indirectly, these measures have potential effects on checks for value equality. Moreover, free objects require special initialization. However, reference equality and field accesses behave exactly the same for free objects as they do for regular objects.

The implementation of free objects in the MLVM requires a new kind of constraint that restricts the types that a free object may use at runtime. To that end, the solver manager component has been extended, managing type constraints in addition to the previously existing constraints.

For the moment, these considerations only apply to logic variables with class types or interface types. Future work can consider the introduction of free arrays, i. e., array-type logic variables.

1. Chapter 11: Jan C. Dageförde and Herbert Kuchen. 'Free Objects in Constraint-logic Object-oriented Programming'. In: *Proceedings of the ACM on Programming Languages (OOPSLA)*. 2020. Under review

2. Chapter 14: Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming*. Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8

7 Conclusion

With this chapter, Part I of this thesis is concluded. Section 7.1 presents and summarizes the contributions of this work. Afterwards, Section 7.2 notes current limitations. Last but not least, Section 7.3 provides a perspective regarding future research opportunities.

7.1 Contributions

The research presented in this work has set out to improve the state of the development of object-oriented software that involves constraint-logic search. To that end, the objective of this research has been to design and to develop a programming language that provides integrated support for object-oriented programming and constraint-logic programming. The main output of this research is the CLOOP language Muli that, based on Java, provides features from object-oriented programming as well as from constraint-logic proramming. Therefore, the objective has been achieved.

With the support of the Muli compiler and the MLVM runtime environment, Muli allows developers to integrate non-deterministic search with other, deterministic business logic in a single Muli application. To that end, constraint-logic features are first-class citizens in a programming language that primarily uses an object-oriented syntax. Free variables (i. e., logic variables) in Muli can be of primitive types as well as of class or interface types. Since the MLVM supports symbolic execution, free variables can be used in the same contexts as regular variables of compatible types, regardless of whether the variables are numeric, boolean, or objects. Therefore, Muli's seamless integration of non-deterministic search and object-oriented programming facilitates the development of applications that interleave constraint definition and search with imperative statements.

The MLVM structures the non-deterministic execution of Muli search regions in a search tree and makes this structure explicit. For the development of the MLVM as well as during the development of Muli applications, this explicit structure is helpful for explaining non-deterministic search in Muli applications. The search tree also serves as a basis for the implementation of strategies how the MLVM explores the search space

of a search region. This allows Muli application developers to select a strategy that is suitable for a given search problem.

All things considered, using Muli for the development of business applications that require search relieves developers from having to separate object-oriented application parts from search-related parts of the application. Moreover, using Muli avoids deficiencies that would arise from manual integration with Prolog or other external tools for search, and does not result in vendor lock-in regarding the used constraint solver. Since Muli is syntactically and semantically based on Java 8 and maintains backwards compatibility with Java, developers with experience in Java can easily learn Muli and start using it for development.

The research has culminated in several contributions for practice as well as for theory.

Contributions for practice As a result of using Java as a base language, Muli is a programming language whose syntax is mainly an object-oriented one. This could be helpful in an attempt to establish Muli among mainstream programming languages. At the same time, the constraint-logic features that Muli offers are helpful for applications that rely on search as part of their business logic. Even though the MLVM is a research prototype that may not cater to every practical need, and that has room for improvement in future work, its stability is assured using a set of test cases that are executed on every change to the source code. Moreover, the source code of all tools that have been developed as part of this research and that are presented in this dissertation are publicly available,[7] thus facilitating the adoption of the programming language.

Contributions for theory Furthermore, the results of this research add to the scientific body of knowledge. The concept of constraint-logic object-oriented programming is novel and no such programming language existed prior to this research. Moreover, the idea of executing object-oriented programs non-deterministically for purposes other than test-case generation is novel. The Münster Logic-Imperative Language fills these research gaps, presenting a programming language that integrates the object-oriented and constraint-logic paradigms. Furthermore, the concept of dual trails is a novel modification of the WAM trail. The dual trails facilitate the restoration of arbitrary execution states in an environment in which side effects from execution play a role, so that dual trails are a prerequisite for stopping and resuming search in a stateful environment. The trails also facilitate the implementation of

[7] `https://github.com/wwu-pi/muli/`.

breadth-first search as a search strategy that is novel in the context of (stateful) execution of imperative applications. Last but not least, this dissertation and its related publications define language concepts for Muli that can also serve as a blueprint of future CLOOP languages. This encompasses future languages based on Java as well as ones based on other object-oriented programming languages.

7.2 Limitations

Even though this research has been able to achieve its objective, it has its limitations as certain aspects have been disregarded so far. These aspects do not depreciate the accomplishments, but should nevertheless be duly noted.

This research proposes CLOOP as a novel paradigm, however, it only focusses on a single language by developing Muli. Therefore, future research needs to show whether the novel concepts can contribute to the development of further CLOOP languages. Moreover, Muli is based on Java 8. As a result, the development has not considered more recent Java features (yet). Major recent features that might also be useful for Muli applications include the Java Platform Module System [MB17], type inference for local variables [Goe18], and switch expressions [Bie20]. Basing Muli on top of a more recent Java version would also improve the compatibility of Muli applications with existing Java libraries.

As a custom implementation of a JVM, the MLVM does not benefit from advanced features that a standard JVM offers. In particular, the MLVM does not provide just-in-time compilation of the bytecode, so that the bytecode of Muli classes has to be interpreted. Nevertheless, at least the MLVM can be executed on a standard JVM, so that the JVM's just-in-time compilation can optimize the execution of MLVM.

Finally, it should be noted that Muli supports single-threaded applications only. Adding support for multi-threading in the context of non-deterministic search requires sophisticated techniques that reduce the search space.

7.3 Perspectives for Future Research

The progress of this work has by far not exhausted the opportunities for research, neither on CLOOP in general nor on Muli in particular. A non-exhaustive set of ideas for future work is presented subsequently.

So far, Muli is restricted to single-threaded applications, thus breaking a bit of backward compatibility with existing Java applications and libraries. Future research could investigate possible support for multi-threaded Muli applications. Here, differentiations could be made regarding the mode of execution; that is, support for non-deterministic execution of search regions with multi-threading will require different considerations than multi-threaded execution of deterministic parts of an application outside search regions.

Multi-threading provides another opportunity for research. Currently, non-deterministic decisions at choices are evaluated in sequence. However, given sufficient computing resources, it is theoretically possible to evaluate multiple decision alternatives in parallel. To that end, it is particularly important that the execution of decision alternatives is isolated so that they strictly cannot interfere with the execution state of the other alternatives that are executed in parallel.

Another aspect of the MLVM that lends itself for future research is the use of trails for achieving specific execution states. Alternatively, a programming language with immutability guarantees and copy-on-write data structures could be more efficient in representing specific execution states by creating incremental snapshots at choices. This hypothesis can be evaluated experimentally by implementing an alternative to the MLVM in a programming language that provides such guarantees.

The considerations regarding reference-type logic variables are focused on free objects. Future research can investigate how logic variables with an array type should be treated in CLOOP. Last but not least, all the presented concepts for CLOOP can be transferred to another programming language, particularly to one that is based on a different object-oriented language.

References

[Apt+98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington and Andrea Schaerf. *Alma-0: An Imperative Language that Supports Declarative Programming*. 1998. DOI: `10.1145/293677.293679`.

[Apt09] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2009. ISBN: 978-0-521-82583-2.

[AS99] Krzysztof R. Apt and Andrea Schaerf. 'The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming'. In: *Correct System Design*. Ed. by E.-R. Olderog and B. Steffen. Vol. 1710. LNCS. Springer, 1999, pp. 89–113. DOI: `10.1007/3-540-48092-7_5`.

[Bar+09] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia and Cesare Tinelli. 'Satisfiability modulo theories'. In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 825–885. DOI: `10.3233/978-1-58603-929-5-825`.

[BHH04] Bernd Braßel, Michael Hanus and Frank Huch. 'Encapsulating Non-Determinism in Functional Logic Computations'. In: *Journal of Functional and Logic Programming* 2004.28 (2004).

[Bie20] Gavin Bierman. *JEP 361: Switch Expressions (Standard)*. 2020. URL: `https://openjdk.java.net/jeps/361` (visited on 01/05/2020).

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller and Fabian Reck. 'KiCS2: A New Compiler from Curry to Haskell'. In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: `10.1007/978-3-642-22531-4`.

[BS03] Mike Barnett and Wolfram Schulte. 'Runtime verification of .NET contracts'. In: *Journal of Systems and Software* 65.3 (2003), pp. 199–208. DOI: `10.1016/S0164-1212(02)00041-9`.

[BSA83] Andrew G. Barto, Richard S. Sutton and Charles W. Anderson. 'Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems'. In: *IEEE Transactions on Systems, Man and Cybernetics* SMC-13.5 (1983), pp. 834–846. ISSN: 21682909. DOI: 10.1109/TSMC.1983.6313077.

[Cad+11] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann and Willem Visser. 'Symbolic execution for software testing in practice: preliminary assessment'. In: *2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 1066–1071. DOI: 10.1145/1985793.1985995.

[CDE08] Cristian Cadar, Daniel Dunbar and Dawson Engler. 'Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs'. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008* (2008), pp. 209–224.

[Ceb20] Martine Ceberio. *Solvers.* 2020. URL: http://www.constraintsolving.com/solvers (visited on 30/04/2020).

[CM03] Christopher S. Clocksin and William F. Mellish. *Programming in Prolog: using the ISO standard.* 5th ed. Berlin u.a.: Springer, 2003. ISBN: 0-387-00678-8.

[Cuk17] Ivan Cukic. *Functional Programming in C++.* New York: Manning Publications Co., 2017. ISBN: 978-1-61729-381-8.

[Dag19] Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming.* Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8.

[DB08] Leonardo De Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* TACAS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 978-3-540-78799-0.

[DD06] Bruno Dutertre and Leonardo De Moura. 'A fast linear-arithmetic solver for DPLL(T)'. In: *Computer Aided Verification (CAV) 2006.* Ed. by T. Ball and R. B. Jones. Vol. 4144. LNCS. Berlin, Heidelberg: Springer, 2006, pp. 81–94. DOI: 10.1007/11817963_11.

[DK18a] Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*. ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260.

[DK18b] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DK19a] Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

[DK19b] Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing*. Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912.

[DK20a] Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems*. Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review.

[DK20b] Jan C. Dageförde and Herbert Kuchen. 'Free Objects in Constraint-logic Object-oriented Programming'. In: *Proceedings of the ACM on Programming Languages (OOPSLA)*. 2020. Under review.

[DM03] J. Doyle and C. Meudec. 'IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming'. In: *2nd PPPJ*. 2003, pp. 55–58. DOI: 10.1145/957289.957307.

[Doe94] Kees Doets. *From logic to logic programming*. Cambridge, MA: MIT Press, 1994. ISBN: 978-0-26204-142-3.

[DT20] Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13.

[EH07] Torbjörn Ekman and Görel Hedin. 'The JastAdd extensible Java compiler'. In: *OOPSLA*. 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297105.1297029.

[EMK12] Marko Ernsting, Tim A. Majchrzak and Herbert Kuchen. 'Test Case Generation and Dynamic Mixed-Integer Linear Arithmetic Constraint Solving'. In: *21st WFLP*. 2012.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Berlin Heidelberg: Springer, 2003. ISBN: 978-3-642-08712-7.

[Fel12] Jacob Feldman. *JSR 331: Constraint Programming API*. 2012. URL: https://jcp.org/en/jsr/detail?id=331 (visited on 01/05/2020).

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1994. ISBN: 978-81-317-0007-5.

[GH13] Shirley Gregor and Alan R Hevner. 'Positioning and Presenting Design Science Research for Maximum Impact'. In: *MIS Quarterly* 37.2 (2013), pp. 337–355. ISSN: 02767783. DOI: 10.2753/MIS0742-1222240302.

[God+08] P. Godefroid, P. De Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N Tillmann and M Y Levin. 'Automating Software Testing Using Program Analysis'. In: *IEEE Software* 25.5 (2008), pp. 30–37. DOI: 10.1109/MS.2008.109.

[Goe18] Brian Goetz. *JEP 286: Local-Variable Type Inference*. 2018. URL: https://openjdk.java.net/jeps/286 (visited on 01/05/2020).

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 05/05/2020).

[Gre02] Shirley Gregor. 'A Theory of Theories in Information Systems'. In: *Information Systems Foundations: Building the Theoretical Base*. Ed. by S. Gregor and D. Hart. Canberra: Australian National University, 2002, pp. 1–20.

[Han+19] M. Hanus et al. *PAKCS 2.2.0: The Portland Aachen Kiel Curry System*. 2019. URL: https://www.informatik.uni-kiel.de/~pakcs/Manual.pdf (visited on 24/04/2020).

[Han97] Michael Hanus. 'A Unified Computation Model for Functional and Logic Programming'. In: *POPL 97*. 1997, pp. 80–93.

[Hev+04] Alan R. Hevner, Salvatore T. March, Jinsoo Park and Sudha Ram. 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[HKM95] Michael Hanus, Herbert Kuchen and Juan Jose Moreno-Navarro. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[HPR16] Michael Hanus, Bjoern Peemoeller and Fabian Reck. *Module SearchTree*. 2016. URL: `https://www-ps.informatik.uni-kiel.de/kics2/lib/SearchTree.html` (visited on 28/04/2020).

[Hun18] John Hunt. *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. 2nd ed. Springer, 2018. ISBN: 978-3-319-75770-4.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. ISBN: 0-521 826144.

[Kie+09] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer and Michael D Ernst. 'HAMPI: a solver for string constraints'. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. 2009, pp. 105–116. DOI: `10.1145/1572272.1572286`.

[Kin76] James C. King. 'Symbolic execution and program testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: `10.1145/360248.360252`.

[KJ00] Adil Kabbaj and Martin Janta-Polczynski. 'From PROLOG++ to PROLOG+CG: A CG Object-Oriented Logic Programming Language'. In: *Proceedings of ICCS 2000*. Vol. 1867. LNAI. Berlin, Heidelberg: Springer, 2000, pp. 540–554. DOI: `10.1007/10722280_37`.

[KO08] Goh Kondoh and Tamiya Onodera. 'Finding bugs in Java Native Interface programs'. In: *ISSTA '08* (2008), p. 109. DOI: `10.1145/1390630.1390645`.

[Kor85] Richard E. Korf. 'Depth-first Iterative Deepening: An Optimal Admissible Tree Search'. In: *Artificial Intelligence* 27 (1985), pp. 97–109.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu and Willem Visser. 'Generalized Symbolic Execution for Model Checking and Testing'. In: *TACAS'03 Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems*. 2003, pp. 553–568.

[Kri+20] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau and Dierk Ehmke. 'Towards Constraint Logic Programming over Strings for Test Data Generation'. In: *Declarative Programming and Knowledge Management.* Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 139–159. DOI: 10.1007/978-3-030-46714-2_10.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 1084-4309. DOI: 10.1145/785411.785416.

[Lem+04] Christoph Lembeck, Rafael Caballero, Roger A. Müller and Herbert Kuchen. 'Constraint Solving for Generating Glass-Box Test Cases'. In: *Proceedings WFLP '04.* 2004, pp. 19–32.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition.* 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 05/05/2020).

[LK99] Wolfgang Lux and Herbert Kuchen. 'An Efficient Abstract Machine for Curry'. In: *Informatik '99.* Ed. by K Beiersdörfer, G Engels and W Schäfer. Springer Verlag, 1999, pp. 390–399.

[Lou93] Kenneth C. Louden. *Programming Languages: Principles and Practice.* Ed. by Patricia Adams. Belmont, CA, USA: Wadsworth Publ. Co., 1993. ISBN: 0534932770.

[Lux99] Wolfgang Lux. 'Implementing Encapsulated Search for a Lazy Functional Logic Language'. In: *Functional and Logic Programming, 4th International Symposium, FLOPS '99, Tsukuba, Japan, November 1999, Proceedings.* Ed. by Aart Middeldorp and Taisuke Sato. LNCS 1722. Springer Verlag, 1999, pp. 100–113.

[LWS13] Wing Hang Li, David R. White and Jeremy Singer. 'JVM-hosted languages: They talk the talk, but do they walk the walk?' In: *PPPJ '13: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools.* 2013, pp. 101–112. DOI: 10.1145/2500828.2500838.

[MB17] S. Mak and P. Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* O'Reilly Media, 2017. ISBN: 978-1-491-95413-3.

[MBB06] Erik Meijer, Brian Beckman and Gavin Bierman. 'LINQ: Reconciling Objects, Relations and XML in the .NET Framework'. In: *ACM SIGMOD International Conference on Management of data*. 2006, p. 706. DOI: 10.1145/1142473.1142552.

[Meu01] Christophe Meudec. 'ATGen: Automatic test data generation using constraint logic programming and symbolic execution'. In: *Software Testing Verification and Reliability* 11.2 (2001), pp. 81–96. DOI: 10.1002/stvr.225.

[Mic20] Microsoft. *Reference types (C# Reference)*. 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types (visited on 16/04/2020).

[MK09] Tim A. Majchrzak and Herbert Kuchen. 'Automated Test Case Generation Based on Coverage Analysis'. In: *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 2009, pp. 259–266. ISBN: 978-0-7695-3757-3. DOI: 10.1109/TASE.2009.33.

[MK11a] Tim A. Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7. DOI: 10.1007/978-3-642-22531-4_8.

[MK11b] Tim A. Majchrzak and Herbert Kuchen. 'Muggl: The Muenster Generator of Glass-box Test Cases'. In: *Working Papers, European Research Center for Information Systems* 10 (2011).

[MLK04] Roger A. Müller, Christoph Lembeck and Herbert Kuchen. 'A Symbolic Java Virtual Machine for Test Case Generation'. In: *Proceedings IASTED SE 04*. 2004.

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. Cambridge, MA: MIT Press, 1998. ISBN: 978-0-262-13341-8.

[Mvn20] Mvnrepository.com. *Published Jars per Year on Mavencentral*. 2020. URL: https://mvnrepository.com/repos/central (visited on 25/04/2020).

[Ora20a] Oracle. *Java Platform SE 8 API Overview*. 2020. URL: https://docs.oracle.com/javase/8/docs/api/ (visited on 20/04/2020).

[Ora20b] Oracle. *java.util.function.Supplier<T>*. 2020. URL: https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html (visited on 28/03/2020).

[Ora20c] Oracle. *java.util.stream.Stream<T>*. 2020. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html (visited on 29/03/2020).

[Öst+10] Hubert Österle et al. 'Memorandum zur gestaltungsorientierten Wirtschaftsinformatik'. In: *Schmalenbachs Zeitschrift für betriebswirtschaftliche Forschung* 62.6 (2010), pp. 664–672. ISSN: 0341-2687. DOI: 10.1007/bf03372838.

[Ost15] Ludwig Ostermayer. 'Seamless Cooperation of Java and Prolog for Rule-Based Software Development'. In: *Proceedings of the RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, the Special Industry Track and the RuleML 2015 Doctoral Consortium hosted by the 9th International Web Rule Symposium (RuleML 2015), Berlin,* ed. by Nick Bassiliades et al. Vol. 1417. CEUR Workshop Proceedings. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1417/paper2.pdf.

[Pas+19] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. D'Alché-Buc, E. Fox and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[Pef+07] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger and Samir Chatterjee. 'A Design Science Research Methodology for Information Systems Research'. In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages and Xavier Lorca. *Choco Documentation.* TASC, INRIA Rennes. 2016. URL: http://www.choco-solver.org.

[PS09] Tomas Petricek and Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. 1st ed. Manning Publications Co., 2009. ISBN: 978-1-933-98892-4.

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010. ISBN: 978-1-43274-936-1.

[SMB95] Mark S. Silver, M. Lynne Markus and Cynthia Mathis Beath. 'The Information Technology Interaction Model: A Foundation for the MBA Core Course'. In: *MIS Quarterly* 19.3 (1995), pp. 361–390. ISSN: 0276-7783. DOI: 10.2307/249600.

[Sta17] Stack Overflow. *Developer Survey Results 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#technology (visited on 03/05/2020).

[Sta18] Stack Overflow. *Developer Survey Results 2018*. 2018. URL: https://insights.stackoverflow.com/survey/2018#technology (visited on 03/05/2020).

[Sta19] Stack Overflow. *Developer Survey Results 2019*. 2019. URL: https://insights.stackoverflow.com/survey/2019#technology (visited on 03/05/2020).

[TH08] Nikolai Tillmann and Jonathan de Halleux. 'Pex: White Box Test Generation for .NET'. In: *2nd International Conference on Tests and Proofs*. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9.

[The17] The OptaPlanner Team. *OptaPlanner User Guide, Version 7.0.0*. JBoss. 2017. URL: https://docs.optaplanner.org/7.0.0.Final/optaplanner-docs/html_single/index.html.

[TIO20] TIOBE Software. *TIOBE Index for February 2020*. 2020. URL: https://www.tiobe.com/tiobe-index/ (visited on 25/02/2020).

[Tri12] Markus Triska. 'The Finite Domain Constraint Solver of SWI-Prolog'. In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316. DOI: 10.1007/978-3-642-29822-6_24.

[Tri18] Markus Triska. 'Boolean constraints in SWI-Prolog: A comprehensive system description'. In: *Science of Computer Programming* 164 (2018), pp. 98–115. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2018.02.001.

[TS06] N. Tillmann and W. Schulte. 'Unit tests reloaded: Parameterized unit testing with symbolic execution'. In: *IEEE Software* 23.4 (2006), pp. 38–47. ISSN: 0740-7459. DOI: 10.1109/MS.2006.117.

[UFM14] Raoul-Gabriel Urma, Mario Fusco and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT: Manning Publications Co., 2014. ISBN: 978-1-617-29199-9.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. DOI: 10.1017/S1471068403001741.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set*. Tech. rep. Menlo Park: SRI International, 1983.

[Wie03] Jan Wielemaker. 'An Overview of the SWI-Prolog Programming Environment'. In: *Workshop on LP Environments*. 2003.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Berlin, Heidelberg: Springer, 1985. ISBN: 978-3-642-96880-8.

Part II

Included Publications

8 Publication Overview

Part II reproduces the eight publications pertaining to this cumulative dissertation. The individual contributions of the publications have been presented and put into perspective in Part I.

The reproductions are identical in content compared to the original publications. However, their formatting has been adapted to match the formatting of this thesis, for the purpose of improving readability as well as providing an appearance that is consistent with that of Part I.

This overview presents title, authors, and outlets of each publication. Furthermore, it indicates how the respective outlets are ranked according to the CORE conference and journal rankings.[8] For publications that are still under review at the time of writing, the ranking is indicated in parentheses.

Journal publication

No.	Publication	CORE
J1	Chapter 9 [DK19a]: Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001	C[9]

[8] http://portal.core.edu.au/conf-ranks/ and http://portal.core.edu.au/jnl-ranks/, respectively.

[9] Based on the ranking of the journal's direct predecessor until 2018: "Computer Languages, Systems and Structures (COMLAN)".

Book chapter

No.	Publication	CORE
B1	Chapter 10 [DK20a]: Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems.* Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review	(-)

Conference (post-)proceedings

No.	Publication	CORE
C1	Chapter 11 [DK20b]: Jan C. Dageförde and Herbert Kuchen. 'Free Objects in Constraint-logic Object-oriented Programming'. In: *Proceedings of the ACM on Programming Languages (OOPSLA).* 2020. Under review	(A*)
C2	Chapter 12 [DT20]: Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13	C
C3	Chapter 13 [DK19b]: Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing.* Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912	B
C4	Chapter 14 [Dag19]: Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming.* Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8	C
C5	Chapter 15 [DK18a]: Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing.* ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260	B
C6	Chapter 16 [DK18b]: Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5	C

9

A Compiler and Virtual Machine for Constraint-Logic Object-Oriented Programming

Jan C. Dageförde* · **Herbert Kuchen***

Citation Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

Citation

Abstract The development of enterprise software typically relies on object-oriented (OO) programming languages. However, OO languages are not particularly suited for the implementation of software which involves solving complicated search problems with dynamically appearing constraints, e. g. as found in logistics. Aiming to improve this situation, we propose the Münster Logic-imperative Language (Muli). As a constraint-logic OO language, it facilitates an integrated implementation of applications that use aspects from both constraint-logic and object-oriented programming, thus eliminating the need to resort to JNI for the integration of search applications that are written in a (constraint) logic language. Muli extends Java by logic variables and encapsulated search. Its runtime is based on a symbolic Java virtual machine and leverages constraint solvers. Outside of search regions, Muli behaves just like Java.

We motivate the benefits of integrating object-oriented programming and constraint-logic programming into a single language and introduce novel concepts that are required for a seamless integration. Furthermore, we present an operational semantics and transfer concepts and semantics into implementations of a compiler and a virtual machine.

Keywords Programming paradigm integration · programming language · constraint-logic programming · operational semantics · virtual machine implementation.

*University of Münster, Germany

9.1 Motivation

In contemporary software development, object-oriented (OO) programming is prevalent, as languages such as Java and C# continue to dominate usage rankings [Sta18; Sta19; TIO19]. Inheritance and encapsulation of behaviour and structure are examples of features that contribute to reusability as well as maintainability, thus making them useful for most industry applications [Lou93]. However, there are scenarios in which languages from other paradigms are more suitable.

Consider search problems. Constraint-logic programming languages, such as Prolog with the CLP(FD) package, allow developers to declaratively specify a search space in terms of variables and their constraints [Tri12]. As a result, the search for solutions is performed implicitly by the runtime environment and the included constraint solver. In contrast, solving search problems in Java requires either a manual implementation of an imperative solver or relies on importing third-party constraint solver libraries. Self-made implementations of solvers are often highly specialised towards a given problem, which might be beneficial for performance but harms generalisability. In contrast, several third-party constraint solvers are efficient for a variety of problems, but do not provide a standardised application programming interface (API), which results in a lock-in effect. Another option, using (e. g.) Prolog via the Java Native Interface (JNI), is tedious and error-prone due to the nature of the JNI [KO08].

In an effort to remedy this situation, we propose a novel approach to integrating constraint-logic and OO paradigms based on Java. The *Münster Logic-Imperative Language (Muli)*, a constraint-logic object-oriented programming language. Instead of developing yet another constraint solver library, of which there are many (cf. e. g. [PFL17; The18; Kuc03]), our solution provides means for constraint-logic programming within Java programs by adding the concept of free variables directly to the language, i. e. variables that are not initialised to a particular value but to a symbolic value of a certain Java type. This is combined with symbolic execution by a specialised Java virtual machine (JVM) that adapts concepts from the Warren Abstract Machine [War83]. Within code parts that we refer to as *search regions*, execution becomes non-deterministic whenever branching conditions involve one or more free variables whose domains are insufficiently constrained so that more than one branch is applicable. By backtracking, the JVM ensures execution of all applicable branches whose conditions satisfy all imposed constraints.

Muli is particularly suited for enterprise applications of which most of the business logic can be expressed adequately in Java, but which occasionally require solving search problems whose details have been assembled in previous inputs or calculations. An

example of such an application is a truck scheduling system which dynamically adapts its schedule depending on traffic information and newly arriving orders, thus incrementally adding constraints.

Building on prior work presented at SAC 2018 [DK18a], this paper introduces Muli as a constraint-logic OO language and motivates concepts for language and runtime that are required to achieve this integrated paradigm, including an operational semantics. To these ends, our paper is structured as follows. We start off by describing novel language concepts and the resulting syntactic extension in Section 9.2, with a corresponding compiler detailed in Section 9.3. Furthermore, an operational semantics for a core subset of Muli is described in Section 9.4. Section 9.5 presents a custom implementation of a runtime environment, detailing structures and runtime concepts required for the execution of Muli applications. Using sample applications, we discuss the advantages and weaknesses of this approach in Section 9.6. Related work is summarised in Section 9.7. In Section 9.8, we then draw a short conclusion and sketch future work.

9.2 Muli Language

Muli is a language extension to Java, with Java 8 as the reference language. Additions to the language are kept to a minimum, and we entirely refrain from making modifications to existing Java concepts and features in order to minimise the burden on Java developers to understand Muli programs. As a result, Muli is a superset of Java, so that every Java program is also a Muli program that can be compiled and executed by Muli.

Our approach follows some design principles that we deem useful. First, we want to solve search problems supported by a custom-tailored SJVM. Second, we want search to be encapsulated. As a result, non-deterministic execution is only performed if explicitly required, whereas other parts of the program remain deterministic and cause no overhead w. r. t. Java. Third, we refrain from adding more special syntax than absolutely necessary, especially for defining constraints. For example, we do not want to add operators for constraints that can be expressed using relational Java operators. Fourth, since Java programs are not executed lazily, Muli should not be evaluated lazily either, in contrast to integrations of logic programming with other paradigms (cf. e. g. Curry [Han97]). Last but not least, Muli should be considered an extension of Java, as opposed to an entirely new language. This implies that functionality (and therefore understanding!) of Java constructs remains unchanged and performance of deterministic program parts should not be adversely affected.

We decided to use Java as the reference language as it is a ubiquitous programming language which is well-known and well-understood among most developers. Moreover, it comprises advantageous features of OO imperative languages, such as platform independence, inheritance, and encapsulation of data and operations [Lou93]. In contrast to programs written in C++, which are compiled and then executed directly on the machine, Java programs compile to intermediate bytecode and are then executed on a (software) virtual machine. As a result, it is relatively easy to implement a custom runtime to evaluate modifications to the language. Even though no official formal operational semantics exists, Java and its corresponding runtime are documented extensively in natural language ([Gos+15] and [Lin+15], respectively), which facilitates both conceiving an extension and deriving implementations.

9.2.1 Language Concepts

Extending Java to become a constraint-logic OO language requires a few concepts that are novel to Java. First, we need to add the concept of *logic variables.* Actually, given the SJVM, any Java variable can be considered a logic variable. However, regular Java enforces that every variable must be initialised to a particular value before it is used. In contrast to that, Muli introduces *free variables* using the `free` keyword, indicating that they are initialised, although not bound to a particular value. In principle, a logic variable can be of any type. However, full support is limited to logic variables of primitive types for now, whereas an extension towards full support of reference types is going to be part of future work.

Second, we add *encapsulated search*, adapting the identically named concept from the functional constraint-logic language Curry [Han+95], to provide an abstraction from non-deterministic execution. Within encapsulated search, non-deterministic execution can happen, whereas any program part outside encapsulation is deterministic, just as regular Java programs are. We refer to a program part inside encapsulation as *search region.* An encapsulated search region is executed symbolically.

Whenever conditional branching occurs that involves insufficiently constrained (logic) variables, non-determinism is introduced. As a consequence, *constraints* are imposed incrementally once a valid branch is chosen. Its branching condition is imposed as an additional constraint in conjunction with the ones that existed previously. Afterwards, symbolic execution continues. Later, execution of the branch is backtracked, the former constraint is removed, and the next branch is chosen analogously. By deriving constraints from the branching condition, we avoid having to introduce additional keywords,

symbols, or classes for the definition of constraint. This gives Muli a representational advantage over constraint solver libraries that require instantiating their proprietary object representations in order to define (and impose) constraints. More insight on this advantage is provided in Section 9.6.

A search region is specified as a method that takes no parameters as input and that returns a value. Such a method definition is consistent with the `Supplier` functional interface (cf. [Gos+15]), i. e., it can be specified as an appropriate lambda expression or by explicitly implementing `Supplier`. Effectively, the result can be multiple return values due to non-determinism. The values returned by a search region are considered *solutions* that the encapsulation collects and returns to its caller. Furthermore, we enable developers to explicitly cut execution branches, resulting in immediate backtracking without adding a solution.

Generally, we consider runtime exceptions that occur during execution of a search region as a kind of solution, as they are just another result of the execution. Although they do not represent a particular value, they may be of interest to the surrounding application. To facilitate control over this behaviour, we propose operators that configure encapsulated search and the expected kinds of solutions. The most general case is that all solutions of a search region, including exceptions, are to be returned (`getAllSolutionsEx`). This general case can be modified to return the first solution (`getOneSolutionEx`), to discard exceptions (`getAllSolutions`), or in combination to return the first non-exception solution (`getOneSolution`).

As an introductory example, Listing 9.1 presents a simple Muli application that makes use of the constraint-logic OO programming style. The example application searches and prints the square root of a fixed number. We express this by the constraint $x == y/x$ (i. e. $x == \lfloor\sqrt{y}\rfloor$) over integer variables x and y, resorting to a constraint solver for finding x. As in Java, a method with the signature **`public static void`** `main(String[])` is used by the SJVM as the entry point. Computation remains deterministic, i. e. non-searching, until encapsulation begins. Assuming that we are interested in only one solution that should not be an exception, we use the `getOneSolution` operation that returns a single non-exception value. We use this operation in `main()` in order to create an encapsulated search region that calls the method `sqrt()`. For elegance, the search region is expressed as a lambda abstraction. Alternatively, a method reference could be used, thus facilitating reuse of search regions across an application. By using a lambda abstraction rather than an expression in the argument of the encapsulation operators, we make sure that the search region is not immediately evaluated, but only under control of the encapsulated search mechanism.

```
1 public static void main(String[] args) {
2   int i = Muli.getOneSolution(() -> sqrt(5));
3   System.out.println(i); }
4 public static int sqrt(int y) {
5   int x free;
6   if (x == y/x) return x;
7   else throw Muli.fail(); } // Not defined.
```

Listing 9.1: Muli program that searches the (integer) square root of 5 and prints the result (class header omitted).

```
1 public static void main(String[] args) {
2   Stream<Solution<Integer>> factorials = Muli.getAllSolutionsEx(() -> {
3     int n free;
4     return fact(n); } );
5   factorials.limit(100).forEach(i -> System.out.println(i)); }
6 private static int fact(int n) {
7   if (n == 0) return 1;
8   else if (n >= 1) return n * fact(n - 1);
9   else throw Muli.fail(); } // Not defined.
```

Listing 9.2: Muli program that searches factorials using non-deterministic evaluation and prints the first 100 of them (class header omitted).

In `sqrt()`, `x` is declared as a free variable which might later be bound to a value. The branching condition of the `if` statement cannot be evaluated to a single boolean value, as `x` is unconstrained. Therefore, the constraint `x == x/y` is added to the constraint store and the computation continues with the first branch of the `if` statement, namely `return x`. Since the `return` statement finishes the considered execution branch, the constraint solver searches and finds a solution satisfying the accumulated constraints (here consisting of a single constraint) and returns the obtained value for `x`, namely 2, as a result. Solutions that do not satisfy the above constraint are cut off by the `fail()` operator.

An example of finding multiple solutions is printed in Listing 9.2. In this case, `getAllSolutionsEx()` is used to start encapsulated search. This operator returns multiple solutions that may also include thrown exceptions, using a stream of `Solution` objects that each encode one solution.

In this example, the search region declares a free variable `int n` that is passed as an argument to a method `fact()` which is supposed to compute $n!$. Since `int n` is still unconstrained, all three execution branches remain possible and will therefore be executed

```
2  class MuliIncrementalArgs {
3    public static void main(String[] args) {
4      int x = Muli.getOneSolution(() -> {
5        int x free; int i = 0;
6        while (i < args.length) {
7          if (!(x<Integer.parseInt(args[i++])
8             && x>Integer.parseInt(args[i++])))
9            break; }
10     });
11     System.out.println(x); } }
```

Listing 9.3: Incrementally adding constraints from user input in Muli.

non-deterministically. In our implementation, this means that all possible branches will be tried one after another by backtracking and that all found solutions will be delivered to the resulting stream. One branch is exempted from the overall solution using the `fail()` operator, whereas any other exception would be considered a solution. The first branch imposes the constraint `n == 0` and returns the constant 1. The second branch imposes the constraint `n >= 1` and returns an arithmetic expressions over `n` and recursion. Since `n` is still not bound to a fixed integer value, the expression cannot be evaluated yet and will be represented symbolically. Note that, by recursion, further branching is introduced. These branches will bind `n` to 1, 2, 3, ..., respectively, such that the expression can then be evaluated to an integer and returned as a result. Note also that the finally resulting stream of factorials is infinite. This is not a problem as long as only a finite part of it is actually needed and computed, such as in our example that only prints the first 100 factorials.

Muli realises additional potential in search problems that are not fully defined at once, but that add new, incremental constraints during execution. For instance, Listing 9.3 exhibits a program that iteratively adds constraints over a free variable `x` based on the passed command line parameters and, eventually, finds a solution for `x`. Consider also a variation that derives additional constraints from user input at runtime, e. g. via `BufferedReader.readLine()`, where the full constraint system cannot be known before the application starts.

As a side note, a solution can be a data structure containing the values of possibly several free variables, in case these values are required outside of encapsulation as part of the solution. For instance, this is achieved by the `Assignment` structure that is used in Listing 9.6.

In general, each solution can be a single value if constraints over the involved variables are sufficiently restrictive. In other cases, the solution might describe a search space, i. e. a symbolic expression accompanied by its relevant constraints. If a particular value is required from that search space, the developer needs to explicitly use `solve()` to *label* the variables, i. e. successively try specific values for them, as it is usual in constraint solving. We decided not to do this implicitly, as developers might want to refine search spaces by constraints in later search regions, which they would be unable to do if labelling occurred in the meantime. For example, `solve()` could be called right before returning a solution:

```
1 int x free; if (x >= 4 && x < 7) {
2   Muli.solve(x);
3   return x; }
```

thus ensuring that a single value for `x` (where $x \in \{4, 5, 6\}$) is returned, instead of a logic variable with a domain that is reduced accordingly. A more complex example of using `solve()` in the context of an actual logic problem is provided in the discussion (cf. Listing 9.6).

Moreover, Muli also leverages concepts that were introduced in Java 8. Search regions implement the `Supplier` functional interface, i. e., they are classes with a single method that does not accept parameters and produces a value. As a result, a search region can be elegantly specified using a lambda abstraction with no parameters and a return value, thus implicitly implementing that functional interface. Alternatively, if a search region is used in multiple places, a reference to an appropriate static method is allowed, thus avoiding code duplication. In either case, the evaluation of the search region is deferred until encapsulated search begins. Solutions of encapsulated search are returned to the caller by means of the Stream API in order to facilitate easy handling of solutions by the application.

9.2.2 Syntactic Extension of Java

Our examples suggest that only minimal language extensions are necessary in order to implement these concepts. Syntactically, they are limited to adding the **free** keyword. Given Java's EBNF rules for declaring a field (adapted from [Gos+15]):
FieldDeclaration ::= FieldModifier Type VariableDeclarator (, VariableDeclarator)*;*
VariableDeclarator ::= VariableDeclaratorId (= VariableInitializer)?;
we can add **free** as an alternative to initialisation by changing the *VariableDeclarator* rule to:

*VariableDeclarator ::= VariableDeclaratorId (**free** | (= VariableInitializer))?;*
Free local variables in methods are enabled by an analogous syntax extension. Method parameters cannot be declared free as parameter values are supplied by the caller. However, the caller may pass a free variable as a parameter, but for that purpose no special syntax is required.

Note that the keyword could even be avoided completely. In pure Java, using a `FreeVariable<T>` class with a generic type `T` or an `@FreeVariable` annotation come to mind as alternatives. However, using a class with generic type introduces much overhead, especially when considering boxing and unboxing for free variables of primitive types. Annotations also fall short, since annotations of local variables are not preserved until runtime, whereas only those of class fields remain accessible by the runtime. Using declarations without initialisation is also insufficient as this results in uninitialised local variables or, in the case of fields, in implicitly initialised instance variables. Instead, a compiler can parse the **free** keyword and transform it into bytecode, which is then interpreted by a specialised runtime.

9.2.3 Muli Classpath

The remaining concepts do not require syntactic changes to the language. Therefore, no further additions to the compiler are necessary. Instead, we propose a small library that will be on the classpath during compilation and execution.

A class `Muli` implements encapsulated search operators and the `fail()` operator as static methods that influence the SJVM's runtime behaviour. The most general encapsulated search operator is `getAllSolutionsEx()`, from which further operators are derived.

For reasons of readability we implemented the classpath library directly using Java as far as possible. However, the library has to be able to change the state of the VM in order to switch the execution mode before and after encapsulation, to record solutions, and to enforce backtracking. Methods with that purpose are declared **private static native** and therefore do not provide a direct Java implementation within the library. Instead, the Muli runtime engine provides their actual implementations that perform state changes.

9.3 A Future-Proof Muli Compiler

Variables and class fields that are declared free need to be represented in bytecode accordingly. The JVM specification provides a set of attribute structures that can be extended arbitrarily without breaking bytecode compatibility [Lin+15]. In bytecode,

each attribute declares its type using a string constant, together with its total length in bytes. Every JVM implementation is required to read all attributes and silently skip attributes whose names it does not recognise, using the declared length. We leverage this by specifying custom attribute types that are ignored by regular JVM implementations but that will be picked up by the Muli one. The benefit of staying bytecode compatible with the JVM is that existing parsers and metaprogramming tools are also usable for Muli.

Compiled bytecode comprises four kinds of attribute structures that allow adding custom attribute types [Lin+15]. For instance, for each field of a class there is a `field_info` structure. This structure records information on that field, namely name, descriptor, and a table (i. e. a list) of arbitrary attributes encoding additional details that make use of the behaviour described above. Similarly, the `ClassFile` structure contains (among other things) a table of attributes describing details of the class, the `method_info` structure describes specifics of a method, and `Code_attribute` provides details on the code of a method.

In order to represent free fields, we add a `FreeField` attribute to the `field_info` structure. Since every field maintains its own structure, the mere existence of a `FreeField` attribute is sufficient to indicate that that field is free, whereas its absence implies a regular variable. Therefore, no additional data is required (as illustrated by the empty `FreeField` class in Figure 9.1).

Java maintains a table of local variables in bytecode as part of the structure `Code_attribute` that is associated with a method, thus representing all variables of a method in a single tabular attribute `LocalVariableTable`. Extending the existing attribute by a boolean flag (indicating whether a variable is a free or a regular one) is a feasible alternative in theory, but that would break bytecode compatibility. Therefore, we add a new `FreeVariablesTable` attribute to the `method_info` structure. This attribute maintains one entry per free variable, each comprising a reference to an entry in the `LocalVariableTable` by their index (`FreeVariableEntry` in Figure 9.1), thus specifying which of the local variables is a free one.

As a result, Muli requires a compiler that is able to understand the changed syntax and that generates bytecode as specified above, so that information on free variables and fields can be used at runtime. We have constructed a Muli compiler based on the extensible compiler framework ExtendJ (formerly JastAddJ) [EH07]. Leveraging ExtendJ as a compiler framework comes at several benefits. In contrast to writing a compiler from scratch, we avoid having to redefine the entire frontend and backend, which is useful given that our modification to the original Java syntax is minor. Furthermore,

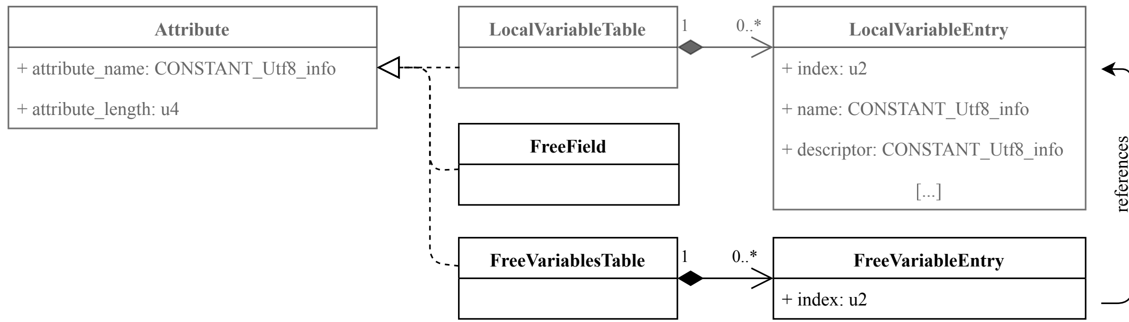


Figure 9.1: Attribute structures generated into the compiled bytecode. Types used in this figure follow the specification of [Lin+15], where u*n* signifies an *n*-byte unsigned integer and `CONSTANT_Utf8_info` is a string constant.

```
FreeVariableDeclarator : VariableDeclarator;
FreeFieldDeclarator : FieldDeclarator;
```

Listing 9.4: Declaration of additional AST subtypes for the Muli compiler.

compilers developed using ExtendJ are easily modifiable because extensions are organised in layers, i. e. the compiler for Java 8 is defined by a small extension layer on top of the Java 7 compiler (which, in turn, is an extension layer to the Java 6 compiler, and so on) [EH07]. Each extension layer modifies the extended compiler in an aspect-oriented way, particularly regarding scanner, parser, and bytecode generation. Similarly, we add a small extension layer for Muli on top of the (ExtendJ) Java 8 compiler. As a consequence, it will be fairly easy to rebase the Muli compiler onto a later Java compiler as soon as we intend to make new Java language features accessible to Muli developers as well.

Our Muli compiler imports abstract syntax tree (AST), parser, and bytecode generator from ExtendJ's Java 8 modules. In the frontend, the AST is extended by two declarator subtypes for free variables (cf. Listing 9.4), that are instantiated by the parser when it encounters the `free` keyword instead of a regular variable initialisation. The parser is modified according to the EBNF specification provided in Subsection 9.2.2. The backend picks up instances of the new AST types and generates the aforementioned attributes into the classes' bytecode correspondingly.

9.4 Operational Semantics of Muli Programs

Following the design goals of our language, execution of Muli programs is identical to the behaviour of a standard JVM [Lin+15] for deterministic program parts, i. e. except within

encapsulated search. In contrast, execution of search regions *inside* of encapsulation can become non-deterministic. This changes the semantics of Java and adds subtleties that need to be explicated, particularly regarding the interaction of imperative statements, free variables, and non-determinism.

Therefore, we formally define the semantics for non-deterministic evaluation of search regions (adapted from [DK18b]), focussing on an imperative, procedural core subset of Muli (and Java). The formulated semantics is helpful to get an understanding of the mechanics behind concepts that are novel to imperative and OO programming and serves as a formal basis for implementing the symbolic JVM. It can also be used for reasoning about applications developed in Muli. In particular, the interaction of imperative statements, free variables, and non-determinism is of interest

For simplicity, this core language abstracts from inheritance, multi-threading, and reflection, because those features do not exhibit interesting behaviour w. r. t. our semantics. Apart from multi-threading, Muli's symbolic JVM supports these features exactly according to the JVM specification [Lin+15] (but does not add interesting details w. r. t. non-determinism). The decision against adding support for multi-threading is sensible as this feature would be highly detrimental to performance in non-deterministic search. The subsequent rules define that non-deterministic branching occurs whenever there are multiple alternatives for statements or expressions that could be evaluated next (as we will demonstrate, for instance, for the rules If_t and If_f). However, in the presence of more than one active thread, there are always multiple alternatives. Specifically, one per active thread in addition to one per non-deterministic instruction that is a candidate in a thread. Consequently, allowing multi-threading in combination with non-deterministic search results in state space explosion, which would require additional sophisticated techniques to combat. In the meantime, until appropriate techniques exist, it is sensible to forbid multi-threading in order to avoid this.

We begin with a description of the syntax of our core language. Variables are taken from a finite set $Var = \{x_1, \ldots, x_m\}$, for simplicity all of type integer ($m \in \mathbb{N}$). In addition, let $Op = AOp \cup BOp \cup ROp = \{+, -, *, /\} \cup \{\&\&, ||\} \cup \{==, !=, <=, >=, <, >\}$ be a finite set of arithmetic, boolean, and relational operation symbols, respectively. We focus on binary operation symbols. Furthermore, $\mathcal{M}$ is a finite set of methods.[10]

The syntax of arithmetic expressions and boolean expressions as well as statements is described by the following grammar. *AExpr*, *BExpr*, and *Stat* denote the sets of all

[10]In this presentation, they are in fact functions as we do not consider object-orientation.

arithmetic expressions, boolean expressions, and statements, respectively, that can be constructed by the rules of this grammar.

$e ::= c \mid x \mid e_1 \oplus e_2 \mid m(e_1, \dots, e_k)$
    where $c \in \mathbb{Z},\ x \in Var,\ e_1, \dots, e_k \in AExpr,\ \oplus \in AOp,\ m \in \mathscr{M},\ k \in \mathbb{N}$,

$b ::= e_1 \odot e_2 \mid b_1 \otimes b_2 \mid \texttt{true} \mid \texttt{false}$
    where $e_1, e_2 \in AExpr,\ b_1, b_2 \in BExpr,\ \odot \in ROp,\ \otimes \in BOp$,

$s ::= \texttt{;} \mid \texttt{int}\ x\texttt{;} \mid \texttt{int}\ x\ \texttt{free;} \mid x = e\texttt{;} \mid e\texttt{;} \mid \{s\} \mid s_1\ s_2 \mid$
      $\texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (b)\ s \mid \texttt{return}\ e\texttt{;} \mid \texttt{fail;}$
    where $x \in Var, e \in AExpr,\ b \in BExpr,\ s, s_1, s_2 \in Stat$.

Note, in particular, the possibility to declare unbound logic variables using e. g. `int` x `free;`.

After describing the syntax of the core language, let us now define its semantics. In the sequel, let $\mathscr{A} = \{\alpha_0, \dots, \alpha_n\}$ be a finite set of memory addresses ($n \in \mathbb{N}$). Moreover, let

$$Tree(\mathscr{A}, \mathbb{Z}) = \mathscr{A} \cup \mathbb{Z} \cup \{\oplus(t_1, t_2) \mid t_1, t_2 \in Tree(\mathscr{A}, \mathbb{Z}), \oplus \in Op\}$$

be the set of all symbolic expression trees with addresses and integer constants as leaves and operation symbols as internal nodes.

We provide a reduction semantics, where the computations depend on an environment, a state, and a constraint store. Let $Env = (Var \cup \mathscr{M}) \to (\mathscr{A} \cup (Var^* \times Stat))$ be the set of all environments, each of which maps a variable to an address and a function to a representation $((x_1, \dots, x_k), s)$ that describes its parameters and code. As an additional restriction, elements of Env may neither map variables to parameters and code nor functions to addresses. We consider functions to be in global scope and define a special initial environment $\rho_0 \in Env$ that maps functions to their respective parameters and code. Moreover, let $\Sigma = \mathscr{A} \to (\{\perp\} \cup Tree(\mathscr{A}, \mathbb{Z}))$ be the set of all possible memory states. In $\sigma \in \Sigma$, a special address α_0 with $\sigma(\alpha_0) = \perp$ is reserved for holding return values of method invocations. Furthermore, $CS = \{\texttt{true}\} \cup Tree(\mathscr{A}, \mathbb{Z})$ is the set of all possible constraint store states. Since constraints are specific boolean expressions, only conjunctions and relational operation symbols such as $==$ and $>$ will appear at the root of such a tree. As a result, the constraint store will comprise a conjunction of atomic boolean expressions.

In the sequel, $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$; if needed, we will also add discriminating indices. We will use the notation $a[x/d]$ when modifying a state or environment a, meaning

$$a[x/d](b) = \begin{cases} d & \text{, if } b = x \\ a(b) & \text{, otherwise.} \end{cases}$$

A free variable is represented by a reference to its own location in memory. Consequently, $\sigma(\rho(x)) = \rho(x)$ if x is a free variable. Initially, a constraint store γ is empty, i. e. it is initialised with `true`. During execution of a program, constraints may incrementally be added to the store. This is done by imposing a conjunction of the existing constraints and a new constraint, thus replacing the constraint store by the new conjunction. As a result, the constraint store is typically described by a conjunction of atomic boolean expressions. For the purposes of describing an operational semantics, we treat the constraint solver as a black box.[11] In our virtual machine implementation, the constraint solver is exchangeable and any solver implementation can be used that meets our requirements as outlined in Subsection 9.5.3.

Note that the above definitions limit variables to an integer type in an attempt to focus the discussion of the semantics. Nevertheless, Muli supports other types which can be represented by making minor modifications to syntax and the subsequent rules. Support for logic variables of non-primitive types is part of future work, therefore non-primitive types are disregarded here. Internally, the JVM represents boolean variables as integers [Lin+15], so adding support for variables of type **`boolean`** is straightforward. The only remaining non-integer primitive types are **`float`** and **`double`**. As they use the same arithmetic expressions as **`int`** (i. e., the rules do not need to be modified), support for these types only depends on the used constraint solver's capabilities – a pure finite domain solver will not provide efficient support for floating-point domains, whereas an SMT solver with support for floating-point arithmetic and finite domains will be well suited.

Moreover, our definition of functions does not fully cover the concept of methods in object-oriented languages, since we abstract from classes and, therefore, inheritance. However, a function in our semantics can be compared to a static method, since a function in this semantics can access and modify its own arguments and variables, but not instance variables of an object. Static fields could be modelled as global variables, i. e. further entries in ρ_0.

Since classes, inheritance, instance variables, and static variables have little influence on the interaction between imperative statements, free variables, and non-determinism, the semantics of object orientation can be considered (almost) orthogonal to our semantics.

[11] A very simple constraint solver could just take equality constraints into account. In this case, $\gamma \vDash x == v$, if $\gamma = b_1 \wedge \ldots \wedge b_k$ and for some $j \in \{1, \ldots, k\}$ $b_k = (x == v)$.

9.4.1 Semantics of Expressions

Let us start with the semantics of expressions. The semantics of expressions is described by a relation $\rightarrow \subset (Expr \times Env \times \Sigma \times CS) \times ((\mathbb{B} \cup Tree(\mathscr{A}, \mathbb{Z})) \times \Sigma \times CS)$, which we use in infix notation. Note that evaluating an expression can, in general, change state and constraint store as a side effect, although only the Invoke rule actively does so. We will point out expressions that make use of this, whereas the others merely propagate changes (if any) resulting from the evaluation of subexpressions.

The treatment of constants and variables is trivial.

$$\langle c, \rho, \sigma, \gamma \rangle \rightarrow (c, \sigma, \gamma), \text{ if } c \in \mathbb{Z} \cup \mathbb{B} \tag{Con}$$

$$\langle x, \rho, \sigma, \gamma \rangle \rightarrow (\sigma(\rho(x)), \sigma, \gamma) \tag{Var}$$

Nested arithmetic expressions without free variables are evaluated directly, whereas expressions comprising free variables result in a (deterministic) unevaluated (!) symbolic expression ($\in Tree(\mathscr{A}, \mathbb{Z})$).

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\quad v_1, v_2, v = v_1 \oplus v_2 \in \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_2, \gamma_2)} \tag{AOp1}$$

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\quad \{v_1, v_2\} \not\subseteq \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (\oplus(v_1, v_2), \sigma_2, \gamma_2)} \tag{AOp2}$$

A boolean expression of the form $e_1 \odot e_2$ is evaluated analogously.

Coherent with Java, conjunctions of boolean expressions are evaluated non-strictly. The rules for the non-strict boolean disjunction operator `||` are defined analogously to the following rules for `&&`.

$$\frac{\langle b_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \gamma \models \neg v_1}{\langle b_1 \ \&\&\ b_2, \rho, \sigma, \gamma \rangle \rightarrow (\mathtt{false}, \sigma_1, \gamma_1)} \tag{And1}$$

$$\frac{\langle b_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \gamma \not\models \neg v_1,\ \ (b_2, \sigma_1, \gamma_1) \rightarrow (v_2, \sigma_2, \gamma_2)}{\langle b_1 \ \&\&\ b_2, \rho, \sigma, \gamma \rangle \rightarrow (\wedge(v_1, v_2), \sigma_2, \gamma_2)} \tag{And2}$$

We consider a function invocation to be an expression as well, as the caller can use its result in a surrounding expression. Evaluation of the function is likely to result in a state change as well as in additions to the constraint store. Invoking m implies that its description $\rho(m)$ is looked up and corresponding fresh addresses $\alpha_1, \ldots, \alpha_k$, one for each of its k parameters, are created. The corresponding memory locations are initialised by the

caller. Note that the respective values can contain free variables. $\sigma_{k+1}(\alpha_0)$ will contain the return value from evaluating the `return` statement in the body, whose semantics will be defined later (cf. rule Ret). As the compiler enforces the presence of a `return` statement, we can safely assume that $\sigma_{k+1}(\alpha_0)$ holds a value after reducing s. Invoke resets that value to $\perp$ for further evaluations within the calling method. We use the shorthand notation $\bar{a}_k = (a_1, \dots, a_k)$ for vectors of k elements.

$$\frac{\begin{array}{c}\langle e_1, \rho, \sigma, \gamma\rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1\rangle \rightarrow (v_2, \sigma_2, \gamma_2),\ \ \dots, \\ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1}\rangle \rightarrow (v_k, \sigma_k, \gamma_k),\ \rho(m) = (\bar{x}_k,\ s), \\ \langle s, \rho_0[{}^{\bar{x}_k}/_{\bar{\alpha}_k}], \sigma_k[{}^{\bar{\alpha}_k}/_{\bar{v}_k}], \gamma_k\rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}),\ \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle m(e_1, \dots, e_k), \rho, \sigma, \gamma\rangle\ \rightarrow (r, \sigma_{k+1}[\alpha_0/\perp], \gamma_{k+1})} \quad \text{(Invoke)}$$

9.4.2 Semantics of Statements

Next, we describe the semantics of statements by a relation $\rightsquigarrow\ \subset (\mathit{Stat} \times \mathit{Env} \times \Sigma \times \mathit{CS}) \times (\mathit{Env} \times \Sigma \times \mathit{CS})$, which we also use in infix notation.

A variable declaration changes the environment by reserving a fresh memory location α for that variable. A free variable is represented by a reference to its own location. Enclosing declarations in a block ensures that changes of the environment stay local.

$$\langle \texttt{int}\ x;, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha], \sigma, \gamma) \quad \text{(Decl)}$$

$$\langle \texttt{int}\ x\ \texttt{free};, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha], \sigma[\alpha/\alpha], \gamma) \quad \text{(Free)}$$

$$\frac{\langle s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle \{\ s\ \}, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1)} \quad \text{(Block)}$$

As a particularity of a constraint-logic OO language, an assignment `x = e` cannot just overwrite a location in memory corresponding to `x`, since this might have an unwanted side effect on constraints that involve `x` and refer to its former value. This side effect might render such constraints unsatisfiable after they have been imposed and checked, thus leaving a currently executed branch in an inconsistent state. We avoid this by assigning a new memory address α_1 to the variable on the left-hand side. At the new address, we store the result from evaluating the right-hand side. Consequently, old constraints or expressions that involve the former value of `x` are deliberately left untouched by the assignment. In contrast, later uses of the variable refer to its new value. The environment is updated to achieve this behaviour.

$$\frac{\langle e, \rho, \sigma, \gamma\rangle \rightarrow (v, \sigma_1, \gamma_1)}{\langle x = e, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha_1], \sigma_1[\alpha_1/v], \gamma_1)} \quad \text{(Assign)}$$

Since the syntax does not enforce that no statements follow a `return` statement, we provide sequence rules that take into account that the state may hold a value in α_0 (indicating a preceding `return`) or not ($\bot$). Further statements are executed iff the latter is the case. Otherwise, further statements are discarded as a preceding `return` has already provided a result in α_0.

$$\frac{\langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) == \bot, \quad \langle s_2, \rho_1, \sigma_1, \gamma_1 \rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)}{\langle s_1 \;\; s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)} \tag{Seq}$$

$$\frac{\langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) \neq \bot}{\langle s_1 \;\; s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \tag{SeqFin}$$

The two following rules for if-statements introduce non-determinism in case that the constraints neither entail the branching condition nor its negation.In the implementation, the applicability of these rules depends on the constraint propagation abilities of the employed constraint solver, which may allow the runtime environment to detect infeasible branches early (cf. Subsection 9.5.3).

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \nvDash \neg v, \quad \langle s_1, \rho, \sigma_1, \gamma_1 \wedge v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \tag{If$_t$}$$

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \nvDash v, \quad \langle s_2, \rho, \sigma_1, \gamma_1 \wedge \neg v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \tag{If$_f$}$$

As with `if`, the evaluation of a `while` statement can also result in the introduction of non-determinism.

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \nvDash \neg v, \quad \langle s, \rho, \sigma_1, \gamma_1 \wedge v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2), \; \langle \texttt{while}\ (b)\ s, \rho_1, \sigma_2, \gamma_2 \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)}{\langle \texttt{while}\ (b)\ s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)} \tag{Wh$_t$}$$

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \nvDash v}{\langle \texttt{while}\ (b)\ s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1 \wedge \neg v)} \tag{Wh$_f$}$$

All branching rules If$_f$, If$_t$, Wh$_f$, and Wh$_t$ could be accompanied by more efficient ones that deterministically choose a branch if its condition does not involve free variables, i. e. without having to consult the constraint store. We omit these rules in an effort to keep our definitions concise, as the provided ones can also handle these cases.

We assume that the code of a user-defined function is terminated by a `return` statement, i. e. the existence of this statement needs to be ensured by the compiler. The corresponding return value is supplied to the caller by storing it in α_0, causing remaining statements of

the function to be skipped (cf. rule SeqFin), and letting the caller extract the result from α_0 (cf. rule Invoke). The `return` statement is handled as follows:

$$\frac{\langle e, \rho, \sigma, \gamma \rangle \to (v, \sigma_1, \gamma_1)}{\langle \texttt{return}\ e, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1[\alpha_0/v], \gamma_1)} \quad \text{(Ret)}$$

Furthermore, we do not define an evaluation rule involving a `fail` statement. This is intentional, as the evaluation of such a statement leads to a computation that fails immediately.

The following (optional) substitution rule allows simplifying expressions and results.

$$\frac{\gamma \models \gamma(\alpha) == v,\ \ \langle s, \rho, \sigma[\alpha/v], \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \quad \text{(Subst)}$$

When variables are not sufficiently constrained to concrete values, *labelling* can be used to substitute variables for values that satisfy the imposed constraints [FA03]. This non-deterministic rule is applied with the least priority, i. e. it should only be used if no other rule can be applied. Otherwise, it would result in a lot of non-deterministic branching, thus preventing the constraint solver from an efficient reduction of the search space by constraint propagation.

$$\frac{\gamma \not\models \sigma(\alpha) \neq v,\ \ \langle s, \rho, \sigma[\alpha/v], \gamma \wedge (\sigma(\alpha) == v) \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \quad \text{(Label)}$$

9.4.3 Evaluation of an Example Program

We demonstrate the use of the reduction rules using a simple example program. We compute one possible result of the Muli method depicted in Listing 9.5 that is invoked by an additional method

```
public static int search() { return twoPower(0); },
```

thus yielding a solution 2^x non-deterministically, where x is an integer greater or equal to 0.[12]

In order to present the computations in a readable form, the code of `twoPower` is abbreviated by s_1. Furthermore, s_1 includes the statements s_2 = **return** `Math.pow(2, y);` and s_3 = **return** `twoPower(y+1);`. Moreover, nested expressions use infix notation, e. g., instead of $+(0,1)$ we write $0 + 1$. Last but not least, we decompose the full computation into a couple of lemmas in order to simplify understanding it. The computation is

[12]Correspondingly, a program `Muli.getAllSolutions(() -> return search())` would enumerate *all* 2^x, $x \geq 0$.

```
1  public static int twoPower(int y) {
2    int coin free;
3    if (coin == 0) return Math.pow(2, y);
4    else return twoPower(y+1); }
```

Listing 9.5: Muli program that non-deterministically searches for a solution 2^a where a integer, $a \geq y$ for a given y.

presented top-down. If you prefer a bottom-up fashion, just read the lemmas in reverse order. Each step specifies the names of the rules that are applied in it.

Initially, let $\rho_0 = \{search \mapsto (\epsilon, \texttt{return twoPower(0);}),\ twoPower \mapsto ((y),\ s_1)\}$. As intermediate results, some auxiliary definitions are needed: $\rho_1 = \rho_0[y/\alpha_1]$, $\rho_2 = \rho_1[coin/\alpha_2]$, $\sigma_1 = \sigma_0[\alpha_1/0]$, $\sigma_2 = \sigma_1[\alpha_2/\alpha_2]$, and $\sigma_3 = \sigma_2[\alpha_0/1]$. In addition, the initial constraint store is trivially satisfiable ($\gamma_1 = true$) and the initial state is $\sigma_0 = \{\alpha_0 \mapsto \bot\}$. Computation begins with the evaluation of `search()` as follows.

$$\cfrac{\cfrac{\langle 0, \rho_0, \sigma_0, \gamma_1 \rangle \to (0, \sigma_0, \gamma_1)\ \text{(Con)},\quad \rho_0(twoPower) = ((y), s_1),\ \text{(Lemma 1)},\ \sigma_3(\alpha_0) = 1}{\langle \texttt{twoPower(0)}, \rho_0, \sigma_0, \gamma_1 \rangle \to (1, \sigma_3[\alpha_0/\bot], \gamma_2)}\ \text{(Invoke)}}{\langle \texttt{return twoPower(0);}, \rho_0, \sigma_0, \gamma_1 \rangle \rightsquigarrow (\rho_0, \sigma_3, \gamma_2)}\ \text{(Ret)}$$

Subsequently, invoking `twoPower` with the parameter 0 changes environment and state, so that the parameter is known for the evaluation of the method body. The sequence of statements in the body evaluates as

$$\cfrac{\langle \texttt{int coin free;}, \rho_1, \sigma_1, \gamma_1 \rangle \rightsquigarrow (\rho_1[coin/\alpha_2], \overbrace{\sigma_1[\alpha_2/\alpha_2]}^{\sigma_2}, \gamma_1)\ \text{(Free)},\quad \sigma_2(\alpha_0) == \bot, \text{(Lemma 2)}}{\langle \texttt{int coin free; if(coin == 0)}\ s_2\ \texttt{else}\ s_3, \rho_1, \sigma_1, \gamma_1 \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_2)}\ \text{(Seq)} \qquad \text{(Lemma 1)}$$

The declaration of a free variable is evaluated deterministically, whereas evaluating **if** introduces non-determinism as there are possible bindings for *coin* that render the condition either **true** or **false**. Therefore, either If_t or If_f are applicable, respectively. For the purpose of this example, we continue with choosing the If_t rule, thus following the **true** branch.

$$\dfrac{\dfrac{\langle coin, \rho_2, \sigma_2, \gamma_1\rangle \to (\alpha_2, \sigma_2, \gamma_1)\ \text{(Var)},\quad \langle 0, \rho_2, \sigma_2, \gamma_1\rangle \to (0, \sigma_2, \gamma_2)\ \text{(Con)},\ \{\alpha_2, 0\} \not\subseteq \mathbb{Z}}{\langle coin == 0, \rho_2, \sigma_2, \gamma_1\rangle \to (\alpha_2 == 0, \sigma_2, \gamma_1)}\ \text{(AOp2)},\quad \gamma_1 \nvDash \neg(\alpha_2 == 0),\ \text{(Lemma 3)}}{\langle \texttt{if(coin == 0)}\ s_2\ \texttt{else}\ s_3, \rho_2, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_2)}\ (\text{If}_t) \qquad \text{(Lemma 2)}$$

The constraint store is updated, reflecting that the **true** branch has been chosen: $\gamma_2 = \gamma_1 \wedge \alpha_2 == 0$. As a result, future evaluations take the additional constraint into account. Subsequently, evaluation continues with the code of the **true** branch. Given that in this example the common Java method `Math.pow(a,b)` only operates on bound variables, we treat it as a black box that evaluates to a^b without making modifications to state and constraint store. Therefore, the following invocation is abbreviated.

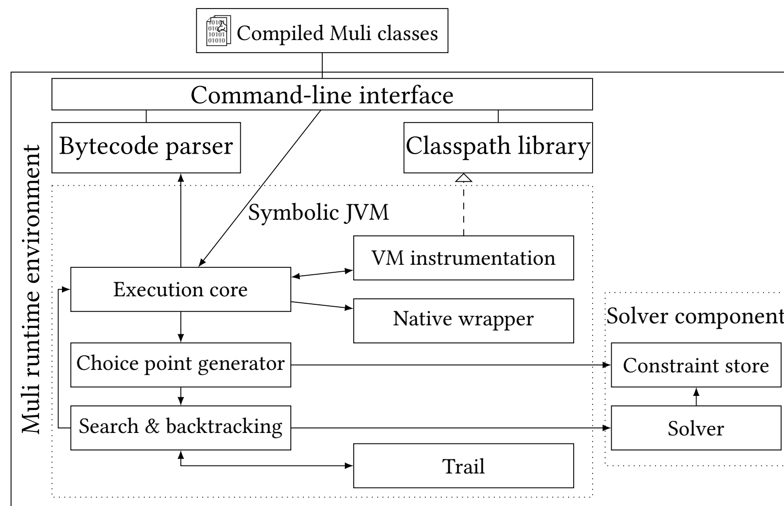
$$\dfrac{\dfrac{\langle 2, \rho_2, \sigma_2, \gamma_2\rangle \to (2, \sigma_2, \gamma_2)\ \text{(Con)},\quad \langle y, \rho_2, \sigma_2, \gamma_2\rangle \to (0, \sigma_2, \gamma_1)\ \text{(Var)},\ 1 = 2^0}{\langle \texttt{Math.pow}(2, \texttt{y}), \rho_2, \sigma_2, \gamma_2\rangle \to (1, \sigma_2, \gamma_2)}\ (Invoke, abbrev.)}{\langle \texttt{return Math.pow}(2, \texttt{y});, \rho_2, \sigma_2, \gamma_2\rangle \rightsquigarrow (\rho_2, \underbrace{\sigma_2[\alpha_0/1]}_{\sigma_3}, \gamma_2)}\ \text{(Ret)} \qquad \text{(Lemma 3)}$$

In the final state, $\sigma_3 = \sigma_0[\alpha_1/0, \alpha_2/\alpha_2, \alpha_0/1]$. Therefore, the final result is $\sigma_3(\alpha_0) = 1$, assuming a constraint store $\gamma_2 = \alpha_2 == 0$. Other possible outcomes of non-deterministic search can be computed analogously by choosing differently in Lemma 2.

In this example, the free variable `coin` is only used in a condition in order to create non-deterministic choice, but never becomes part of a solution. Therefore, the Label rule is never used, thus keeping the state space minimal. As an additional reference, [DK18b] demonstrates and discusses the evaluation of a different example program that also includes the use of the Label rule.

9.5 A Backtracking, Symbolic Virtual Machine

In addition to using Java as the reference language for an extension, we also choose Java as an implementation platform for the virtual machine runtime. Incidentally, this makes the resulting constraint-logic OO language just as platform-independent as Java. As a positive side effect, we are able to leverage the multitude of third-party libraries written for the JVM that are useful for our work, constraint solvers in particular.



[DK18a]

Figure 9.2: Components structure of the Muli runtime environment.

The heart of the Muli runtime environment is the SJVM, i. e. a custom virtual machine that symbolically executes Java programs. Muli's SJVM builds on earlier work that was directed at automated glass-box test case generation for Java programs [MK09], allowing a tester to load a Java class and specify a method for which test cases are to be generated. Symbolic execution of that method in the SJVM resulted in a systematic discovery of all execution paths while creating a set of path constraints involving the method parameters. Finally, a constraint solver is used to provide particular values for unit test cases. The Muli runtime environment generalises that work, thus evolving the SJVM into a self-contained general purpose runtime. A user interacts with the command-line interface (CLI) that accepts a list of bytecode-compiled Muli classes as a classpath and a compiled Muli class to be run (in the following; *Application*). The CLI adds the Muli classpath library (see Subsection 9.2.3) to the classpath list, parses the application class from bytecode, initialises the SJVM, and invokes the class's `public static void main(String[])` method on the execution core of the SJVM. Figure 9.2 illustrates the structure of the runtime environment, depicting relevant components and their interrelations.

Since Muli programs are compiled to JVM-compatible bytecode, a specialised bytecode parser is not required. Therefore, the runtime environment executes all applications, regardless of whether they were implemented in Java or Muli. Nevertheless, only Muli programs can specify logic variables that are picked up by the runtime. Java programs by themselves will only be executed deterministically. However, Muli programs are fully able to reference Java classes and interact with them non-deterministically. Consequently,

it is possible that Java methods are executed symbolically, as long as they are invoked from a Muli program.

We explain the components and the concepts that influenced their implementation in the following. As Muli's SJVM is implemented in Java, it also runs on a JVM which will be referred to as the *enclosing JVM* for distinction.

9.5.1 Data Structures

The Muli runtime environment implements all data structures that are required to execute Java programs, according to the JVM specification [Lin+15]. This includes data structures that contain object representations of class files, i. e. the result after parsing them. Standard classes of Java, such as `java.lang.String`, need not be re-implemented, as they are parsed from the enclosing virtual machine's classpath. The most important one, the *Frame stack*, consists of frames, each representing an executed method call. The top of the stack corresponds to the method that is currently being executed. When it invokes another method, a frame for the invoked method is instantiated and pushed to the stack. On return, the topmost element of the frame stack is popped and the SJVM continues execution of the new top. Recursive invocations of the same method result in multiple frame instances for that method, each individually recording its respective state, so that the recursion can be unrolled later.

Within a frame, the represented execution state comprises the program counter, values of local variables, and an *operand stack*. As in a regular JVM, most bytecode instructions operate on the operand stack, taking their input(s) from the top of the stack and pushing computation results. In contrast to a regular JVM, the operand stack does not only contain constant primitive values or addresses to objects. Instead, elements can also be symbolic representations of free variables or expressions that result from computations involving another symbolic representation (cf. rule AOp2 in Section 9.4).

For simplicity, the SJVM does not implement its own heap. Instead, it shares the heap with the enclosing JVM. Moreover, the SJVM also does not implement own garbage collection mechanisms, as those of the enclosing JVM are sufficient. Our rationale behind this is that an own implementation of the heap would not provide advantages over the existing one for Muli, since the heap does not influence symbolic execution or search. Nevertheless, backtracking affects objects on the heap as well.

The aforementioned structures are required for a bytecode-compatible runtime that is modified to support symbolic execution and search. More importantly, the SJVM implements data structures derived from the Warren Abstract Machine (WAM) that is

specifically designed to execute Prolog programs [War83]. In the WAM, the local stack comprises *environments*, which can be roughly compared to Muli's and Java's frame instances, and *choice points*, for which there is no corresponding structure on a regular JVM. Thus, the SJVM stores choice points in an additional *choice point stack*. Each choice point maintains a *trail*, which is a concept also borrowed from the WAM [War83]. The trail is implemented as a stack as well. Each element represents an operation that must be performed on some component of the SJVM in order to undo a state change.

Additionally, the solver component of Muli maintains the *constraint store*. During execution of a program, constraints may be incrementally added to the store. Typically, the constraint store is described by a conjunction of atomic boolean expressions that correspond to branching criteria. Every choice point also maintains references to constraints that were added by it, in order to remove them from the store on backtracking.

Last but not least, the SJVM has status flags that control its execution. Execution can either be *deterministic*, i. e. non-searching, or *non-deterministic*, i. e. searching. The latter state is only assumed during encapsulated search, ensuring deterministic execution outside the encapsulation. Further flags include e. g. the search strategy (currently only iterative deepening depth-first search is supported) and the requested logging level, which can provide helpful output during our development of the VM.

9.5.2 Symbolic Types

Muli supports all types known from regular Java, including reference types and arrays. However, in order to accommodate for logic variables, additional types are introduced. According to the JVM specification, two basic kinds of types need to be distinguished: Primitive types and reference types [Lin+15]. However, we further split considerations of reference types into array reference types and object reference types due to their distinct structure.

Logic arrays are represented by instances of an `Arrayref` class, maintaining a logic array's element type (e. g. **`int`**), its dimensions, and its element values, which can be either regular values or logic variables. Similarly, logic objects are represented using `Objectref` instances, each containing its class type and a map of its fields to their respective values, which can also be logic variables. A logic variable of numeric primitive type is an instance of class `NumericVariable`. It contains a flag indicating the particular primitive type. It does not have a value, but its domain is restricted by constraints in the constraint store. Analogously, logic boolean variables are described by `BooleanVariable` instances.

When encountering an instruction that performs operations on variables, the semantics of the SJVM does not distinguish between logic variable types and regular variable types. For example, primitive `int` variables are type-compatible to `NumericVariable` representations with integer type. Performing an `iadd` operation on an `int` variable and an `NumericVariable` will result in a symbolic expression representing the addition. The result is then pushed to the operand stack.

9.5.3 Solver Component

The logic variable types are part of the *Solver component*, which is the runtime's abstraction layer from constraint solvers. It specifies types that are used to generate variables, expressions, and constraints during execution without having to consider particularities of constraint solvers. Before they become part of a constraint, expressions collected during symbolic execution are not mapped into a particular solver's object representation.

Currently, Muli integrates two constraint solvers from which a user can choose. Muconst is an SMT solver that was originally developed with automated glass-box test case generation in mind [Lem+04]. It supports linear and non-linear arithmetic theories as well as SAT solving. Its distinguishing advantage over other solvers is its handling of rounding errors in floating point solutions, ensuring that solutions still satisfy all constraints after rounding [EMK12]. Alternatively, the finite domain solver JaCoP is integrated, which is a free software constraint solver library for Java [Kuc03]. JaCoP's constraint propagation achieves early, computationally inexpensive detection of infeasible branches for applications that mostly involve finite domain numeric variables.

The SJVM is able to work with multiple constraint solvers and abstractly defines constraints, so that the constraint solver can be treated as a black box for most considerations. Nevertheless, we formulate the following requirements for a constraint solver to be applicable: It needs to be able to label variables in order to find solutions where constraints are not sufficiently restrictive for finding solutions. Moreover, variables and labelling strategies for both finite domain problems and floating point problems need to be present and the constraint solver must be able to handle combinations of these problems.

9.5.4 Symbolic Execution, Encapsulated Search, and Choice Points

Symbolic execution affects the interpretation of many Java bytecode instructions. For example, when an instruction loads a free variable, a corresponding symbolic representa-

Triggering bytecode instruction	Choice point type	Possible constraints
FCmpg, FCmpl, DCmpg, DCmpl	floating point comparison	$=, <, >$
LCmp	long comparison	$=, <, >$
If<cond>, If_icmp<cond>	**if** instruction, integer comp.	$=, \neq, <, \leq, >, \geq$
Lookupswitch, Tableswitch	**switch** instruction	$=, \notin$

[DK18a]

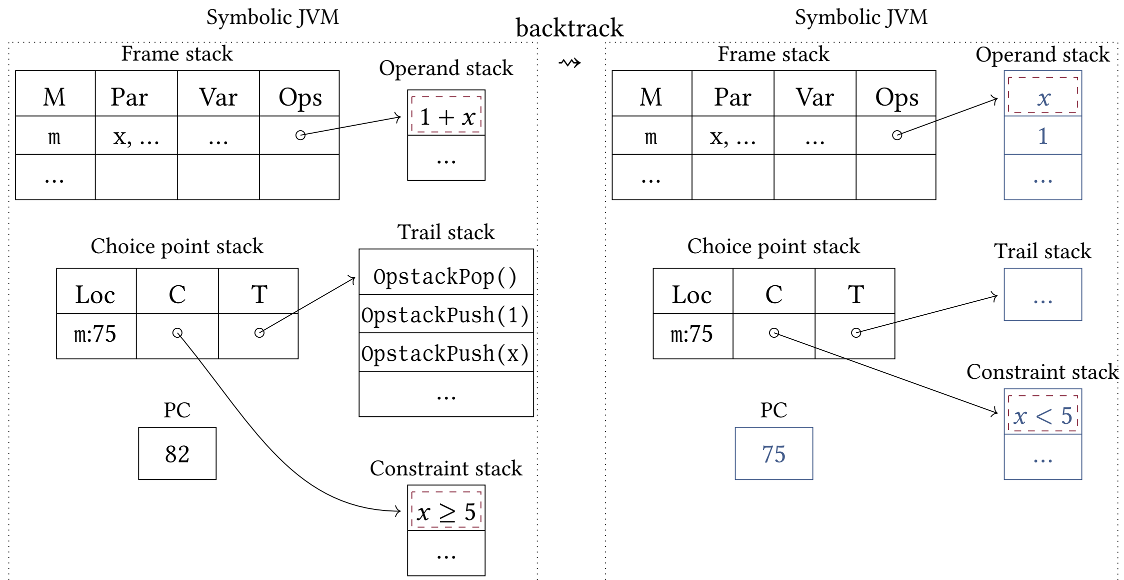
Table 9.1: Bytecode instructions, resulting choice points, and applicable constraint types. <cond> is one of eq, ne, lt, le, gt, or ge.

tion is pushed to the operand stack. Arithmetic operations that manipulate the operand stack involving symbolic representations push a symbolic representation of the result. Analogously, relational operators result in symbolic relational expressions.

Non-determinism may be introduced if a branching condition contains logic variables or symbolic expressions (cf. e. g. rules If_f, If_t in Section 9.4). On the bytecode level, this corresponds to e. g. a conditional jump instruction such as ifne with two possible outcomes. Also, bytecode instructions that may cause an exception (e. g., getfield, invokeinterface, invokevirtual, and checkcast), can cause branching. Whenever non-determinism is introduced, the runtime environment instantiates a choice point and pushes it to the choice point stack, representing the state before branching, the possible choices, as well as state regarding the choices that already been evaluated. Depending on the branch that is chosen first, the branching condition or its negation will be pushed to the constraint stack and the execution will be continued at the corresponding instruction. The other branch will be considered after backtracking. Table 9.1 shows a selection of typical bytecode instructions which may cause branching, as well as their corresponding choice point types and possibly generated constraints.

Each choice point maintains a trail, which is a stack that, for every executed instruction, records the operation that will be necessary to reverse the effects of that instruction. For example, executing an instruction that takes two elements from the operand stack and pushes one results in recording three trail elements. One that will take one element from the operand stack and two that will push the values of the previous elements to the stack.

At the end of a symbolic execution path (i. e. when encountering the final **return** or a non-caught exception e. g. caused by **throw**), the SJVM will backtrack to the most recent choice point and undo the corresponding changes recorded on the trail by replaying its trail stack. Moreover, the constraints imposed since the choice point will be removed from the constraint store. Afterwards, the next choice is realised by imposing its constraint and execution continues. Figure 9.3 illustrates how backtracking rolls back the trail, removes



[DK18a]

Figure 9.3: Effect of backtracking on execution state. As a result of backtracking, the operand stack, trail stack, constraint stack, and program counter have changed (highlighted blue in the web version of this article). Operand stack elements with dashed lines reside on the heap, which we omitted for simplicity.

a choice point's constraint, and imposes the next choice's constraint before execution continues. When no further choice can be realised, the choice point is removed from the choice point stack and backtracking to the previous choice point occurs.

To illustrate the execution of non-deterministic search using an example, consider the program presented in Listing 9.2 that generates factorials. The first levels of the symbolic execution tree, or search tree, that corresponds to this program are depicted in Figure 9.4. Execution of the search region begins by declaring a free variable `n` that is passed to `fact()`. There, the free variable used directly as part of boolean (in)equalities, resulting in non-deterministic branching. To that end, a choice point (Choice_0 in Figure 9.4) with three choices is created.[13]

Choosing the first alternative binds `n` to the value `0` by imposing the corresponding constraint and returns `1`, i. e. `0`!. This marks the end of this path through the search tree. Consequently, the returned value is a solution of this search region (Solution_0) that will later be returned after encapsulated search is finished. After backtracking to Choice_0

[13]In fact, bytecode **`if`** is binary, as a single condition can evaluate to either **`true`** or **`false`**. As a result, Choice_0 would actually correspond to two choice points in this program with two alternatives each, one being an alternative of the other. One then represents the **`else if`** and **`else`** branches. In this presentation, we merge the two into a single choice point for simplicity.

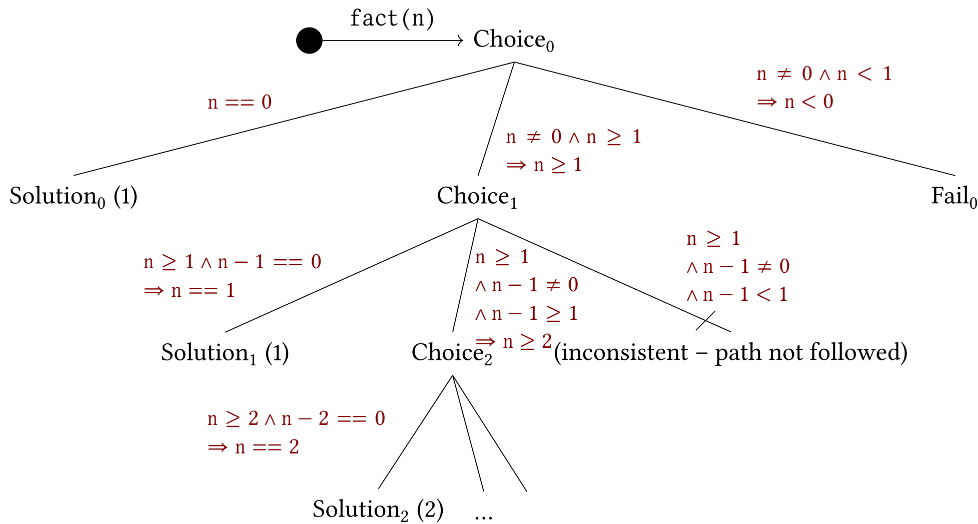
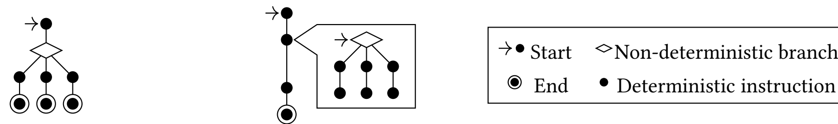


Figure 9.4: Excerpt from the search tree that is effectively generated by executing the search region depicted in Listing 9.2.

(thus removing the previous constraint and reversing effects on the SJVM), choosing the second alternative imposes two constraints in conjunction. First, the negation of the first constraint, i. e. $n \neq 0$. Second, $n \geq 1$ as that is the condition for **else if**. These constraints can be simplified to $n \geq 1$. The second alternative results in recursion, passing $n - 1$ as the parameter. Consequently, Solution$_1$ is reached by making the appropriate choice in *Choice*$_1$ and adding the constraint $n - 1 == 0$ to the constraint stack, thus imposing it in conjunction with the previous constraint. Considering Choice$_0$ again, the constraint for the third alternative is the conjunction of negations of both conditions, which simplifies to $n < 0$. As the factorial is only defined for non-negative integers, we invoke `Muli.fail()` in order to actively discard this branch (we mark this by denoting Fail$_0$). Hence, no solution is added. Note that the path constraints for making an analogous choice in Choice$_1$ are inconsistent as they simplify to $n > 1 \wedge n < 2$. Therefore, the SJVM will not make that choice in the first place, thus saving execution time.

For the search strategy, we currently rely on iterative deepening depth-first search, following the ideas of Java PathFinder [VPK04] and Muggl [MK11b]. Like depth-first search, iterative deepening depth-first search facilitates memory-efficient search. However, depth-first search does not cope well with situations in which a sequence of choosing the first alternatives leads to a long path in the search tree (or to an infinite one, such as from the evaluation of infinite loops), whereas other alternatives would result in finding a solution earlier. In contrast, iterative deepening depth-first search ensures that we will



[DK18a]

Figure 9.5: Unrestricted symbolic execution versus encapsulated symbolic execution.

quickly find solutions corresponding to short paths, which is particularly useful if just a few solutions are required.

As we do not want to support non-determinism outside of encapsulated search regions, non-deterministic jumps are restricted to the searching mode of the SJVM. Therefore, encapsulation bounds the execution tree at the end of search regions, ensuring that effects of symbolic execution and backtracking remain local. Consequently, at the end of encapsulated search regions, the control flow is linearised again and the collected solutions (or solution spaces) are returned to the caller (see Figure 9.5). Encapsulated search can also be nested, thus achieving search hierarchies. If an unbound logic variable or a symbolically represented expression is accessed outside of a search region, an exception will be thrown.

It is debatable whether input/output should be disallowed in search regions, as this introduces side effects that cannot be backtracked by the SJVM. These issues are known from Prolog [Sco10]. On the other hand, such side effects may be wanted, as they facilitate printing (partial) solutions, logging, and asking end-users to supply additional data that may only be relevant in certain execution paths. Therefore, we decided not to forbid possible non-backtrackable side effects. Instead, as with many advanced programming constructs, we require software developers to assume responsibility to check whether this is expected behaviour.

9.6 Discussion

Our approach towards an integration of constraint-logic and OO programming is useful for applications that are mainly programmed in an object-oriented language, e. g. Java, while requiring a substantial amount of search. We expect it to be particularly suited for programs in which new constraints are discovered over time that are incrementally added to the existing set of constraints. The implementation of the constraint store facilitates such applications by reusing results from former searches when new constraints are

```
1  public static Assignment money() {
2    int s free, e free, n free, d free, m free, o free, r free, y free;
3    if (domain(s,e,n,d,m,o,r,y) && s != 0 && m != 0 &&
4              diff(s,e,n,d,m,o,r,y)) {
5      if (           1000*s + 100*e + 10*n + d +
6                     1000*m + 100*o + 10*r + e ==
7           10000*m + 1000*o + 100*n + 10*e + y) {
8        Muli.solve(s,e,n,d,m,o,r,y);
9        return new Assignment(s,e,n,d,m,o,r,y);
10     } else throw Muli.fail();
11   } else throw Muli.fail(); }
```

Listing 9.6: Muli search region implementing the Send More Money Puzzle; class headers and helper functions omitted.

added, thus preventing recomputation of partial solutions unless this cannot be avoided. Further benefits are achieved by solvers with effective constraint propagation.

We quantify the performance of Muli programs in our runtime environment. Moreover, we demonstrate how Muli improves the programming style for search problems over pure Java using example applications.

The example application in Listing 9.1 already demonstrates constraint solving, although with a problem that can be solved trivially in an imperative language as well. Less trivial is an application that solves the *Send More Money* Puzzle: Eight free integer variables s, e, n, d, m, o, r, and y need to be labelled with values from 0 to 9 such that every binding is different from the others, while satisfying the constraint $1000s + 100e + 10n + d + 1000m + 100o + 10r + e = 10000m + 1000o + 100n + 10e + y$. Additionally, $s \neq 0$ and $m \neq 0$. This can be specified as a Muli search region as shown in Listing 9.6.

The helper method `diff` imposes the constraint that every variable be different from the others, while `domain` limits the variables' domains to $\{0, \dots, 9\}$. The `Assignment` class that is used here is a simple data structure comprising the eight variables, which is used to return all eight bindings.

We have compared the runtime of the application provided in Listing 9.6 with that of a corresponding pure Java application that attempts an imperative solution using eight **for** loops (one per variable) and backtracking. As an additional example application we implemented the Safe Lock Key puzzle, each also in Muli and pure Java. Moreover, to be able to observe the runtime behaviour of a set of problems with increasing size we implemented the n-Queens problem, again in Muli and pure Java. We then executed it

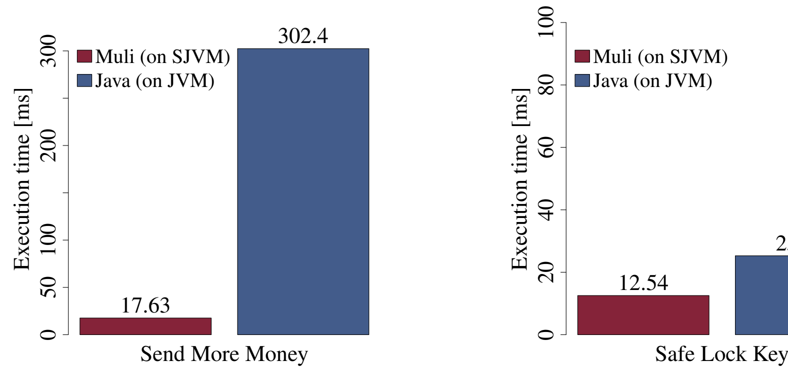


Figure 9.6: Comparison of sample implementations by mean execution time (in milliseconds), each averaged over 500 executions.

with increasing *n*, thus being able to compare the implementations for problems with a similar structure while increasing the solution space as well as the number of imposed constraints.

The Muli applications have been executed on the Muli SJVM, using the JaCoP-based finite domain solver in the solver component. In order to exclude possible overhead of the SJVM, we have executed the pure Java applications on a regular OpenJDK JVM (version 1.8.0_191). All experiments were executed on an Intel Core i5-5200U CPU, using Ubuntu 18.04.2 with a 4.15.0 x64 Kernel. Each application attempts to solve the puzzle 510 times. The first 10 results are dropped in order to disregard effects of Just-In-Time compilation of the enclosing JVM.

The experimental results for the Send More Money problem indicate that the constraint-logic OO implementation, while more elegant to write, is also consistently more than an order of magnitude faster at finding a solution for the problem at hand (cf. Figure 9.6). Similarly, the Muli implementation of the Safe Lock Key puzzle is faster than its counterpart in Java, except for the maximum execution time, where the Muli implementation was slightly slower (37.13 ms compared to 33.31 ms in Java). The relative difference between minimum and maximum is higher for Muli in both sample applications (e. g., Send More Money in Muli ranges from 8.74 ms to 43.59 ms, whereas in Java it ranges from 294.39 ms to 416.28 ms). This could be due to the increased need for garbage collection over (unused) object representations of symbolic expressions and constraints, whereas the regular Java version can naturally use primitive values only. Nevertheless, we can still see a major advantage in performance for the Muli variants, particularly for larger problems. This improvement can be attributed to constraint propagation by the finite domain solver, allowing for more efficient search than simple backtracking mechanisms.

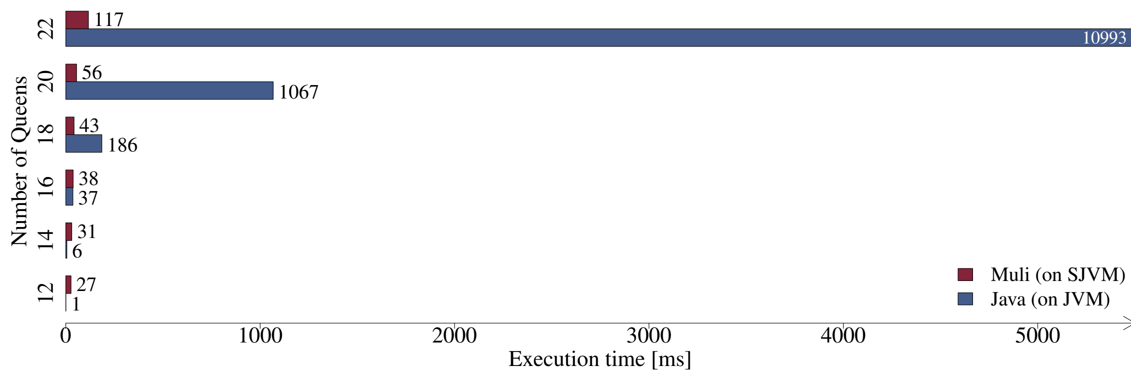


Figure 9.7: Comparison of implementations solving the n-Queens problem with increasing n by mean execution time (in milliseconds), each averaged over 500 executions.

We have limited ourselves to executing the n-Queens experiments to $n \in \{12, 14, \dots, 22\}$, because the solution space for $n = 12$ is already larger than those of the Send More Money and Safe Lock Key problems; and the execution time of the Java implementation already exceeds more than 10 seconds per execution with $n = 22$. The results of these experiments demonstrate that, for smaller n, a pure Java implementation has an advantage over a Muli implementation (cf. Figure 9.7). This can be attributed to the fact that a pure Java solution has less overhead as opposed to a Muli program that first defines free variables and later has to solve them explicitly. Furthermore, after decades of development the official OpenJDK JVM is of course more optimised than our novel, custom SJVM. However, that advantage diminishes quickly with increasing n, i. e., with increasing solution space, where the benefits of a constraint-logic OO implementation become visible. The execution times of the Muli implementation increase with increasing n, which can be explained with the time that is required for defining the additional constraints as well as for labelling additional variables. Despite this increase, the total execution time of an average single run for $n = 18$ using Muli is less than a quarter of the corresponding average time of the Java implementation. For $n = 22$, the time spent finding a solution using Muli is still hardly noticeable by a human (about 117 ms), whereas executing the Java implementation requires more than ten seconds, i. e. about two orders of magnitude more time, and is therefore impractical.

Another highly interesting application scenario for Muli is automated glass-box test case generation, as it is a prime example for incrementally added constraints [EMK12]. In order to generate JUnit assertions for a Java method, we need to create an appropriate output from free parameters and the method's results. For the factorial method from Listing 9.2, this can be done by changing the search region as demonstrated in Listing 9.7.

```
1  Muli.getAllSolutions(() -> {
2    int n free;
3    int factorial = fact(n);
4    String test = "assertEquals(" + factorial + ", fact("+n+"));");
5    return test; });
```

Listing 9.7: Modification of the factorials search region from Listing 9.2 in order to generate JUnit assertions for testing `fact()`.

Moreover, consider that Listing 9.7 could be extended to generating integration test cases by calling sequences or compositions of methods. As `fact(n)` has an infinite search space, writing the assertion into a file before returning it results in output into that file until execution is interrupted (manually).

Furthermore, the example in Listing 9.3 demonstrates how Muli facilitates a simpler programming style. In contrast, writing a similar program in Java is much more challenging, particularly for handling backtracking and negation in case that the constraint store is rendered inconsistent by a recent addition (cf. Listing 9.8), thus introducing more potential for implementation mistakes. Declaring a logic variable in Muli is syntactically close to declaring any variable in Java, as `int name free;` is not a long way from `int name;`. In contrast, using the JaCoP API results in the formulation `IntVar name = new IntVar(store, name, min, max)` which requires developers to think of several aspects in addition to just the variable itself. First, they need to use the JaCoP-specific object representation of a logic variable that corresponds to the intended base type (such as, `IntVar` for `int`). Second, they have to specify the constraint store that is in use and remember to have it instantiated first. Third, the declaration is partly redundant; having to specify an internal name in addition to the variable's name. Consequently, the shorter Muli version requires less syntactic and cognitive overhead and is, therefore, more natural to a developer who is used to programming in Java. Additionally, the constraint system that is active in Muli can always be assumed to be consistent, because imposing a new constraint that would render the constraint system inconsistent results in immediate backtracking. In contrast, developers who are using a library need to actively check for consistency.

The comparison of the program versions provided in Listings 9.3 and 9.8 demonstrates that Muli also has an advantage w. r. t. the representation of constraint-logic problems compared to using a library in Java. This advantage can be emphasised by comparing the Muli implementation of Send More Money (Listing 9.6) to an implementation that uses

```
3  class JacopIncrementalArgs {
4    public static void main(String[] args) {
5      Store store = new Store();
6      int i = 0;
7      IntVar x = new IntVar(store, "x", -1000000, 1000000);
8      while (i < args.length) {
9        // Prepare for later backtracking.
10       int backtrackingLvl = store.level+1;
11       store.setLevel(backtrackingLvl);
12       // Create constraints from user input.
13       int c1 = Integer.parseInt(args[i++]);
14       int c2 = Integer.parseInt(args[i++]);
15       Constraint cons1 = new XltC(x, c1);
16       Constraint cons2 = new XgtC(x, c2);
17       store.impose(cons1);
18       store.impose(cons2);
19       if (!store.consistency()) {
20         // Backtrack and add negations.
21         store.removeLevel(backtrackingLvl);
22         store.setLevel(backtrackingLvl);
23         XgteqC cons1n = new XgteqC(x, c1);
24         XlteqC cons2n = new XlteqC(x, c2);
25         Or or = new Or(cons1n, cons2n);
26         store.impose(or);
27         store.consistency();
28         break; } }
29     System.out.println(x); } }
```

Listing 9.8: Implementation of adding constraints from user input incrementally and of manual backtracking requires more effort in Java (in combination with the JaCoP solver) than in Muli.

JaCoP as a library (Listing 9.9). Again, the JaCoP implementation requires the explicit initialisation of a constraint store, which is implicit in Muli. On the one hand, JaCoP provides benefits for logic variables with small domains, as the initial domain can already be provided as part of the variable declaration. Nevertheless, the declaration of logic variables in Muli is arguably easier, given that it requires just a slight addition to a regular declaration. On the other hand, the definition of a linear constraint in JaCoP is harder to read, as the involved logic variables are defined in one array, whereas the respective weights need to be defined in a separate array. Additional complexity results from the fact that JaCoP does not allow using an arithmetic expression as the right-hand side (directly),

```
6  class JacopSendMoreMoney {
7    public static Assignment jacopSendMoreMoney() {
8      Store store = new Store();
9      IntVar s = new IntVar(store, "s", 1, 9);
10     IntVar e = new IntVar(store, "e", 0, 9);
11     IntVar n = new IntVar(store, "n", 0, 9);
12     IntVar d = new IntVar(store, "d", 0, 9);
13     IntVar m = new IntVar(store, "m", 1, 9);
14     IntVar o = new IntVar(store, "o", 0, 9);
15     IntVar r = new IntVar(store, "r", 0, 9);
16     IntVar y = new IntVar(store, "y", 0, 9);
17     // Define constraints.
18     IntVar[] vars = {s, e, n, d, m, o, r, y};
19     store.impose(new Alldifferent(vars));
20     IntVar[] eqVars = {s,     e,    n,   d,  m,     o,    r,   e,
21       m,      o,     n,    e,   y};
22     int[] eqWeights = {+1000, +100, +10, +1, +1000, +100, +10, +1,
23       -10000, -1000, -100, -10, -1};
24     Constraint equation = new LinearInt(eqVars, eqWeights, "==", 0);
25     store.impose(equation);
26     // Label variables.
27     Search<IntVar> label = new DepthFirstSearch<IntVar>();
28     SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(
29       store, vars, new IndomainMin<IntVar>());
30     label.labeling(store, select);
31     return new Assignment(s, e, n, d, m, o, r, y);
32     } }
```

Listing 9.9: Implementation the Send More Money problem in Java (in combination with the JaCoP solver) also requires more effort than in Muli.

only either a constant or a single logic variable. Consequently, we had to express the expected sum by reformulating the Send More Money equation. In contrast, the Muli representation of the constraint is more straightforward, as it facilitates defining the constraint by using Java's arithmetic and relational operators.

Moreover, constraint solving is transparent in Muli – once a free variable is sufficiently constrained so that it has a single allowed value, accessing the variable results in retrieving its value. In contrast, accessing a JaCoP variable merely retrieves the logic variable's object representation, regardless of whether it can already be simplified. Its value can only be obtained after explicitly executing search, followed by getting the value from the search result. This is similar in other solvers; for instance, Choco (cf. [PFL17]) offers

	Java LOC	Muli LOC	Δ
Send More Money	34	28	-17.6 %
Safe Lock Key	36	27	-25 %
n-Queens	30	25	-16.7 %

Table 9.2: Lines of code (LOC) required for implementing the experiments, not counting lines that are empty or comments.

a `getValue()` method that can be invoked on object representations of logic variables, but it only returns values after search has been executed (successfully). Generally, the API offered by JaCoP is similar to that of Choco, which is why we refrain from also discussing the differences between Choco and Muli in more detail here. For instance, where JaCoP requires developers to write `new IntVar(store, name, min, max)`, Choco uses the formulation `store.intVar(name, min, max)`. However, the fact the APIs are similar but not identical demonstrates the lack of standardisation. As a consequence, solver libraries are usually not interchangeable, resulting in lock-in scenarios. In contrast, Muli has an advantage in that it handles solver-specific details transparently, providing the same API regardless of the constraint solver used in the solver component.

The simpler programming style that Muli offers is also reflected in the fact that the pure Java version requires three times as many lines of code. This reduction in lines of code holds in general, because the handling of logic variables and constraint store manipulations are implicit and therefore never require any code. Furthermore, constraints can be expressed more directly using boolean expressions instead of instantiating their corresponding object representations. Muli programs can also require fewer lines of code compared to their pure Java counterparts. For the experiments used in the evaluation above, lines of code are reduced by up to a quarter (cf. Table 9.2).

Moreover, the pure Java version requires a constraint solver-specific implementation for logic variables, constraint definition, and backtracking, so that the used constraint solver cannot easily be exchanged. In contrast, the Muli program makes use of implicit backtracking and negation by the SJVM. Established Java operators and the `if` control structure are used to add constraints, as well as the primitive `int` type to declare both free and non-free variables. Last but not least, the constraint solver is exchangeable by configuring the SJVM's solver component, thus facilitating later migrations to more advanced constraint solvers.

Based on the discussed experiments, we conclude that constraint-logic OO programming is able to reduce execution times for search applications, at least in comparison to pure Java implementations. Moreover, the constraint-logic OO programming style

enables applications that interleave constraint-logic parts with imperative parts (such as requesting additional user input) during search, which is far less convenient and more error-prone to achieve without an SJVM using pure Java with a solver library.

Free variables create a little overhead at runtime, since information about them is stored in bytecode and the SJVM has to accommodate symbolic types and expressions. However, this overhead is limited to logic variables and expressions that involve them, whereas deterministic computations that do not use these concepts are not handled symbolically. Therefore, no overhead is added for non-symbolic, deterministic applications. Compared to state-of-the-art JVM implementations, e. g., OpenJDK or OpenJ9 (formerly IBM J9), we actually expect the Muli SJVM to be inferior as it does not perform any optimisations at runtime, such as just-in-time or ahead-of-time compilation. Therefore, we have not conducted any experiments to that end. Similarly, we do not expect Muli to perform better than constraint solver libraries such as JaCoP, since Muli transparently makes use of such an (interchangeable) solver library in its backend. However, those established JVM implementations lack constraint-logic programming features, while existing constraint solver libraries do not support imperative programming. Therefore, neither is able to execute any of the programs that make use of features from both paradigms. In contrast, Muli is novel in that it provides an integrated approach to the constraint-logic and object-oriented programming paradigms.

While we are certain that the idea of constraint-logic OO programming is beneficial to a range of use cases, we acknowledge that some limitations apply to our results. Our current implementation supports constraints over primitive variable types only. Nevertheless, recall that Muli already supports the use of classes (including inheritance), in addition to logic variables in (primitive) object fields. Consequently, Muli applications are not limited to using only non-OO features. Instead, objects can also be used in the formulation of constraint-logic problems. In the future, constraint-logic OO programming would benefit from support for solving constraints over object graphs or arrays. However, this is an endeavour that we will tackle in upcoming work.

9.7 Related Work

There are many libraries that add constraint programming to Java (and, therefore, to JVM languages). Choco [PFL17] and OptaPlanner [The18] are examples that seem to have gained attraction from research and industry. However, their interfaces are non-standardised, so they can be unintuitive to use and hard to exchange. The finalised JSR

331 defines a standard for constraint programming and solving in Java, but efforts seem to have ceased since 2012 [Fel12]. Furthermore, they share the disadvantage that they always work somewhat separate from the Java program that leverages them. Thus, imperative or OO code parts are not seamlessly integrated. Instead, the imperative code can only invoke the constraint solver, but it cannot intervene with the search for a solution. In contrast, Muli allows integrating search and an imperative style tightly, allowing to freely mix both paradigms in the most appropriate way to solve a given problem. Muli uses constraint solver libraries internally to check the consistency of branching constraints in order to skip infeasible paths, as well as to find specific solutions after sets of constraints have been collected. However, the solver libraries are used transparently, i. e., developers do not need to adapt Muli code to the solver they intend to use, as solver-specific encodings are created by the SJVM.

As an alternative to Muli, there are several approaches which add OO features such as inheritance to a (constraint) logic programming language, often to Prolog. For instance, Visual Prolog extends Prolog by OO features, primarily aiming at artificial intelligence applications [Sco10]. Similarly, McCabe presents an OO language based on Prolog [McC92]. Another OO layer on top of Prolog is presented by Shapiro and Takeuchi, focussing on concurrency [ST83]. Similarly, Prolog++ adds OO features to Prolog [Mos94]. tuProlog approaches the integration of Prolog and Java differently, by providing a Prolog implementation written in Java [DOR05]. This enables Prolog programs to run on the JVM, thus facilitating integrated applications without a need for the JNI as well. Yet, this results in applications implemented using two different languages, running in two separate environments on the JVM. Therefore, non-deterministic program parts are separated from imperative program parts and cannot be mixed. As an example of a non-Prolog-based language, Mozart/Oz is a constraint language that, among concurrency and lazy evaluation, also offers OO features [Van+03]. Again, this approach has a declarative focus. Compared to Muli, all these approaches have a different flavour and runtime behaviour. They are mainly declarative languages which simulate object-orientation. Assignments and state changes are not provided (natively). Although approaches adding OO features to a (constraint) logic programming language are interesting for declarative programmers, it is unlikely that they will receive much attention from mainstream OO developers due to the unfamiliar programming style.

Closest to Muli are approaches that extend OO programming with concepts from constraint-logic programming. All those that we are aware of are aimed at automated software testing. This includes glass-box test case generators, such as Muggl [MK09] and IBIS [DM03], that add symbolic execution and constraint programming to Java bytecode

execution. Pex works similarly for the .NET intermediate language by analysing programs in .NET's intermediate language [TH08]. Similar to the symbolic execution approach that this work leverages for Java, there is ongoing work for adding symbolic execution and constraint solving to the imperative programming language Rust [Ren18; LAL18]. Other work already attempted integrating logic programming into Java by extending Muggl's symbolic VM into a self-contained runtime [MK11a]. However, their approach falls short in the handling of logic variables, as only class fields can be declared free using annotations. Entire methods are declared either searching or non-searching by annotation, so defining search regions is tedious. The annotation is barely visible, thus harming effective understanding of an application.

Alma-0 is a language that integrates elements from logic programming into an imperative language [Apt+98; AS99]. It extends the Pascal-based language Modula-2 by defining several new keywords that facilitate the encoding of first-order logic formulae. In particular, keywords are introduced for expressing negation, disjunction, and existential quantification. An in-depth comparison of the language (and programs written in it) to Muli would be desirable, however, neither the Alma-0 compiler nor related resources are available anymore as development has ceased for over a decade. Nevertheless, we can point out a few conceptual similarities and differences. Programs written in Alma-0 are also executed non-deterministically. As its basis is a purely imperative language, Alma-0 data structures can only encapsulate data, whereas Muli objects also encapsulate behaviour as is usual in Java. Furthermore, non-deterministic branching occurs wherever logic variables can hold more than one value, thus labelling instantly, similar to the mechanisms of Curry and Prolog. In contrast, Muli only introduces non-determinism only when there are multiple alternatives for the control flow, collecting constraints in the process, and defers labelling of a logic variable until a specific value is required for it.

There are also several integration approaches involving further programming paradigms. Contemporary OO languages have added features that originate in functional programming, making combinations of functional programming and OO programming increasingly prominent among OO software developers. In Java, a combination of lambda abstractions and the Stream API enables development in a functional programming style where appropriate [UFM14], whereas LINQ provides similar functionality for C# [MBB06]. Also, Scala integrates functional and OO programming and also runs on the JVM [Ode+17].

Functional and constraint-logic programming have also been integrated. For example, Curry combines both paradigms using a Haskell-based syntax extended by logic variables, non-determinism, and encapsulated search [Han+95; AJ16; Bra+11; LK99]. Encapsulated search and variable definition concepts of Curry provided ideas for our constraint-logic

OO language, although the implementation of these concepts in an imperative context fundamentally differs from theirs, particularly w. r. t. the handling of side effects.

9.8 Conclusions and Future Work

Muli achieves the integration of imperative OO programming with constraint-logic programming. Although Muli is a new programming language, it preserves the Java syntax and the functionality of Java concepts outside encapsulated search, and adds non-deterministic execution inside encapsulation as formalised by the operational semantics. Moreover, programs written in Muli compile to bytecode that, in theory, could also be read by a regular JVM (even though a regular JVM would ignore Muli-specific parts of the bytecode). Instead, it is accompanied by a custom SJVM runtime environment that supports symbolic execution, encapsulated search, backtracking, and constraint solving within Muli programs. In terms of portability, our prototype is just as platform-independent as Java as a result of having chosen Java for the implementations of compiler and SJVM.

Compared to other attempts at adding constraint solving to Java, our approach is novel in that the entire runtime is capable of searching and backtracking (but only when requested explicitly, i. e. within encapsulated search). The Muli runtime environment implicitly traverses search spaces that are described by search regions in Muli programs as specified in the operational semantics, and collects solutions so that they can be re-used in later parts of the programs, including subsequent search regions. Implicit traversal gives developers the advantage to selectively mix declarative constraint definition and imperative control flows if appropriate, which is not possible with other approaches in Java.

All in all, we believe that our approach achieves a smooth integration of constraint-logic and OO paradigms, thus enabling Java developers to leverage the benefits of constraint-logic programming in a native Java style. In addition, developers are no longer required to manually invoke Java constraint solvers, or to bother with integrating external search applications, Prolog or otherwise, via JNI. This avoids a lot of programming effort and reduces potential for mistakes.

The presented operational semantics provides the basis for implementations of compiler, symbolic JVM, and tools for processing Muli programs. Our own prototypical implementations, including the classpath library, compiler, and runtime, have been made

available on GitHub as free software.[14] We welcome contributions of any kind, including reports of issues that arose or successes that you would like to share after trying out our approach.

Our work results in further novel ideas that we have yet to tackle. The degree of freedom that imperative (OO) programming already offers makes it impossible to decide for a constraint-logic OO program whether its search regions create an infinite search space. Currently, this can prevent encapsulated search from terminating, which is undesirable. Future work will attempt to work on separating producer and consumer of solutions to achieve means for interrupting encapsulated search, for providing intermediate solutions, and for continuing search.

Furthermore, future work will tackle solving for constraints involving non-primitive variables, such as solving of arrays, objects, and object graphs. Moreover, we intend to evaluate the integration of additional constraint solvers and will consider providing more convenient means to declare combinatorial constraints. Additionally, we want to extend reflection for Muli programs, thus exposing useful information about logic variables at runtime, particularly their domains, thus enabling Muli programs to inspect their own state.

Our approach would be very useful in solving such search problems that are also optimisation problems. For now, an application would first have to compute all solutions and then iterate over all solutions to find the optimum. Here, support for optimisation problems during encapsulated search would be very convenient. However, this is not trivial since branching conditions are found dynamically, so that we never know the entire optimisation problem unless we execute it. We hope to find a sophisticated solution that integrates symbolic execution with optimisation problems.

References

[AJ16] Sergio Antoy and Andy Jost. ‘A New Functional-Logic Compiler for Curry: Sprite’. In: *LOPSTR 2016*. 2016. DOI: 10.1007/978-3-319-63139-4_6.

[Apt+98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington and Andrea Schaerf. *Alma-0: An Imperative Language that Supports Declarative Programming*. 1998. DOI: 10.1145/293677.293679.

[14] https://github.com/wwu-pi/muli.

[AS99] Krzysztof R Apt and Andrea Schaerf. 'The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming'. In: *Correct System Design, Recent Insight and Advances.* Berlin, Heidelberg: Springer-Verlag, 1999, pp. 89–113.

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller and Fabian Reck. 'KiCS2: A New Compiler from Curry to Haskell'. In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: 10.1007/978-3-642-22531-4.

[DK18a] Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *SAC 2018.* ACM, 2018, pp. 1185–1194. ISBN: 978-1-4503-5191-1/18/04. DOI: 10.1145/3167132.3167260.

[DK18b] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DM03] J. Doyle and C. Meudec. 'IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming'. In: *2nd PPPJ.* 2003, pp. 55–58. DOI: 10.1145/957289.957307.

[DOR05] Enrico Denti, Andrea Omicini and Alessandro Ricci. 'Multi-paradigm Java-Prolog integration in tuProlog'. In: *Science of Computer Programming* 57.2 (2005), pp. 217–250. ISSN: 01676423. DOI: 10.1016/j.scico.2005.02.001.

[EH07] Torbjörn Ekman and Görel Hedin. 'The JastAdd extensible Java compiler'. In: *OOPSLA.* 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297105.1297029.

[EMK12] Marko Ernsting, Tim A. Majchrzak and Herbert Kuchen. 'Test Case Generation and Dynamic Mixed-Integer Linear Arithmetic Constraint Solving'. In: *21st WFLP.* 2012.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming.* Berlin Heidelberg: Springer, 2003. ISBN: 978-3-642-08712-7.

[Fel12] Jacob Feldman. *JSR 331: Constraint Programming API.* 2012. URL: https://jcp.org/en/jsr/detail?id=331.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition.* 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[Han+95] M Hanus, H Kuchen, J J Moreno-Navarro, JR Votano, M Parham and LH Hall. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Han97] Michael Hanus. 'A Unified Computation Model for Functional and Logic Programming'. In: *POPL 97*. 1997, pp. 80–93.

[KO08] Goh Kondoh and Tamiya Onodera. 'Finding bugs in Java Native Interface programs'. In: *ISSTA '08* (2008), p. 109. DOI: 10.1145/1390630.1390645.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 10844309. DOI: 10.1145/785411.785416.

[LAL18] Marcus Lindner, Jorge Aparicius and Per Lindgren. 'No Panic! Verification of Rust Programs by Symbolic Execution'. In: *International Conference on Industrial Informatics (INDIN)*. 2018, pp. 108–114. DOI: 10.1109/INDIN.2018.8471992.

[Lem+04] Christoph Lembeck, Rafael Caballero, Roger A. Müller and Herbert Kuchen. 'Constraint Solving for Generating Glass-Box Test Cases'. In: *Proceedings WFLP '04*. 2004, pp. 19–32.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[LK99] Wolfgang Lux and Herbert Kuchen. 'An Efficient Abstract Machine for Curry'. In: *Informatik '99*. Ed. by K Beiersdörfer, G Engels and W Schäfer. Springer Verlag, 1999, pp. 390–399.

[Lou93] Kenneth C Louden. *Programming Languages: Principles and Practice*. Ed. by Patricia Adams. Belmont, CA, USA: Wadsworth Publ. Co., 1993. ISBN: 0534932770.

[MBB06] Erik Meijer, Brian Beckman and Gavin Bierman. 'LINQ: Reconciling Objects, Relations and XML in the .NET Framework'. In: *ACM SIGMOD International Conference on Management of data*. 2006, p. 706. ISBN: 1595932569. DOI: 10.1145/1142473.1142552.

[McC92] F. G. McCabe. *Logic and Objects*. Prentice-Hall international series in computer science. Prentice Hall, 1992. ISBN: 9780135360798.

[MK09] Tim A. Majchrzak and Herbert Kuchen. 'Automated Test Case Generation Based on Coverage Analysis'. In: *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, July 2009, pp. 259–266. ISBN: 978-0-7695-3757-3. DOI: 10.1109/TASE.2009.33.

[MK11a] Tim A Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7.

[MK11b] Tim A. Majchrzak and Herbert Kuchen. 'Muggl: The Muenster Generator of Glass-box Test Cases'. In: *Working Papers, European Research Center for Information Systems* 10 (2011).

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/.

[PFL17] Charles Prud'homme, Jean-Guillaume Fages and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. 2017. URL: http://www.choco-solver.org.

[Ren18] David Renshaw. *Seer: Symbolic Execution Engine for Rust*. 2018. URL: https://github.com/dwrensha/seer.

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010. ISBN: 9781432749361.

[ST83] Ehud Shapiro and Akikazu Takeuchi. 'Object oriented programming in Concurrent Prolog'. In: *New Generation Computing* 1.1 (1983), pp. 25–48. ISSN: 02883635. DOI: 10.1007/BF03037020.

[Sta18] Stack Overflow. *Developer Survey Results 2018*. 2018. URL: https://insights.stackoverflow.com/survey/2018%5C#technology.

[Sta19] Stack Overflow. *Developer Survey Results 2019*. 2019. URL: https://insights.stackoverflow.com/survey/2019%5C#technology.

[TH08] Nikolai Tillmann and Jonathan de Halleux. 'Pex: White Box Test Generation for .NET'. In: *2nd International Conference on Tests and Proofs*. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9.

[The18] The OptaPlanner Team. *OptaPlanner User Guide, Version 7.11.0*. JBoss. 2018. URL: https://docs.optaplanner.org/7.11.0.Final/optaplanner-docs/html%5C_single/index.html.

[TIO19] TIOBE Software BV. *TIOBE Index for April 2019*. 2019. URL: https://www.tiobe.com/tiobe-index/.

[Tri12] Markus Triska. 'The Finite Domain Constraint Solver of SWI-Prolog'. In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316. DOI: 10.1007/978-3-642-29822-6_24.

[UFM14] Raoul-Gabriel Urma, Mario Fusco and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT: Manning Publications Co., 2014. ISBN: 9781617291999.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. DOI: 10.1017/S1471068403001741.

[VPK04] Willem Visser, Corina S. Pasareanu and Sarfraz Khurshid. *Test Input Generation with Java PathFinder*. 2004. DOI: 10.1145/1013886.1007526.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set*. Tech. rep. Menlo Park: SRI International, 1983.

10

Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming

Jan C. Dageförde* · **Herbert Kuchen***

Citation Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems*. Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review.

Abstract The Münster Logic-Imperative Language (Muli) is a constraint-logic object-oriented programming language, suited for the development of applications that interleave constraint-logic search with deterministic, imperative execution. For instance, Muli can generate graph structures of neural networks using non-deterministic search, interleaved with immediate evaluation of each generated network regarding its fitness. Furthermore, it can be used for finding solutions to planning problems. In this paper, we explain and demonstrate how these application problems are solved using Muli.

Keywords Constraint-logic object-oriented programming · artificial neural networks · planning problems · applications.

*University of Münster, Germany

10.1 Motivation

Constraint-logic object-oriented programming augments object-oriented programming with concepts and features from constraint-logic programming [DK19a]. As a result, logic variables, constraints, and non-deterministic application execution become available in an object-oriented context, facilitating the search for solutions to constraint-logic problems from an object-oriented application in an integrated way.

The **Mü**nster **L**ogic-**I**mperative language (Muli) is such a constraint-logic object-oriented language. Earlier publications on Muli focused on developing the language and its runtime environment, using artificial examples and constraint-logic puzzles for the purpose of demonstration and evaluation. With the current work, we demonstrate that Muli can be used for solving practical problems as well. We present and discuss the following application scenarios:

- The generation of graph structures for simple feed-forward neural networks designed to solve the pole balancing problem (see Section 10.3).
- Solving vehicle routing problems with dynamic constraint systems (see Section 10.4).

To start off, Section 10.2 introduces concepts of constraint-logic object-oriented programming with Muli. Concluding the paper, Section 10.5 presents related work, followed by final remarks in Section 10.6.

10.2 Constraint-Logic Object-Oriented Programming

As a rather novel paradigm, constraint-logic object-oriented programming languages feature the benefits of object-oriented programming while offering logic variables, constraints, and search, as known from constraint-logic programming. In this paper we use Muli, a constraint-logic object-oriented programming language based on Java [DK19a].

Muli uses the `free` keyword to declare variables as *logic variables*. For example,

```
int operation free;
```

declares a logic variable with an integer type. Instead of assigning a constant value to `operation`, the logic (or free) variable will be treated symbolically, unless it is sufficiently constrained such that it can be safely substituted by a constant. Logic variables can be used interchangeably with other variables of the same type, so that they can be used in the formulation of arithmetic expressions or conditions [DK18]. Furthermore, they can be passed to methods as parameters.

Logic variables are used as part of *constraints*. For simplicity, Muli does not provide a dedicated language feature for imposing constraints. Instead, a constraint is derived

from relational expressions, whenever their evaluation results in branching execution flows. As an abstract example, consider the following condition that involves the logic variable `operation`:

if (operation == 0) { s_1 } **else** { s_2 }.

Since `operation` is not sufficiently constrained, the condition can be evaluated to **true** as well as to **false** (but not both at the same time). Consequently, evaluating the condition causes the runtime environment to make a non-deterministic *choice*. From that point on, the runtime environment evaluates the available alternatives non-deterministically. When an alternative is selected, the runtime environment imposes the corresponding constraint. In the example above, when the runtime environment selects s_1 for further evaluation it imposes `operation` == 0, ensuring that later evaluations cannot violate the assumption that is made regarding the value of `operation`.

Search problems are specified in Muli in methods that are termed *search regions* for they will be executed non-deterministically. Consider a problem that looks for the smallest integer e that can be expressed in two different ways as the sum of two positive integer cubes (the Hardy-Ramanujan number, namely, 1729). The corresponding constraint is:

$$\begin{aligned} & e = a^3 + b^3 = c^3 + d^3 \\ & \wedge a \neq c \wedge a \neq d \\ & \wedge a, b, c, d, e \in \mathcal{N} - \{0\} \end{aligned}$$

A Muli search region that calculates e using this constraint is implemented by the method `solve()` as depicted in Listing 10.1, assuming that there is a method `cube`$(n) = n^3$ and another method `positiveDomain`$(x_1, \ldots, x_n)$ that imposes the constraint $x_i \in \mathcal{N} - \{0\}\ \forall 1 \leq i \leq n$.

```
1  class Taxicab {
2    int solve() {
3      int a free, b free, c free, d free, e free;
4      positiveDomain(a, b, c, d, e);
5      if (a != c && a != d &&
6          cube(a) + cube(b) == e &&
7          cube(c) + cube(d) == e) {
8        return e; }
9      else throw Muli.fail(); } }
```

Listing 10.1: Muli search region that calculates the Hardy-Ramanujan number.

The runtime environment realizes *search* transparently: It takes non-deterministic decisions at choices. Once a solution has been found, the runtime environment backtracks until the most recent choice is found that offers an alternative decision. Afterwards, it takes that alternative decision and continues execution accordingly. In the backend, the runtime environment leverages a *constraint solver* for finding appropriate values for logic variables that satisfy all imposed constraints. Furthermore, when a branch of a choice is selected, the solver checks whether the current constraint system has become inconsistent in order to cut off infeasible execution branches early. Found solutions are collected by the runtime environment and made available to the invoking application. Conceptually, following a sequence of decisions at choices, in combination with backtracking to take different decisions, produces a search tree that represents execution paths. In such a search tree, inner nodes are the choices whereas the leaves represent ends of alternative execution paths [DT20]. Execution paths in Muli end with a *solution*, e. g., a return value, or with a *failure*, e. g., if an execution path's constraint system is inconsistent. Moreover, applications sometimes require an *explicit failure* denoting the end of an execution path without a solution. An explicit failure is expressed by `throw Muli.fail()`, which is specifically interpreted by the runtime environment to end search and backtrack to the next alternative.

In Muli, execution of the main program is deterministic. In contrast, all non-deterministic search is *encapsulated*, thus giving application developers control over search. `Muli.muli()` accepts a search region, i. e. either a lambda expression or a reference to a method, and returns a stream of `Solution` objects. The search region that is passed to `Muli.muli()` is the method that will be executed non-deterministically. For instance, search for the Hardy-Ramanujan number from the example in Listing 10.1 is started with

```
Stream<Solution> solutions = Muli.muli(Taxicab::solve);,
```

thus offering a stream of solutions that can be consumed from the `solutions` variable. Muli uses the Java Stream API in order to evaluate solutions non-strictly, thus allowing applications to assess a returned solution individually before continuing search to obtain additional solutions [DK19b]. This is made possible with the help of an adaptation of the trail structure of the Warren Abstract Machine (WAM) [War83]. In contrast to the WAM trail, the Muli trail records changes to all elements of the execution state in order to be able to revert them. Furthermore, Muli features an inverse trail (or forward trail) that is leveraged when search at a specific point is resumed, i. e., when the consumer of the stream queries another element.

10.3 Generation of Graph Structures for Neural Networks

A current research trend in artificial neural networks (ANN) is that not only the weights of the inputs of each neuron are corrected via back-propagation, but also the structure of the network is adapted [PC04]. Thus, the goal is to find the smallest ANN producing an acceptable output quality. A application implemented in Muli can generate structures of directed acyclic graphs that define an ANN. In this section, we implement the application `NNGenerator` that demonstrates how the non-deterministic evaluation of Muli search regions can be used to systematically generate a set of feed-forward ANNs.Each generated ANN is then trained against a specific problem; in our case balancing a single pole on a moving cart as illustrated in Figure 10.1 [BSA83]. Every time that a network is generated, `NNGenerator` assesses the network's fitness in order to decide whether its output quality is acceptable, and continues the search for better ANN graph structures otherwise.

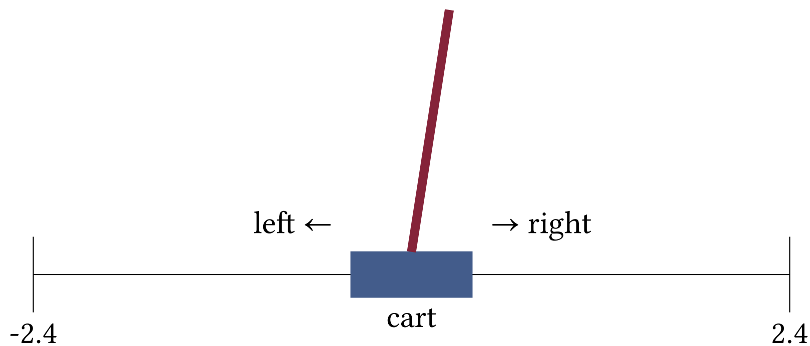
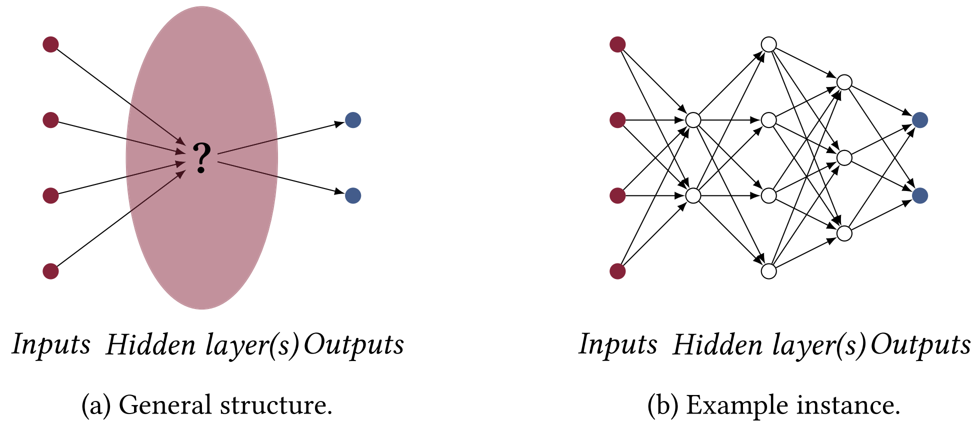


Figure 10.1: The pole balancing problem as simulated by the CartPole-v1 implementation from OpenAI.

For the generated ANNs we use Python, because PyTorch [Pas+19] is a powerful Python-based library for the implementation of ANNs and because the OpenAI Gym collection of reinforcement learning tasks [Ope20] provides a simulation environment for the pole balancing problem, namely *CartPole-v1*, implemented in Python. Moreover, this provides us with the opportunity to demonstrate that Muli applications can integrate applications written in other programming languages as well.

The *CartPole-v1* simulation provides a so-called environment that our application will interact with. As long as the pole is in balance, the environment accepts one of two actions, *left* and *right* (as illustrated in Figure 10.1), that move the pole-balancing *cart* into a specific direction. As a result of an action, the environment updates its state and returns four observations:



(a) General structure.

(b) Example instance.

Figure 10.2: Feed-forward neural networks for solving the pole balancing problem.

- The position of the cart $\in [-2.4,\ 2.4]$,
- the velocity of the cart $\in [-\infty,\ \infty]$,
- the angle of the pole, and $\in [-41.8\,°,\ 41.8\,°]$, and
- the velocity of the tip of the pole $\in [-\infty,\ \infty]$.

These observations can be used to make a decision about a subsequent action. We generate feed-forward neural networks with reinforcement learning. Of these networks, two parameters are fixed: The input layer contains four nodes, one per observation, whereas the output layer contains two nodes, namely the probability of selecting the *left* action and that of selecting the *right* action, accordingly. The next step is decided by comparing the output nodes and choosing the action with the highest probability. The step is then passed to the environment. The structure of the hidden layer(s) is not fixed and will be generated by `NNGenerator`. The general structure of the intended ANNs is illustrated in Figure 10.2a, and a concrete instance is exemplarily given in Figure 10.2b.

We first describe the Muli application `NNGenerator` that generates graph structures for the hidden layers of ANNs, followed by a subsection with details on the neural network implementation using PyTorch. Afterwards, we experiment with `NNGenerator` and present the results.

10.3.1 Generating Neural Network Graph Structures from a Muli Application

The goal of the `NNGenerator` Muli application is to search for directed acyclic graphs that will constitute the hidden layers of the final neural networks, with the aim of producing the simplest graph structure. The simplest network has no hidden layer; i. e., all input

nodes are directly connected to all output nodes. Starting from the simplest network, two operations that modify the graph are possible:

1. `AddLayer` Add a hidden layer (with one node as a starting point), or
2. `AddNode` add a node to one of the existing hidden layers.

We can implement `NNGenerator` as a Muli search region that enumerates graphs by non-deterministically choosing one of these operations and, for the `AddNode` operation, by non-deterministically selecting one of the existing layers. As an implementation detail we add a third operation, `Return`, that returns the current graph structure as a solution. The other two operations recursively invoke the generation method in order to select the next operation. Listing 10.2 shows the recursive implementation of the generation method and exhibits the use of the free variable `int operation free` in conditions, thus implementing the non-deterministic choice for one of the three operations, as well as `int toLayer free` for selecting a layer in the `AddNode` case.

As the search region of Listing 10.2 does not have a termination criterion, an infinite number of solutions is found (i. e., infinitely many graphs with all numbers and sizes of layers). Returning all of them in a fixed array is impossible. However, Muli offers an encapsulated search operator that delivers solutions lazily and returns immediately after a solution has been found, while maintaining state such that search can be resumed for additional solutions on demand [DK19b]. For our application, the operator is invoked as

```
Stream<Solution<Network>> solutions = Muli.muli(NNGenerator::generateNetwork);
```

As a result, individual solutions can be obtained from the `solutions` stream.

Another caveat of the application is the selected search strategy. Even though the Muli runtime environment takes non-deterministic decisions at choices, the decisions are not random. Instead, it will systematically traverse the choices of the search region. With a depth-first search strategy, this means that the generated graphs are probably bad solutions: First, a graph with no hidden layers; second, a graph with one hidden layer with a single node; third, a graph with two hidden layers and one node each, and so on. Under a depth-first search assumption and with the presented search region, there would never be layers with more than one node except for the input and output layers. Rewriting the search region does not help either, as that would only generate graphs with a single layer and an ever-increasing number of nodes on that layer. As a remedy, Muli offers the well-known iterative deepening depth-first search (IDDFS) strategy [DT20] ensuring that every number of layers and every size of each layer can eventually be considered. In order to use IDDFS we have to slightly modify the encapsulated search operator call:

```
1  Network generateNetwork() {
2    return generateNetwork( new Network(4, 2) ); }
3
4  Network generateNetwork(Network network) {
5    int operation free;
6    switch (operation) {
7      case 0: // Return current network.
8        return network;
9      case 1: // Add layer.
10       network.addLayer();
11       return generateNetwork(network);
12     default: // Add node. But where?
13       if (network.numberOfLayers > 0) {
14         int toLayer free;
15         for (int layer = 0; layer < network.numberOfLayers;
16                                layer++) {
17           if (layer == toLayer) {
18             network.addNode(layer);
19             return generateNetwork(network);
20           } else {
21             // Add at a different layer!
22           } }
23         throw Muli.fail();
24       } else {
25         throw Muli.fail(); } } }
```

Listing 10.2: Muli search region that systematically generates graph structures by non-deterministic selection of operations.

```
Stream<Solution<Network>> solutions = Muli.muli(NNGenerator::generateNetwork,
                SearchStrategy.IterativeDeepening);
```

Listing 10.3 shows how the solution stream is used. The `forEach` consumer demands and obtains individual solutions from the stream. `n.toPyCode()` creates Python code that implements an ANN according to the generated graph (for details on what the code looks like see Subsection 10.3.2), and the helper method `writeAndRun()` writes the generated code into a `.py` script. Afterwards, the script is run via `Runtime.getRuntime().exec()`. We assume that the generated Python application prints the network's fitness after training and use to standard output, so that output is captured and stored in the `fitness` variable. In Listing 10.3 we consider a solution "good enough" (thus ending search) if its cumulative fitness value is greater than 400.

```
solutions.forEach(solution -> {
  Network n = solution.value;
    // Execute python script.
    String fitness = NNGenerator.writeAndRun(n.toPyCode());
    // Quit if a working neural network is found.
    if (Float.parseFloat(fitness) > 400) {
      System.out.println(n.toString());
      System.exit(0); } });
```

Listing 10.3: Processing the solution stream in Muli.

10.3.2 Using Generated Neural Networks to Solve the Pole Balancing Problem

In our feed-forward ANNs we assume that all layers are linear ones. In addition to that, between every layer we use dropout [Sri+14] to randomly cancel the effect of some nodes with a probability of 0.6, in combination with a rectified linear unit activation function ensuring that values are positive [HSS15]. Finally, the output layer values are rescaled using the softmax activation function, ensuring that each output is in the range $[0, 1]$ and that the sum of the two outputs is 1. Initially, the edge weights assume the default values provided by PyTorch for `nn.Linear` layers. Afterwards, the network is trained in order to learn weights such that the network can balance the pole for as long as possible. To that end, we use the Adam optimizer [KB15] with a learning rate of 0.01 and train the network using a monte-carlo policy gradient method for 500 episodes, each for a maximum of 500 steps. We process an entire episode and learn new weights based on the rewards obtained in throughout that episode, before continuing with the next episode.

The `toPyCode()` method of `NNGenerator` will generate Python code that implements ANNs according to the above specification of the network and to the structure that was generated. In the end, we do not want to generate full implementations of ANNs for every found graph. After all, major parts of the resulting programs are static and could therefore be implemented once and then be used as a library by all generated networks. We implement a Python class `ENN` that implements the ANN itself using PyTorch, and we provide two methods `train()` and `use()` that each accept an instance of `ENN` in order to work with it. The Muli application `NNGenerator` can generate small Python programs that import `ENN`, `train()`, and `use()`. The generated programs then instantiate the `ENN` class according to the parameters found by Muli and use the provided methods. Listing 10.4 provides an example of the code that is generated from the `NNGenerator` application,

demonstrating that implementation details of the ANN are abstracted away into the library. Subsequently, we provide more details about the class and the methods.

```
net = ENN(ins = 4, hidden = [100, 50], out = 2)
train(net)
fitness = use(net)
print(fitness)
```

Listing 10.4: Structure of the Python program as generated by the `NNGenerator` Muli application. Note that the constructor parameters of `ENN` are shown exemplarily; they need to be substituted according to a specific configuration.

In its constructor, the `ENN` class accepts three parameters: The number of input nodes, an ordered list containing the numbers of nodes on the inner layers, and the number of output nodes. For instance, the network illustrated in Figure 10.2b is instantiated by invoking `ENN(ins = 4, hidden = [2, 4, 3], outs = 2)`. Since the number of inner layers and the number of nodes on each layer is expressed as an array, `ENN` is able to construct an ANN with arbitrary hidden layers, allowing `NNGenerator` to specify the hidden layer. Listing 10.5 demonstrates how the constructor parameters, and the list of hidden layers in particular, are used to represent the network. In Listing 10.5, the `forward()` method specifies the sequential model, inserting the additional layers as described above.

The `train()` method accepts the `ENN` instance and creates an Open AI Gym environment using the CartPole implementation. It then starts a training loop with 500 episodes. At the beginning of every episode, the environment is reset to an initial state. An episode ends either when the pole is out of balance, or when the maximum of 500 steps is reached. As soon as an episode ends, the network weights are learned according to the description above, thus preparing the network for the next episode.

The trained network is passed to the `use()` method that creates a new OpenAI Gym environment and performs a single simulation of the pole balancing problem, up to a maximum of 500 steps. In order to allow `NNGenerator` to judge the quality of a final, i. e., generated and trained, network, we define a fitness function based on the position of the cart that is applied after every step and summed over all steps that the pole is in balance:

$$f(position) = -0.1736 * position^2 + 1$$

$f(position)$ is 1 when the cart is at the centre and decreases to 0 when the cart is nearing one of the edges at -2.4 or 2.4. As a consequence, solutions that keep the pole near the centre, with just minor movement, are favoured. A perfect solution would keep the

```
class ENNPolicy(nn.Module):
  def __init__(self, ins, hidden, outs):
    # < some initialization omitted >
    lastnodes = ins
    for nodes in hidden:
      newlayer = nn.Linear(lastnodes, nodes, bias=False)
      self.layers.append(newlayer)
      lastnodes = nodes
    # Final layer:
    newlayer = nn.Linear(lastnodes, outs, bias=False)
    self.layers.append(newlayer)
    self.layerout = newlayer
  def forward(self, x):
    args = []
    for layer in self.layers[:-1]:
      args.append(layer)
      args.append(nn.Dropout(p=0.6))
      args.append(nn.ReLU())
    args.append(self.layers[-1])
    args.append(nn.Softmax(dim=-1))
    model = torch.nn.Sequential(*args)
    return model(x)
```

Listing 10.5: Python class `ENN` that creates hidden layers dynamically from the constructor parameters.

pole balanced for all 500 steps and $\sum_{i=1}^{500} f(position_i)$ is approximately 500. We augment the `use()` method to record the fitness values throughout all steps and to return the cumulative fitness value. The last two lines of Listing 10.4 demonstrate how the sum is printed to the standard output, so that it can be read and judged by `NNGenerator`.

10.3.3 Experiments

We conduct two experiments with `NNGenerator` in order to evaluate Muli's ability to generate directed acyclic graphs. In the first experiment we are interested in the smallest ANN that is able to solve the pole balancing problem, i. e., whose cumulative fitness is greater than 400. Incidentally, the structures that were generated until finding an adequate ANN all only have a single hidden layer. The smallest network capable of solving the problem has just ten nodes on a single hidden layer (Table 10.1). The generation of the first network takes longer than that of the other, larger ones. This can be attributed to the

Nodes on first layer	Generation time [ms]	Training time [s]	Fitness	Solved
1	115.712	6.898	14.987	no
2	14.832	8.402	9.992	no
3	2.446	8.117	76.888	no
4	2.350	10.662	14.987	no
5	2.426	15.710	125.005	no
6	1.930	18.086	75.889	no
7	1.991	18.825	125.491	no
8	2.488	38.483	52.977	no
9	2.351	17.703	371.425	no
10	2.240	50.963	499.623	**yes**

Table 10.1: Graph structures generated before the smallest neural network that solves the problem is found. For each network, the time spent on its generation (in milliseconds) and training (in seconds) are indicated as well as its fitness.

just-in-time compilation of the JVM that increases the speed of generating subsequent solutions. Moreover, it was also the first network to be generated at all, so that the generation time includes some initialization effort for the virtual machine and the search region. In contrast, subsequent graphs are created by local backtracking and/or by applying minor modification operations, so generating those is quicker.

In the second experiment we are interested in the ability to generate larger hidden layers. To that end, we multiply the number of nodes added in the `AddNode` step by 50. Moreover, we switch the order in which IDDFS takes decisions, thus favouring larger networks over smaller ones first. The first generated ANN is already able to solve the pole balancing problem. Therefore, execution could already be stopped after that according to the termination criterion in Listing 10.3. However, we are curious about additional solutions, so we remove that criterion. Table 10.2 exemplarily shows the first 15 generated networks that were able to solve the problem, i. e., whose cumulative fitness is greater than 400 each. In fact, all these networks exhibit a value of over 483, and most of them are able to reach a cumulative fitness greater than 499. Not shown in Table 10.2 are networks that are unable to solve the pole balancing problem. As an additional finding, both experiments indicate that the generation of graph structures with `NNGenerator` is faster than training the ANNs afterwards. This is expected since the structural modifications between two graph structures are minor, whereas each generated ANN has to be trained from scratch.

Hidden layers	Generation time [ms]	Training time [s]	Fitness
[400]	155.547	80.583	499.952
[350, 50]	0.044	35.114	499.390
[300]	0.060	94.019	499.653
[300, 100]	2.224	47.930	499.899
[250]	0.040	77.497	499.704
[250, 150]	1.474	126.405	499.569
[250, 50]	0.031	52.727	499.508
[200, 150]	0.042	98.520	499.298
[200, 100]	0.027	112.001	498.324
[200, 50, 100]	0.036	115.636	483.321
[150]	0.025	76.728	499.223
[150, 200]	0.025	73.006	498.517
[150, 150, 50]	0.028	80.517	498.564
[150, 50]	0.028	87.862	499.916
[100, 100, 50, 100]	0.036	101.163	499.540

Table 10.2: Times spent on generating (in milliseconds) and training (in seconds) the first 15 generated large neural networks that were able to solve the problem.

10.4 Solving a Dynamic Scheduling Problem with Constraint-Logic Object-Oriented Programming

Another application which can benefit from interleaved deterministic object-oriented computation and non-deterministic search can be found in logistics. Imagine a logistics company which runs a large number of trucks carrying goods from various sources to destinations. New orders arrive on the fly while trucks are running. Each order has a quantity of a certain good (that has a specific size and weight), a source location, a destination, an earliest and latest pick-up time, a latest delivery time, and so on. Moreover, the trucks have a maximum capacity w. r. t. volume and weight of goods that are transported at the same time. Consequently, the current set of orders imposes a set of constraints. The current schedule is based on a solution that satisfies these constraints and, optionally, on an optimal solution that maximizes the revenues of the accepted orders.

The described problem is transferred into a class structure as illustrated in Figure 10.3. Dispatching new orders to trucks results in additional constraints regarding size and weight, ensuring that trucks are not over capacity after scheduling. For an array of trucks, the constraints can be formulated with Muli using the code presented in Listing 10.6. Capacity violations result in an explicit failure, whereas a successful dispatch of all goods will result in the return value `true` and Truck-Order relationships are set via `addOrder()`

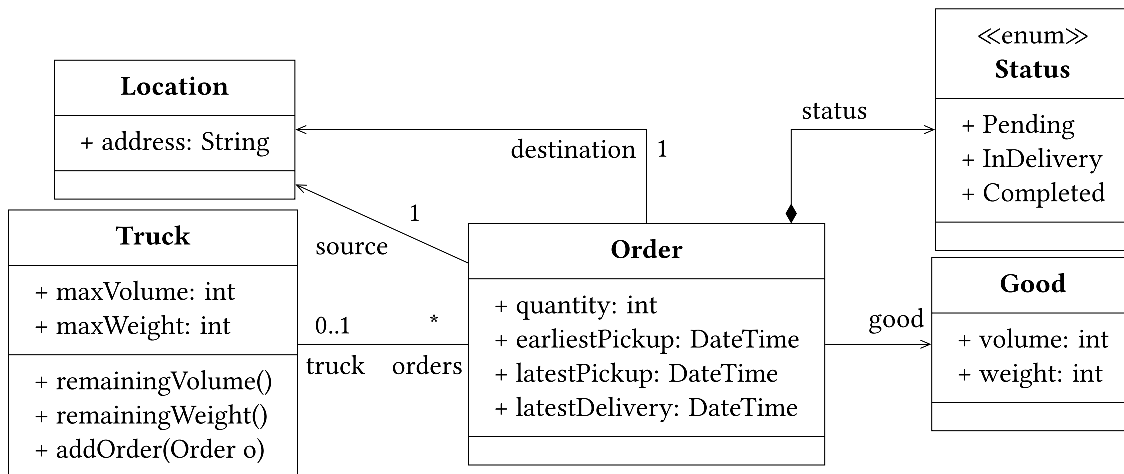


Figure 10.3: Class structure that models our logistics planning problem.

accordingly. Constraints w. r. t. location and pickup/delivery timing are formulated analogously.

The solution can be found by non-deterministic search and constraint solving, e. g., using the `Muli.muli()` encapsulated search operator. However, the encapsulated search does not only deliver a solution. It also delivers a representation of the search space for potential later use. After the solution has been found, deterministic computations are required for instance for keeping track of the current positions of the trucks and for communicating the determined schedule with truck drivers.

As soon as new orders arrive, a new solution of the now extended scheduling problem has to be found. As a consequence, this is a dynamic problem in which the entire set of constraints is not known prior to the start of an application. Instead, the set of constraints develops over time. Now, the additional constraints caused by the new order can be added to the saved representation of the search space and a new encapsulated search can be started producing a new solution (and a new representation of the search space). The possibility to continue search based on the previous solution facilitates faster search, as opposed to solving the constraint problem from scratch.

10.5 Related Work

There are several approaches that extend object-oriented (OO) programming or imperative programming with concepts from constraint-logic programming, see e. g. [DM03; MK11; Ren18; TH08]. The integration of the two paradigms that these approaches achieve is not as smooth as the integration provided by Muli. Typically, these approaches show a clear

```
boolean dispatch(Truck[] trucks, Order[] newOrders) {
  for (Order o : newOrders) {
    int selection free;
    selection = domain(truck, 0, trucks.length-1);
    int orderWeight = o.quantity * o.good.weight;
    int orderVolume = o.quantity * o.good.volume;
    for (int truck = 0; truck < trucks.length; truck++) {
      if (truck == selection) {
        if (orderWeight <= trucks[truck].remainingWeight() &&
            orderVolume <= trucks[truck].remainingVolume()) {
          trucks[truck].addOrder(o);
        } else {
          throw Muli.fail(); } } } }
  // All orders dispatched; good to go.
  return true;
}
```

Listing 10.6: Muli code snippet to dispatch orders to trucks with non-deterministic search, modelling weight and volume constraints.

syntactic and semantic separation of the imperative and object-oriented part. Moreover, none of these approaches provides encapsulated search.

An alternative to using Muli is to just call a constraint solver from an OO language, such as JaCoP or Choco from Java [Kuc03; PFL17]. However, this also does not lead to a seamless integration of both paradigms. In particular, alternating deterministic OO computations and non-deterministic search are more cumbersome.

There are also approaches adding object-orientation to (constraint) logic languages [McC92; Mos94; Sco10; ST83; Van+03]. However here, the object orientation is just syntactic sugar and constraint-logic features are used to simulate the object orientation. This typically causes some performance penalty compared to pure OO languages. Also, these languages keep the declarative flavour and do not provide assignments. Thus, they will hardly be considered by object-oriented programmers, whereas Muli is very close to Java and hence easier to use for developers who are used to object-oriented languages.

The general idea of Muli's encapsulated search was taken from the functional-logic programming language Curry [AJ16; Bra+11; HKM95; LK99]. However, in contrast to Curry, our encapsulated search can deal with side-effects, which causes the implementation to be quite different.

10.6 Conclusion and Outlook

With the present work we use the Muli programming language for the development of applications that solve practical search problems. As the first example, the `NNGenerator` application leverages non-deterministic execution for the systematic generation of directed acyclic graphs that are used to describe the structure of PyTorch-based ANNs. The networks generated by `NNGenerator` solve the pole balancing problem; this problem can be substituted for different ones as the ANNs are problem-agnostic. Moreover, `NNGenerator` runs and evaluates each generated network, judging whether one of them is good enough or whether to proceed search in order to find additional networks. As the second example, we discuss how to apply Muli to a scheduling problem from logistics, demonstrating how to model constraints in an constraint-logic object-oriented way.

The presented applications demonstrate the practical applicability of Muli. Compiler and runtime environment are publicly available as open source software on GitHub,[15] inviting others to use Muli in research or for their practical applications. Future work will use Muli for further planning problems, refining the language and the runtime environment in the process.

References

[AJ16] Sergio Antoy and Andy Jost. ‘A New Functional-Logic Compiler for Curry: Sprite’. In: *LOPSTR 2016*. 2016. DOI: 10.1007/978-3-319-63139-4_6.

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller and Fabian Reck. ‘KiCS2: A New Compiler from Curry to Haskell’. In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: 10.1007/978-3-642-22531-4.

[BSA83] Andrew G. Barto, Richard S. Sutton and Charles W. Anderson. ‘Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems’. In: *IEEE Transactions on Systems, Man and Cybernetics* SMC-13.5 (1983), pp. 834–846. ISSN: 21682909. DOI: 10.1109/TSMC.1983.6313077.

[DK18] Jan C. Dageförde and Herbert Kuchen. ‘An Operational Semantics for Constraint-Logic Imperative Programming’. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[15] https://github.com/wwu-pi/muli.

[DK19a] Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

[DK19b] Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing*. Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912.

[DM03] J. Doyle and C. Meudec. 'IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming'. In: *2nd PPPJ*. 2003, pp. 55–58. DOI: 10.1145/957289.957307.

[DT20] Jan C Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. Springer, 2020. DOI: 10.1007/978-3-030-46714-2_13.

[HKM95] Michael Hanus, Herbert Kuchen and Juan Jose Moreno-Navarro. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[HSS15] Kazuyuki Hara, Daisuke Saito and Hayaru Shouno. 'Analysis of function of rectified linear unit used in deep learning'. In: *2015 International Joint Conference on Neural Networks*. 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280578.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. 'Adam: A method for stochastic optimization'. In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (2015), pp. 1–15. URL: https://arxiv.org/abs/1412.6980.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 10844309. DOI: 10.1145/785411.785416.

[LK99] Wolfgang Lux and Herbert Kuchen. 'An Efficient Abstract Machine for Curry'. In: *Informatik '99*. Ed. by K Beiersdörfer, G Engels and W Schäfer. Springer Verlag, 1999, pp. 390–399.

[McC92] F. G. McCabe. *Logic and Objects*. Prentice-Hall international series in computer science. Prentice Hall, 1992. ISBN: 9780135360798.

[MK11] Tim A Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7. DOI: 10.1007/978-3-642-22531-4_8.

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[Ope20] OpenAI. *CartPole-v1*. 2020. URL: https://gym.openai.com/envs/CartPole-v1/ (visited on 18/03/2020).

[Pas+19] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H Wallach, H Larochelle, A Beygelzimer, F d'Alché-Buc, E Fox and R Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[PC04] R M Palnitkar and J Cannady. 'A Review of Adaptive Neural Networks'. In: *IEEE SoutheastCon, 2004. Proceedings*. 2004, pp. 38–47. DOI: 10.1109/SECON.2004.1287896.

[PFL17] Charles Prud'homme, Jean-Guillaume Fages and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. 2017. URL: http://www.choco-solver.org.

[Ren18] David Renshaw. *Seer: Symbolic Execution Engine for Rust*. 2018. URL: https://github.com/dwrensha/seer (visited on 03/05/2019).

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010. ISBN: 9781432749361.

[Sri+14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. 'Dropout: a simple way to prevent neural networks from overfitting'. In: *J. Mach. Learn. Res.* 15 (2014), pp. 1929–1958.

[ST83] Ehud Shapiro and Akikazu Takeuchi. 'Object oriented programming in Concurrent Prolog'. In: *New Generation Computing* 1.1 (1983), pp. 25–48. ISSN: 02883635. DOI: 10.1007/BF03037020.

[TH08] Nikolai Tillmann and Jonathan de Halleux. 'Pex: White Box Test Generation for .NET'. In: *2nd International Conference on Tests and Proofs*. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. DOI: 10.1017/S1471068403001741.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set.* Tech. rep. Menlo Park: SRI International, 1983.

11

Free Objects in Constraint-Logic Object-Oriented Programming

Jan C. Dageförde[*] · Herbert Kuchen[*]

Citation Jan C. Dageförde and Herbert Kuchen. 'Free Objects in Constraint-logic Object-oriented Programming'. In: *Proceedings of the ACM on Programming Languages (OOPSLA)*. 2020. Under review.

Abstract Constraint-logic object-oriented programming is useful in the integrated development of business software that occasionally solves constraint-logic problems. So far, work in constraint-logic object-oriented programming was limited to considering constraints that only involve logic variables of primitive types; in particular, boolean, integer, and floating-point numbers. However, the availability of object-oriented features calls for the option to use logic variables in lieu of objects as well. Therefore, support for reference-type logic variables (or *free objects*) is required. With the present work, we add support for reference-type logic variables to a Java-based constraint-logic object-oriented language. Allowing free objects in statements and expressions results in novel interactions with objects at runtime, for instance, non-deterministic execution of invocations on free objects (taking arbitrary class hierarchies and overriding into account). In order to achieve this, we also propose a dynamic type constraint that restricts the types of free objects at runtime.

Keywords Constraint-logic object-oriented programming · reference-type logic variables · programming language implementation · runtime systems.

[*]University of Münster, Germany

11.1 Programming with Free Objects

With *constraint-logic object-oriented programming (CLOOP)*, software engineers are offered an integrated programming paradigm for the development of business software that occasionally requires search for solutions to constraint-logic problems. As a mixed paradigm, CLOOP provides well-known benefits of object-oriented programming languages (e. g., encapsulation of data and behaviour) as well as of constraint-logic programming (declarative specification and solving of search problems). For example, the CLOOP language *Muli* is based on Java and adds logic variables, symbolic execution, constraints, and encapsulated search. Muli code is executed by the Muli Logic Virtual Machine (MLVM), which is a custom Java virtual machine that also provides support for symbolic execution and constraint solving [DK19]. Syntactically, logic variables in Muli can be of arbitrary types. However, so far, constraints can only be defined over logic variables of *primitive* types. Primitive logic variables may still be assigned to fields of objects, so they can already be used in an object-oriented context. Nevertheless, adding support for logic variables that represent entire objects requires additional work.

Adding support for reference-type logic variables (*free objects* in particular) raises interesting questions. After all, objects in object-oriented languages (and, therefore, also in CLOOP) encapsulate data *and behaviour*. For instance, consider the following code from Listing 11.1 in the context of the class hierarchy illustrated in Figure 11.1, which will serve as a running example.

```
1 Shape s free;
2 if (s.getArea() == 16) {
3   System.out.println(s.toString()); }
```

Listing 11.1: Excerpt from a constraint-logic object-oriented program that invokes a method on a free object.

As `Shape` merely provides the interface, the invocation of `s.getArea()` can be interpreted in multiple ways depending on the number of implementations of `Shape`. Like in this example, we generally assume that the type of a free object is only partially known, i. e., when a variable that is declared as `Object o` is of type `Object`, `o` may in fact hold an instance of `Object` or of any subtype. Consequently, there is only partial information about the (actual) type of an object, so that there are implications for

- how free objects are instantiated,
- how accesses to fields of a free object are handled,

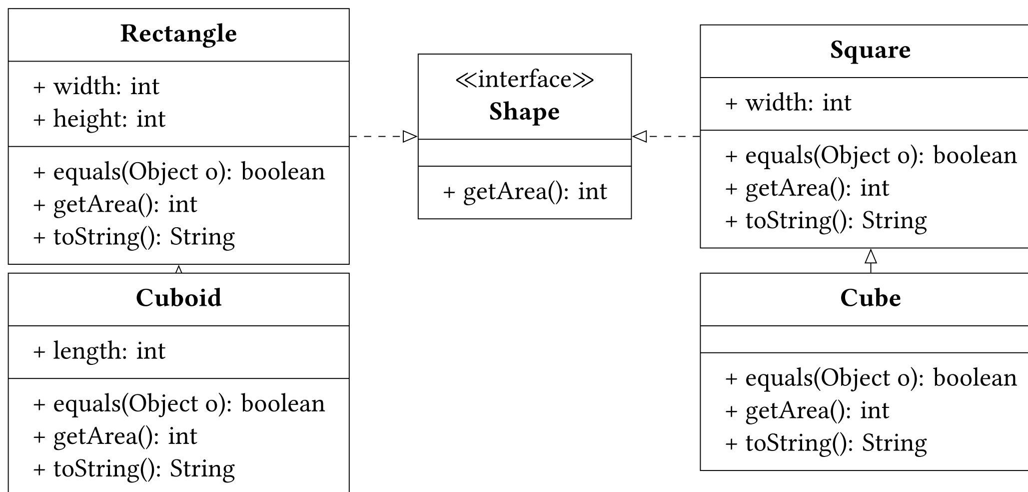


Figure 11.1: Class structure assumed for the running example.

- invocations of methods on free objects,
- type operations on free objects, and
- the notion of equality of free (and regular) objects.

Some of these implications have been discussed conceptually in a research-in-progress paper, but the discussion was incomplete and did not yet result in an implementation [Dag19].

In order to effectively realize the benefits of an integrated CLOOP language, these implications need to be discussed in order to define the semantics and to implement a runtime environment. In CLOOP, we expect the applicable alternatives to be evaluated non-deterministically until all alternatives are considered ("don't know" non-determinism) [DK19]. Therefore, the example in Listing 11.1 would result in at least four lines of output as there are four classes that implement the `Shape` interface.

This paper provides the following contributions to CLOOP, all of which we have exemplarily implemented in a modified MLVM:

- A semantics for non-deterministic method invocations on free objects, i. e., on objects whose type is only partially known (Section 11.3). This is achieved in combination with a dynamic type constraint that restricts the valid types of an object at runtime.
- A discussion of how fields of free objects are accessed (Section 11.4).
- Another application of the dynamic type constraint for the implementation of type operations, namely casts and checks in Subsection 11.5.1, followed by a discussion of equality of (free) objects (Subsection 11.5.2).

- Throughout the paper we use several examples that demonstrate how free objects are useful in programming. In addition, we display larger example applications for demonstration purposes in Section 11.6.

Moreover, related research is outlined in Section 11.7. Finally, Section 11.8 summarizes the contribution and provides an outlook. Subsequently, we start by giving a short introduction to Muli in Section 11.2, followed by a description of preliminaries of free objects (Subsection 11.2.1). All example programs presented in this paper can be compiled with the Muli compiler and executed on our modified MLVM.

11.2 Constraint-Logic Object-Oriented Programming with Muli

We base our work on the ***M**ünster **L**ogic-**I**mperative Language (Muli)*.[16] Muli is a CLOOP language whose syntax and semantics are based on those of Java 8 [DK19]. The Muli Logic Virtual Machine (MLVM) is a modification of a JVMS-compliant [Lin+15] Java virtual machine and serves as the runtime environment. We briefly introduce the main features of the Muli programming language.

In Muli, an unbound ("free") variable is declared using the `free` keyword, e. g.,

```
int width free;.
```

At runtime, free variables are treated as logic variables to be used in symbolic expressions. Free objects are declared analogously, but prior to our work their semantics was undefined and the MLVM did not provide an implementation for treating free objects yet. Therefore, the following code was able to compile but the method invocation in the second line failed:

```
Shape s free;
s.getArea();.
```

This issue is tackled in the present paper, adding full support for logic variables that represent objects.

In Muli, the way all (logic and regular) variables are used in boolean or arithmetic expressions is identical to Java. However, an expression that contains unbound variables cannot be evaluated to a constant. Therefore, the MLVM treats those variables symbolic-

[16] https://github.com/wwu-pi/muli.

ally and creates a symbolic expression [DK18].[17] To give an example, after executing the Muli program in Listing 11.2, `five` holds the constant value `5`, whereas `symbolic` holds the symbolic expression `x + 5`.

```
1 int x free;
2 int two = 2, three = 3;
3 int five = three + two;
4 int symbolic = x + five;
```

Listing 11.2: Example that demonstrates symbolic evaluation of expressions that contain logic variables.

Ultimately, symbolic arithmetic expressions can evaluate to numeric constants (e. g., after substituting all contained symbolic variables by appropriate constants). For instance, an arithmetic expression that contains only **int** (logic) variables and **int** constants can be used anywhere where an **int** expression is expected. Therefore, symbolic expressions can be passed as parameter values, assigned to variables, or used as the return value of a method. A symbolic expression is preserved until all comprised logic variables can be substituted by a constant, either via sufficiently specific constraints or via labelling.

The behaviour described so far is deterministic. However, as soon as a symbolic expression is used as part of a condition that leads to branching (e. g., in an **if** statement), it is possible that the execution environment cannot decide on a unique outcome because, given appropriate constraints, a condition could be evaluated to both, **false** and **true**. For example, in the context of Listing 11.2 we could add

```
if (symbolic > 5),
```

which is **true** iff $x \geq 1$, and **false** otherwise. Since `symbolic` is free, both alternatives are equally possible.

Whenever more than one choice is applicable, the MLVM searches over all possible branches [DK18]. The MLVM non-deterministically selects a branch that implies a specific outcome (e. g., the condition shall be **false**). The resulting constraint is imposed on a constraint store that the MLVM maintains as part of its execution state [DK19]. After executing that branch, the MLVM backtracks execution state (constraint store, operand and frame stacks, program counter, and heap values) to the point where a choice was made, and then proceeds with the next branch.

[17] In contrast, arithmetic expressions in Java are immediately evaluated to a constant result, i. e., the original expression is lost immediately after its evaluation. This also happens in Muli for expressions that are constant.

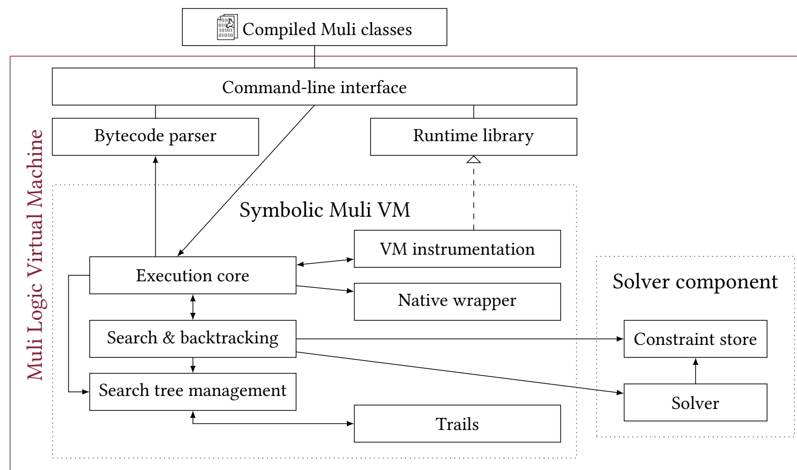


Figure 11.2: Conceptual structure of the MLVM. Adapted from [DK19] and updated in order to reflect recent developments.

With the purpose of limiting the effects of non-deterministic execution, non-deterministic branching is encapsulated. Non-deterministic programs, or *search regions*, are written in lambda expressions or methods and are passed as a parameter to one of Muli's encapsulation methods (e. g., `getAllSolutions()` or `muli()`). The result of an execution branch, i. e., either the final return value or an uncaught exception, becomes a solution to a CLOOP. The encapsulation method collects all solutions and returns them to the calling, deterministic program. Depending on the chosen encapsulation method, the surrounding program can process solutions from an array or from a stream that is evaluated non-strictly, i. e., individual solutions are computed and returned on an on-demand basis.

These features are implemented in the MLVM, whose main components are depicted in Figure 11.2. The execution core is a JVMS-compliant custom Java virtual machine [Lin+15] with modifications for symbolic execution of Java bytecode and encapsulated non-deterministic search [DK19]. The search tree management component maintains a search tree that represents the non-deterministic execution of search regions, where inner nodes represent non-deterministic choices and each leaf corresponds to a solution or an explicit failure. Within the execution core, the evaluation of a bytecode instruction that has non-deterministic behaviour in a search region results in the creation of a representation of the choice and its alternatives. This choice is then passed to the search tree management component, thus updating the search tree. The decision about which alternative to follow is delegated to the search & backtracking component, which imposes a corresponding constraint on the constraint store and checks whether the

resulting constraint system is still satisfiable using the solver component [DK19]. The solver component currently leverages the finite domain solver JaCoP [Kuc03]; alternative solvers can easily be integrated. Once an alternative is selected, the execution core continues execution on the corresponding path. The search tree management component also maintains a set of trails that record side effects during execution. The trails are used during backtracking in order to revert side effects so that a virtual machine state is achieved that is consistent for subsequent evaluations.

11.2.1 Setting the Stage for Free Objects

Reference types As Muli is based on Java, Muli distinguishes the same four distinct kinds of reference types as Java does [Gos+15, § 4.3]: type variables, array types, interface types, and class types. In this work we focus on class and interface types. Subsequently, they are subsumed under *reference types* and their instances are objects, where our focus is on *free objects* in particular. Regarding classes and interfaces, the language C# has a definition of reference types that is congruent with Java's definition [Mic20]. Therefore, even though the considerations in this paper are focused on Muli, they are also applicable to other constraint-logic object-oriented programming languages, e. g., languages based on C#.

Due to the nature of Java (and, therefore, Muli), reference types are not limited to data encapsulation. With the concept of methods, class types and interface types notably encapsulate behaviour. As a consequence of method overriding and runtime polymorphism, the behaviour may also change along the implementation hierarchy. Recall the object-oriented representation of shapes from Figure 11.1 which will serve as our running example. The `Shape` interface prescribes subtypes to implement an appropriate method `getArea()` that calculates the area from relevant field values. Moreover, in Muli, all classes inherit from `java.lang.Object` implicitly as they do in Java [Gos+15, § 4.3.2]. For the purpose of the running example, assume that each shape also overrides the default implementations of `toString()` and `equals()` that were inherited from `java.lang.Object`, facilitating representations of field values in a human-readable form as well as comparisons with other objects.

As a result, when a variable is declared, e. g., `Shape s free`, `s` can in fact hold an instance of any class that implements `Shape`. If `Shape` were a non-abstract class, an instance of `Shape` would be a possible object as well. The fact that the actual type potentially differs from the declared type affects the type casts that can (validly) be performed on `s` at

runtime, as well as the behaviour that is expected from invoking methods on the object. Consequently, adding support for free objects requires

- a non-arithmetic constraint that enforces the type (or, rather, a set of possible types) of a free object, and
- a way to discover the implementation hierarchy from all available classes.

Class discovery In order to tackle the latter, we need to make a decision regarding which classes are considered. In Java and Muli programs, classes may be available to an application even though they are not (yet) in memory from the start of an application. Instead, they reside as `.class` files in a pre-defined location on disk (the so-called class path) and are loaded on-demand by the class loader [Lin+15, § 5.3.5]. As a consequence, we can decide whether only those classes are considered that have already been loaded, as opposed to taking all classes into consideration that are on the class path. The first alternative implies that a fresh program such as

```
Shape s free;
```

might not find any implementations for `s`, unless classes that implement `Shape` were actively loaded, e. g., by constructing dummy objects from relevant classes as in

```
new Rectangle(); new Cuboid(); new Cube(); new Square();
Shape s free;.
```

Since a necessity for creating dummy objects creates additional mental load for developers, we instead propose to consider all available classes on the class path, at the cost of additional overhead for discovering and parsing all classes that are on the class path. In that case,

```
Shape s free;
```

is sufficient to instantiate a free object that can be specialized to any of its subtypes. Performing ex-ante class discovery imposes a limitation, namely that we operate under a closed-world assumption and only take classes into consideration that are present on disk at the start of the application. In Java, applications are able to create and load additional classes on the fly. However, as this feature is used rarely, it is hardly a practical issue that these classes would not be discovered.

Instances of free objects With a declaration `C o free;`, `o` becomes a logic variable that could, in theory, be substituted for either of the following:

1. an existing object from memory (the heap) that is type-compatible with `C`, thus re-using existing objects from different contexts;

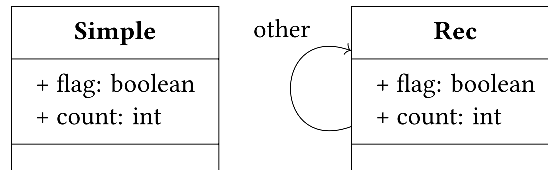


Figure 11.3: Class definitions. With a recursive definition, e. g., for `Rec`, exhaustive generation of concrete objects does not terminate.

2. a fully-generated fresh object, making a non-deterministic choice to branch non-deterministically over all possible alternative instances at the time of declaration; or

3. a fresh symbolic object `o`, which is further specified on an on-demand basis by imposing constraints on `o`.

In other contexts, a declaration `C p` implies that `p` is a fresh object, unless an existing object is assigned to `p` explicitly. As a consequence, we consider the first alternative semantically dangerous. Moreover, if existing objects were re-used, we would need to ensure that their respective scopes are not violated.

The second alternative is problematic as well since we would need to generate concrete object instances, or rather, full(!) object graphs: For an arbitrary class `C` in `C o free;`, the number of possible instances for which `o` can be substituted can be very large or even infinite. Firstly, because there may be (finitely) many specializations for `C`. Secondly, because the definition of `C` (or of a subtype) may be recursive, in turn containing a field of type `C` (or a subtype).[18] For instance, consider the definitions of `Simple` and `Rec` presented in Figure 11.3. The set of possible instances for objects of type `Simple` is a combination of the field values: $\{(true, -2147483648),\ (false, -2147483648),\ (true, -2147483647),\ (false, -2147483647),\ \ldots\}$. Therefore, with 2^{33} possible instances, the set is already very large to branch over, even though the class definition is relatively simple. Consequently, the state space becomes very large if we generate objects and branch non-deterministically at the point of declaration. The situation becomes even worse given a recursive class definition such as `Rec`, for which the generation of an instance does not terminate unless, at some level, `other == null`. For the same reason, exhaustively generating *all* instances is impossible.

For these reasons we resort to the third alternative, i. e., the declaration merely constructs a symbolic variable that can be substituted for a fresh object which is not yet known. For instance, with the definitions from Figure 11.3 and a closed-world assumption,

[18]Alternatively, the same situation occurs if `C` contains a field of a type that has a recursive definition.

```
1 public boolean isRing(C o) {
2   Set<C> seen = new HashSet<>();
3   while (o != null) {
4     if (seen.contains(o)) {
5       return true;
6     } else {
7       seen.add(o);
8       o = o.o; } }
9   return false; }
```

Listing 11.3: A method that checks whether an object o contains a ring structure, such as the one from Figure 11.4.

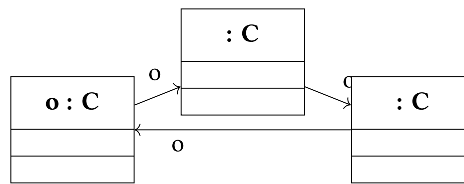


Figure 11.4: Example for an object structure that forms a ring.

an object `Rec s free;` is sufficiently specific since there are no subtypes of `Rec`, and for a symbolic variable it is possible to ignore the recursive type definition. The free object can then be made more specific by using it, e. g., by invoking a method (see Section 11.3) or by adding constraints over its fields (see Section 11.4).

Since we do not re-use existing objects, free objects are always fresh instead of picking applicable instances from memory. Consider a method `isRing(C o)` as displayed in Listing 11.3. As a consequence of excluding re-use, an invocation `C o free; isRing(o)`, cannot construct an object graph as illustrated in Figure 11.4, since each free object of type `C` would only generate another fresh free object of type `C` as its field, without being able to refer to existing objects from the heap. Therefore, generating a ring structure by chance is not possible.

Nevertheless, we can still use non-deterministic search to generate ring structures of arbitrary lengths intentionally. This is achieved with a search region that closes a ring structure by explicit assignment, as demonstrated in Listing 11.4. Using non-deterministic choice over a free boolean variable, the search region either grows the ring by instantiating a fresh free object and assigning it as the next element, or closes it by assigning the first element of the ring as the next.

```
1  public static void main(String[] args) {
2    Stream<Solution<C>> rings = Muli.muli(() -> {
3      C o free; return generateRing(o, o); }); }
4  public C generateRing(C first, C o) {
5    boolean closeRing free;
6    if (closeRing) {
7      o.o = first;
8      return first;
9    } else {
10     C next free;
11     o.o = next;
12     return generateRing(first, o.o); } }
```

Listing 11.4: Using non-deterministic choice for generating ring structures of arbitrary length, such as the one in Figure 11.4.

Instantiating a free object Since free objects are treated symbolically, a lot of instantiation effort is skipped. Specifically, constructors that would normally initialize an object are ignored. This is necessary for two reasons: first, because the MLVM only has partial information about the actual type; and second, because a class may present multiple constructors (and might even block the default constructor). Instead, since at least the supertype is known, the MLVM can initialize fields according to the definition of the supertype by putting appropriate free variables into every field. For example, a free object of type `Rec` (Figure 11.3) is initialized with `{flag = boolean free, count = int free, other = Rec free}`. However, there is one notable exception: We want free objects of a type to be consistent with regular objects of the same type. This implies that *static* fields of a type need to hold the same values for all objects of that type, regardless of whether an object is free. Therefore, if a free object is the first of its type to be instantiated, we do call static initializers of the type to ensure consistency with objects that are created later. If it is not the first, it will consistently use the same static values as other objects of its type.

Now that we have clarified what free objects look like, we continue by discussing specific interactions with free objects. Section 11.3 tackles method invocation, whereas Section 11.4 presents field access. In addition, Subsection 11.5.1 discusses operations that work explicitly on types, and Subsection 11.5.2 explains the notion of equality against the presence of both, free objects and regular objects.

```
1 public static void main(String[] args) {
2   List<Solution<String>> solutions = Muli.getAllSolutions( () -> {
3     Shape s free;
4     if (s.getArea() == 16)
5       return s.getClass().getName();
6     else
7       Muli.fail(); }); }
```

Listing 11.5: A search region that branches over the types of a free object and returns the selected classes' names.

11.3 Method Invocations on Free Objects

Recall the example structure from Figure 11.1, in which `Shape s free;` instantiates a free object that can, in fact, assume one of four distinct actual types. Its actual type is irrelevant, unless the free object is used. Therefore, once we attempt to invoke a method that is defined in `Shape`, e. g., `getArea()`, the type becomes important since the behaviour changes depending on the type. Moreover, given a single free object, using `getArea()` from one implementation and `toString()` from another would result in inconsistent behaviour and therefore does not make sense. As a consequence, when we select an implementation for invocation, we commit the free object to the type that corresponds to the selected implementation. All implementations are equally possible, so choosing is non-deterministic.

For instance, consider the program presented in Listing 11.5. It searches for instances of `Shape s` whose area (as determined by `getArea()`) is 16, and then returns the name of the actual class whose implementation has been selected. Consequently, given the structure in Figure 11.1, we expect a solution array containing the following strings (in any order): {*"Rectangle"*, *"Square"*, *"Cuboid"*, *"Cube"*}.

The program contains two method invocations on s that are discussed in the following. The first invocation is to `s.getArea()` on an unbound s. Non-deterministically selecting and invoking an implementation commits s to a specific type, thus binding s. Therefore, on the second invocation to `s.getClass()`, s is sufficiently specific, so that only a unique implementation of `getClass()` is possible, namely, the one that a class inherits from `java.lang.Object`. Consequently, the second invocation is deterministic. Similarly, an invocation `s.toString()` would be deterministic as well, even though every class provides its own implementation: Resulting from the binding that occurs when `s.getArea()` is

called, the type of `s` is sufficiently specific so that only one `toString()` implementation can possibly be selected while maintaining consistency with previous behaviour.

Generalizing from this example, for a given object `o` on which an invocation `o.m()` is performed, the runtime environment needs to discover the set of types that provide an implementation for `m()`. This discovery needs to take into consideration which types `o` may assume. Algorithm 1 calculates the set of possible implementations for `m()`, as explained subsequently. For non-free objects `o` whose class has a definition for `m()`, the returned set is a singleton (or empty in the invalid case that the type of `o` does not provide an implementation, thus yielding a runtime exception). Therefore, invocation is deterministic. For free objects, the returned set may have more elements. In that case, invocation results in a non-deterministic choice.

Algorithm 1: Discovering the set of method implementations that are candidates for invocation.

```
1  implementations(Object target, Method m)
2      Let impls := { };
3      Method mostSpecificFromSupertypes := jvmsLookup (m, target.class);
4      if mostSpecificFromSupertypes != null then
5          impls += mostSpecificFromSupertypes;
6      Let types := target.getPossibleTypes ();
7      foreach type ∈ types do
8          Method implementation := type.getMethod (m);
9          if implementation != null && !implementation.isAccAbstract () then
10             if type.isAccAbstract () || type.isAccInterface () then
11                 Let subtypes := type.getImmediateInstantiableSubtypes ();
12                 foreach subtype ∈ subtypes do
13                     impls += subtype.getMethod (m);
14             else
15                 impls += implementation;
16     return impls;
```

Methods in Java and Muli can be overloaded, so in order to target a specific overloading, `Method` signifies a combination of a method name and its descriptor, i. e., parameter types and return type [Lin+15, § 4.3.3]. In the beginning, Algorithm 1 initializes an empty set *impls* that will later contain the invocation candidates. Afterwards, `jvmsLookup()` looks up a method implementation upwards along the class hierarchy using the known lookup procedures defined in the JVMS [Lin+15, § 6.5 (*invokeinterface* and *invokevirtual*)]. This implementation is the one that will be invoked if the free object assumes its supertype

(*target*.class). Afterwards, `implementations` looks at each type that the free object may assume (*target*.`getPossibleTypes()`), searching for individual implementations (`getMethod()`). If the type that contains a found implementation is not marked as abstract or as an interface, its implementation is directly added to *impls*. Otherwise, methods from the type's immediate, instantiable subtypes[19] are added to *impls* as the original type could not be instantiated. Finally, *impls* is returned to the MLVM and is used to create the non-deterministic choice for invocation.

When an implementation alternative is selected, the runtime environment has to add a constraint $types(o) = T$ to the constraint store before executing a specific method body, where the set of types T depends on the selected implementation alternative. This constraint is added in order to ensure that, after the MLVM chooses an implementation alternative, it commits to that choice regarding later interactions with the object `o`, thus narrowing its type.

To explain the construction of the set T, have a look at the artificial implementation hierarchy displayed in Figure 11.5: There is a class `A` that implements a method `m()`. `B` inherits from `A` and overrides `m()`, adding custom behaviour. In contrast, `C` inherits from `B` but does not add custom behaviour. Last but not least, `D` inherits from `C` and provides an implementation for `m()`. Now, for a free object `A a` **`free`**, $S = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ contains the possible instance types. On invocation of `a.m()`, Algorithm 1 discovers the implementations provided by `A`, `B`, and `D`. After selecting one of the implementations, the actual type of `a` can still be one from a set of types. Specifically, the type of `a` can either be the type that provides the implementation or one of its subtypes, *except for* subtypes that provide their own implementation (as *their* respective implementation would have needed to be invoked otherwise). We call this reduced set of types T. Exemplarily, this is illustrated in Figure 11.5 where the possible types are constrained to $T = \{\texttt{B}, \texttt{C}\}$ after selecting the implementation `B.m()`. Even though `D` also is a subtype of `B`, it is not part of T as it provides an own implementation of `m()` and would therefore conflict with having chosen `B`'s implementation.

Furthermore, for the sake of completeness, assume that `B`'s implementation of `m()` calls a method `n()`, for which `C` provides its own implementation. Choosing an implementation for `m()` still reduces the set of types to `B` *and* `C`, but the later invocation of `n()` from within `m()` results in further non-deterministic branching over the two types and their respective implementations.

[19] That is, direct subtypes that are not abstract or, otherwise, their immediate instantiable subtypes.

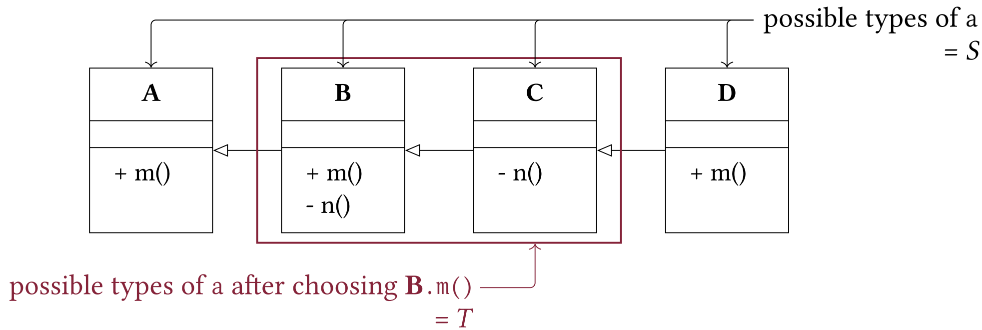


Figure 11.5: Applicable instance types for a given object `A a free` before and after choosing a specific implementation.

In Section 11.8 we present a modification of Muli's operational semantics, esp. Equation (Invoke-ND), that incorporates non-deterministic choice in method invocation. The MLVM implements Algorithm 1 and represents the non-deterministic choice by creating a `Choice` node in the active search tree. The newly created `Choice` node has one branch for each invocable implementation alternative. For each branch, an appropriate type constraint is declared that maintains integrity regarding the selected implementation. When the MLVM selects an alternative from the `Choice` node, the corresponding constraint is imposed by adding it to the constraint store, and the constraint is removed again before an alternative is evaluated.

11.4 Field Access on Free Objects

As in Java, fields of an object are accessed in Muli using a dot notation, e. g., `square.width` given an object `Square square`. In an implementation hierarchy, fields are special in that subclasses cannot override fields that they inherit. For instance, consider the example presented in Listing 11.6. The subclass declares a field with a name that is identical to that of a field in the superclass, and assigns a different value. However, as a result, the original field is merely hidden from the subclass, i. e., instances of the subclass actually have two fields with the same name [Gos+15, §§ 8.3 and 9.3]. In the example of Listing 11.6, this implies that the condition `a.field == b.field` is true, even though `b` is an instance of `Subtype`. Since both instances `a` and `b` are accessed through the `Supertype` type, the field `field` from the superclass is used in both accesses.

The example from Listing 11.6 is limited to non-free objects. But as the implementation hierarchy is irrelevant for accesses to fields of regular objects, accessing fields of free objects also does not need to consider all possible types of an instance. Therefore,

```
1 class Supertype { public int field = 2; }
2 class Subtype extends Supertype { public int field = 1; }
3 class FieldShadowing {
4   public static void main(String[] args) {
5     Supertype a = new Supertype();
6     Supertype b = new Subtype();
7     a.field == b.field; // This is true!
8   } }
```

Listing 11.6: Subclasses can hide fields of their supertypes, but fields are never overridden.

accessing a field of objects, free or non-free, is a deterministic operation that only depends on the type of the variable through which an instance is accessed.

Following the considerations regarding the initialization of free objects from Subsection 11.2.1, field access to a fresh free object yields an appropriate free variable, unless the accessed field is static. So for instance, assuming the class structure from Figure 11.1,

```
Rectangle r free;
return r.width;
```

returns a free variable of type `int`.

11.5 Other Operations on Free Objects

Method invocation and field access constitute basic functionality in object-oriented programs. However, there are further operations on objects that are affected by the introduction of free objects. In the following, we present the handling of explicit operations on the type of free objects, followed by a short discussion of the equality of (free) objects.

11.5.1 Type Operations

The handling of type operations is interesting for free objects as there is only partial information about their types. In Java/Muli bytecode, this affects the implementation of type checks (`instanceof` instruction) and type casts (`checkcast` instruction) [Lin+15, § 6.5]. These two instructions differ in that `instanceof` returns the result of the type check as a boolean value, whereas `checkcast` throws an exception if the (free) object on the operand stack cannot be cast to the intended type (otherwise, `checkcast` does nothing). For free objects, these operations are evaluated non-deterministically if either outcome is equally possible. For instance, consider the snippet in Listing 11.7, in which both type operations

are used. When the condition is evaluated, two cases are possible: Either **instanceof** returns **true** if $types(s) = \{\texttt{Rectangle}, \texttt{Cuboid}\}$ holds, or **false** otherwise. The cast in the **true** branch is always possible since the possible types of s are sufficiently constrained as a result of evaluating **instanceof**. Therefore, the cast operation is deterministic in this example.

```
1 Shape s free;
2 if (s instanceof Rectangle) {
3   Rectangle r = (Rectangle) s;
4   r.width = r.height; }
```

Listing 11.7: Using type operations on a free object.

Generally, whether type operations on a free object are non-deterministic depends on the type constraints that are imposed on the object. A type operation `o` **instanceof** `T` or `(T)o` involves a free object o and a target type T. For the decision whether an operation is non-deterministic, we define the set $SuccessfulTypes_{o,T}$ that contains all types that o may assume and that are also subtypes of or equal to T (with $a \preceq b$ meaning that a is a subtype of b or b itself):

$$SuccessfulTypes_{o,T} = \{t | t \in types(o), t \preceq T\}$$

If o is of a type $\in SuccessfulTypes_{o,T}$, the type operation is successful w. r. t. the Java Virtual Machine Specification [Lin+15, § 6.5]. An additional set, $AdverseTypes_{o,T}$, contains all types for which the operation would fail if o assumes one of these types:

$$AdverseTypes_{o,T} = \{t | t \in types(o), t \npreceq T\}$$

In relation to the set of currently possible types $types(o)$ that the object o may assume, the two sets are disjoint and their union comprises all possible types. $SuccessfulTypes_{o,T}$ and $AdverseTypes_{o,T}$ can be used for determining how both type operations are evaluated: If there is at least one element in $SuccessfulTypes_{o,T}$ the implication is that the type operation can be successful. Similarly, if $AdverseTypes_{o,T}$ is not empty, the type operation can fail. These two cases are not mutually exclusive; if both sets contain at least one element, execution branches non-deterministically.

Similar to how invocation is implemented in the MLVM, this implies that, at runtime, a `Choice` node is created, containing the two alternatives as branches with appropriate type constraints that make use of the sets calculated previously. For the branch that represents

```
1  Shape s free;
2  Rectangle r = new Rectangle();
3  r.width = 100; r.height = 101;
4  return s.equals(r); // Variation 1,
5  // or...
6  return r.equals(s); // Variation 2.
```

Listing 11.8: Example program involving non-determinism in the check for value equality.

a successful type operation on a free object o, the MLVM imposes a constraint $types(o) = SuccessfulTypes_{o,T}$. Similarly, for the alternative branch, $types(o) = AdverseTypes_{o,T}$. Before continuing execution with one of the branches of a non-deterministically evaluated type operation, the MLVM imposes the corresponding constraint, thus ensuring that the assumptions regarding a free object are consistent within a branch of execution.

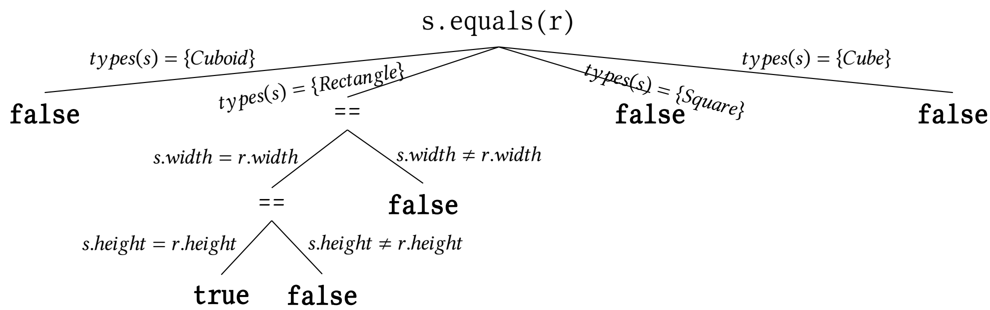
11.5.2 Equality

In Java, and therefore in Muli, there are two distinct notions of equality of objects [Gos+15, § 15.21.3]. *Reference equality* is different from *value equality* in that reference equality compares the addresses of two objects, but not their contents. Therefore, reference equality of two objects implies that they are, in fact, the same. Reference equality is tested using the `==` or `!=` operators. In contrast, value equality between two objects is tested by invoking the `equals()` method on one object, passing the other as an argument.[20] Classes may override `equals()` in order to determine whether two objects are value-equal, thus giving developers the opportunity to decide which fields must be identical for two objects to be considered equal (if any).

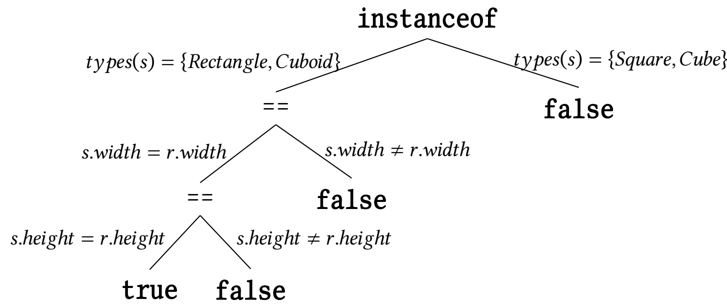
In the context of free objects, we do not need to make any particular considerations on how to handle equality for two reasons. First, using `==` or `!=` on objects deterministically compares the addresses, where free objects make no exception. Second, invoking an `equals()` method on a free object will cause the MLVM to non-deterministically evaluate the method invocation. Since `equals()` does not differ from other methods, no special handling is required other than what we discussed w. r. t. method invocation (see Section 11.3).

For the purpose of illustration, consider Listing 11.8 in the context of our running example. The code uses two variations for comparing the same two objects w. r. t. value equality. Variation 1 invokes the `equals()` method on the free object `s`. In contrast,

[20] The default implementation provided by the `Object` class falls back to comparing reference equality.



(a) Variation 1



(b) Variation 2

Figure 11.6: Execution trees created as a result of calling `equals()` on free (Variation 1) or non-free (Variation 2) objects.

Variation 2 invokes that method on a specific object, but passes `s` as a parameter. The resulting execution flows are illustrated in Figure 11.6. Variation 1 branches immediately on invocation of `equals()`, thus creating one branch per implementation of `equals()`, whereas the invocation itself is deterministic in Variation 2. In contrast, Variation 2 branches primarily because **`instanceof`** is called, checking the type of the free object (cf. Subsection 11.5.1). Both variations create comparable branches in case both the regular object and the free object are instances of `Rectangle`, which is ensured for the free object by imposing an adequate constraint. In that case, `Rectangle`'s implementation of `equals()` also compares the field values, ensuring that the custom value equality criterion is met. Furthermore, the illustration of the execution trees emphasizes that, in contrast to `==`, `equals()` is not commutative for free objects.

11.6 Demonstration

The shape application example has been useful for explaining the concepts introduced by free objects. We proceed by discussing two example applications in order to demonstrate that free objects improve the expressiveness of other Muli applications as well.

```
1  public static void main(String[] args) {
2    Solution<Queen[]> solution = Muli.getOneSolution(() -> {
3      final int n = 8; Board board = new Board(n);
4      Queen[] qs = new Queen[n];
5      for (int i = 0; i < n; i++) {
6        Queen q free; qs[i] = q; }
7      for (int i = 0; i < n; i++) {
8        if (!board.isOnBoard(qs[i])) Muli.fail();
9        for (int j = i+1; j < n; j++)
10         if (board.threatens(qs[i], qs[j]))
11           Muli.fail(); }
12     return qs; });
13   for (Queen q: solution.value)
14     System.out.println("(" + q.x + "," + q.y + ")"); } }
```

Listing 11.9: *n*-Queens search region that makes use of object-oriented features for the implementation of a search problem.

As the first example, consider the *n*-Queens problem as a classic search problem (cf. e. g., [FA03, Section 12.3]). Listing 11.9 presents a solution in Muli, assuming a class structure as illustrated in Figure 11.7. The search region initializes object representations of the board and of the *n* queens. Constraints are imposed by invoking `isOnBoard()` on the board object, thus restricting the positions of queens to $0 < x \leq n$ and $0 < y \leq n$ in accordance with the board size $n \times n$. Moreover, `threatens()` imposes constraints such that two queens may never share a diagonal, row, or column.[21] `Muli.fail()` is invoked when constraints are not fulfilled. As a result, the search region only returns placements that satisfy the constraints. This example results in two interesting observations. First, CLOOP facilitates logical grouping of data and constraint definitions using classes and objects. This is illustrated by the `Board` class that stores its dimensions in a field and derives constraints accordingly, thus demonstrating encapsulation of data and behaviour in CLOOP programs. Consequently, encapsulation in classes can be used for the purpose of structuring the constraint problem, instead of using intransparent encodings that would require additional explanations. Second, we can leverage free objects for encoding the unknown state, i. e., the placement of queens on the board.

As the second example, consider an application that systematically generates directed acyclic graphs using non-deterministic search. Such an application is useful, for example, in order to use the generated graphs to describe the hidden layers of a feed-forward

[21]For reference, the specific implementation of these two methods is displayed in Section 11.8.

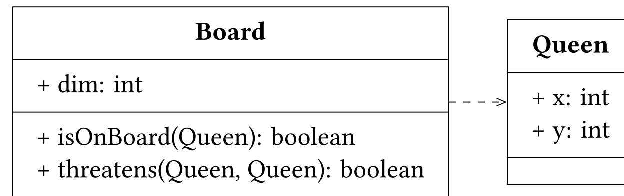


Figure 11.7: Object-oriented representation of board and queens.

artifical neural network (ANN). In an ANN, the simplest structure is an empty graph, so that the ANN's input nodes are directly connected to all output nodes. Starting from the empty graph, two operations increase the size of the graph: Either adding a hidden layer (with one node as a starting point), or adding a node to one of the hidden layers. Using Muli, we can implement a search region that enumerates graph structures by non-deterministically choosing one of these operations. Additionally, for the `AddNode` operation, it non-deterministically selects the layer to which a node is added. The non-deterministic choice for one of the operations can be implemented using the equivalent of a coin flip, i. e., using a free variable `boolean coin free;` and branching over that, adding a layer if it evaluates to `true` and adding a node to a layer otherwise. However, the mappings of `true` and `false` require explanation. Instead, with free objects and non-deterministic choice for method invocation, we can express the non-deterministic choice for selecting an operation using a free object. Given the class structure from Figure 11.8, we can implement a search region that instantiates an empty graph and systematically grows it by non-deterministic choice. The search region is displayed in Listing 11.10. In particular, note the free variable `Operation op free;` that the runtime environment non-deterministically binds to a specific type by invoking `perform()` on it, thus choosing an operation. As an implementation detail, a third operation is added that returns the current graph structure as a solution. In contrast, the other operations perform their modification on the `Graph` object and subsequently invoke `generate()` with the modified graph in order to proceed with the next operation. As a result, the possible variations in behaviour are encapsulated in classes that are named according to their function. Consequently, the search region itself can remain on a high abstraction level, whereas implementation details are moved into the respective classes.

Limitations This work comes with some limitations. First, it only considers objects as logic variables. Other kinds of reference types, esp. arrays, have a different structure and have therefore been left out of scope. Future work needs to tackle arrays as logic variables. Second, a potential parallelization of Muli applications is currently disregarded.

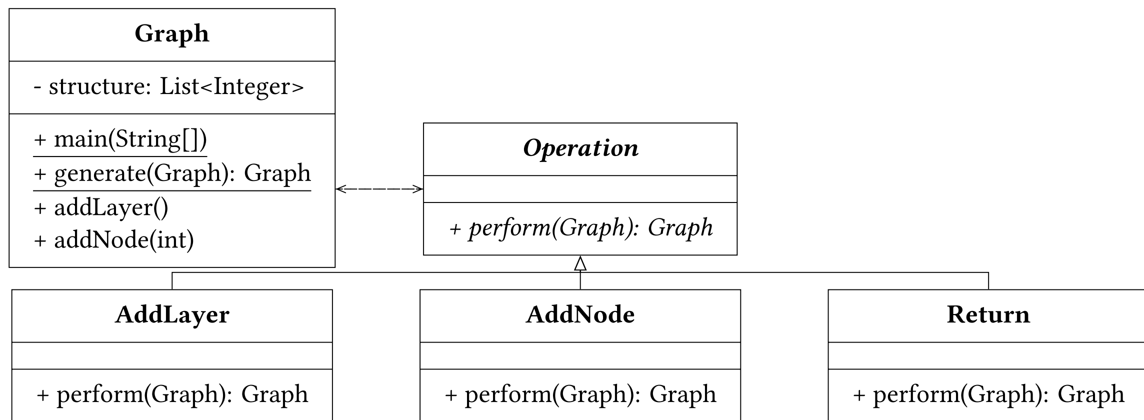


Figure 11.8: Representation of graph modification operations in a class structure for the purpose of non-deterministic choice.

```
public class Graph {
  // <...>
  public static void main(String[] args) {
    Stream<Solution<Graph>> graphs = Muli.muli(() -> {
      Graph basic = new Graph();
      return Graph.generate(basic); });
    // Consume graphs from the stream, e.g., for output.
  public static Graph generate(Graph g) {
    Operation op free;
    return op.perform(g); } }
```

Listing 11.10: Search region that generates directed acyclic graphs using non-deterministic method invocation on a free object.

The reason for that is that non-deterministic search in combination with parallelism results in state space explosion, resulting in search trees that can hardly be managed. Third, all considerations were discussed using a Java-based CLOOP language. However, they should be applicable to other (future) CLOOP languages as well because, for instance, C# uses a similar definition of reference types as that of Java.

11.7 Related Work

The present work is inspired by the concepts introduced by [Dag19]. In addition to the concepts discussed previously, our work is novel in that it provides a set of algorithms and an actual implementation in the context of the MLVM. As opposed to treating free objects symbolically and only resolving them as needed, other work has considered the object

generation approach, particularly against the background of software test-data generation [Kor90; ZL07; DZZ08]. The authors present different ways of generating all possible object instances before they are used in a program. Also in the context of test-case generation, symbolic execution for Java objects has been explored, mutating object types in order to generate test data [LLX17]. However, their approach requires initialization of reference-type logic variables, as opposed to treating them purely symbolically. Moreover, they require bytecode to be augmented with instrumentation instructions for symbolic execution, thus requiring re-compilation even of used libraries. In contrast, the MLVM operates on standard-compliant, unmodified JVM bytecode. Moreover, special kinds of objects, namely strings and lists, have been investigated, but the results are limited to data encapsulation. [Kri+20] translate string operations into constraints using a Prolog-based constraint-handling rules system. Since even the resulting strings are formulated only in Prolog, the applicability in an integrated language is limited. Lists can also be treated symbolically like our free objects, initializing just as much of them as needed for a specific program that uses the lists [KPV03].

Generally, integrating features from declarative paradigms into mainstream programming languages has proven useful. Prominent examples are the integrations of concepts from functional programming, such as the Stream API for Java and LINQ for .NET, but also integrated languages such as Scala [Ode+17; Hun18]. The language Kaleidoscope'91 attempts an integration of an object-oriented language with constraints, facilitating the specification of constraints over fields of one or more objects [FB92]. This already adds a sense of declarativity that is unknown in most current object-oriented programming languages, as fields could be formulated as expressions that combine other fields. Nevertheless, the authors do not discuss objects that are completely free as in the sense of this work, i. e., whose types are only partially known. Other work leverages object-orientation capabilities by using objects for modelling declarative expressions, e. g., for integrating Prolog search into Java programs [Ost15]. Alternatively, there are constraint solver libraries for Java, such as OptaPlanner [The20] and JaCoP [Kuc03]. Neither of these approaches achieve a full integration of constraint-logic features into an object-oriented programming language and merely provide an object-oriented abstraction layer for (some) constraint-logic features.

Closely related to CLOOP is the paradigm of functional-logic programming, most prominently represented by Curry [HKM95]. Analogous to CLOOP, functional-logic programming adds features from logic programming to a functional programming language, resulting in programming languages that are similarly non-deterministic. On the one hand, Curry has algebraic data types that resemble the data-encapsulation behaviour

of Muli objects. On the other hand, neither Curry nor other functional-logic programming approaches consider encapsulation of behaviour. Similarly, constructors in Prolog merely encapsulate data. Therefore, encapsulation of behaviour and its non-deterministic evaluation are novel contributions of the present work.

11.8 Concluding Remarks

In this paper we add the concept of free objects to constraint-logic object-oriented programming. Specifically, this work contributes the concept of logic variables with a class type or an interface type. These are particularly interesting because, in addition to encapsulating data, they also encapsulate behaviour. To that end, we propose and implement a semantics for interacting with free objects at runtime, taking non-deterministic choice over the encapsulated behaviours into account. We demonstrate our concepts by implementing them in the MLVM, i. e., in the runtime environment used by the CLOOP language Muli. A modified MLVM that includes our implementation is available on GitHub.[22]

We have shown that adding free objects to a constraint-logic object-oriented programming language improves the expressiveness of the language. CLOOP languages were already useful in applications that interleave imperative code with non-deterministic search. With the recent additions, CLOOP can also be used to express traditional constraint-logic problems in novel ways using object representations that also encapsulate behaviour, such as *n*-Queens with methods that explain constraints by using descriptive names. Moreover, CLOOP can be used for an effective formulation of new problems.

Even though the considerations in this paper are focused on Muli, they are also applicable to other constraint-logic object-oriented programming languages. For instance, since C#'s definition of reference types is congruent to that in Java, future work could port the results to a (future) constraint-logic object-oriented programming language that is based on C#. Future work sets out to add support for free arrays in order to incorporate another kind of reference type.

Appendix A: Operational Semantics of Muli (Excerpt)

[DK18] define an operational semantics for a core subset of Muli. Here, we first display an excerpt from the reduction rules that are relevant to a rule that we modify with this paper.

[22] `https://github.com/wwu-pi/muli`.

Subsequently, the modified rule is presented. Throughout the definitions, modifications to their respective originals that were necessary in order to add support for free objects are indicated in red.

Symbols In this reduction semantics, computations depend on an environment, a state, and a constraint store [DK18].

- $Env = (Var \cup \mathscr{M}) \to (\mathscr{A} \cup (Var^* \times Stat))$: Set of all environments, mapping variables $\in Var$ to addresses $\in \mathscr{A}$ and methods $\mathscr{M}$ to a tuple $((x_0, x_1, \dots, x_k), s)$, signifying parameters and a code body s. Note that we add x_0 to the original definition. x_0 shall hold the object that a method was invoked on, unless the method is static.
- $\rho_0 \in Env$ is a special initial environment that maps functions to their respective parameters and code (under the simplifying assumption that classes and their methods are in global scope).
- $\Sigma = \mathscr{A} \to (\{\perp\} \cup Tree(\mathscr{A}, \mathbb{Z}))$: Set of all possible memory states.
- A special address α_0 with $\sigma(\alpha_0) = \perp$ is reserved for holding return values of method invocations.
- $CS = \{\texttt{true}\} \cup Tree(\mathscr{A}, \mathbb{Z})$: Set of all possible constraint store states.
- $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$. Discriminating indices are added if necessary.
- $a[x/d]$ is used for modifications to a state or environment a, meaning

$$a[x/d](b) = \begin{cases} d & \text{, if } b = x \\ a(b) & \text{, otherwise.} \end{cases}$$

- The semantics of expressions is described with the infix relation $\to \ \subset (Expr \times Env \times \Sigma \times CS) \times ((\mathbb{B} \cup Tree(\mathscr{A}, \mathbb{Z})) \times \Sigma \times CS)$.
- The semantics of statements is described by the infix relation $\rightsquigarrow \ \subset (Stat \times Env \times \Sigma \times CS) \times (Env \times \Sigma \times CS)$.

Syntax The grammar is only modified slightly, incorporating method invocations on objects and reference type variables. Otherwise, taken from [DK18].

$e ::= c \mid x \mid e_1 \oplus e_2 \mid x.m(e_1, \dots, e_k)$

where $c \in \mathbb{Z}$, $x \in Var$, $e_1, \dots, e_k \in AExpr$, $\oplus \in AOp$, $k \in \mathbb{N}$,

$x.m$ can be resolved to an implementation $i \in \mathscr{M}$,

$b ::= e_1 \odot e_2 \mid b_1 \otimes b_2 \mid \texttt{true} \mid \texttt{false}$

where $e_1, e_2 \in AExpr$, $b_1, b_2 \in BExpr$, $\odot \in ROp$, $\otimes \in BOp$,

$s ::=$ `;` | `int` x`;` | `int` x `free;` | `T` x`;` | `T` x `free;` | $x = e$; | e; | $\{s\}$ | $s_1\ s_2$ |
`if` $(b)\ s_1$ `else` s_2 | `while` $(b)\ s$ | `return` e; | `fail;`
where $x \in Var$, $e \in AExpr$, $b \in BExpr$, $s, s_1, s_2 \in Stat$, `T` is a class or interface type.

Reduction Rules The following reproduces reduction rules from [DK18] as preliminaries, before presenting a rule modification for non-deterministic invocation as required for this paper.

Variable resolution:

$$\langle x, \rho, \sigma, \gamma \rangle \rightarrow (\sigma(\rho(x)), \sigma, \gamma) \tag{Var}$$

Arithmetic expressions, resulting either in a constant value if nested expressions are constant, or in a symbolic expression otherwise:

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\quad v_1, v_2, v = v_1 \oplus v_2 \in \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_2, \gamma_2)} \tag{AOp1}$$

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\ \ \{v_1, v_2\} \not\subseteq \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (\oplus(v_1, v_2), \sigma_2, \gamma_2)} \tag{AOp2}$$

Non-deterministic Invocation Under the assumption of global functions and disregarding object-oriented features, [DK18] originally define the operational semantics of invocation as a deterministic operation:

$$\frac{\begin{array}{r}\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\ \ \dots,\\ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1} \rangle \rightarrow (v_k, \sigma_k, \gamma_k),\ \ \rho(m) = (\bar{x}_k,\ s),\\ \langle s, \rho_0[{}^{\bar{x}_k}/_{\bar{\alpha}_k}], \sigma_k[{}^{\bar{\alpha}_k}/_{\bar{v}_k}], \gamma_k \rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}),\ \ \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle m(e_1, \dots, e_k), \rho, \sigma, \gamma \rangle\ \rightarrow (r, \sigma_{k+1}[\alpha_0/\ \bot], \gamma_{k+1})}$$

Adding support for object-orientation and the availability of multiple implementations for an object, we modify and replace the rule as presented subsequently (changes highlighted in red). The new rule uses the set $implementations(o, m)$ that is calculated using Algorithm 1 in order to find possible implementing types.

$$\frac{\begin{array}{c}\langle o, \rho, \sigma, \gamma \rangle \rightarrow (v_0, \sigma, \gamma),\ \ \langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\\ \ldots,\ \ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1} \rangle \rightarrow (v_k, \sigma_k, \gamma_k),\ i \in implementations(v_0, m),\\ \rho(i) = (\bar{x}_k,\ s),\ \ \langle s, \rho_0[{}^{\bar{x}_k}/{}_{\bar{\alpha}_k}], \sigma_k[{}^{\bar{\alpha}_k}/{}_{\bar{v}_k}], \gamma_k \wedge types(o) = T \rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}),\ \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle o.m(e_1, \ldots, e_k), \rho, \sigma, \gamma \rangle \ \rightarrow (r, \sigma_{k+1}[\alpha_0/\ \bot], \gamma_{k+1})} \quad \text{(Invoke-ND)}$$

For non-free objects *target* whose class has a definition for m, $implementations(target, m)$ is a singleton. Therefore, invocation is deterministic. For free objects, $implementations(target, m)$ may have more elements. In that case, the evaluation of this rule becomes non-deterministic. Moreover, note that a constraint $types(o) = T$ is added after selecting a specific implementation. T depends on the selected implementation alternative as explained in Section 11.3 (illustrated with Figure 11.5).

The new rule depends on the rule in Equation (Var) for resolving the object variable o based on the state of environment and memory, and on the rules in Equations (AOp1) and (AOp2) for substituting parameter expressions. The definition assumes that the environment ρ contains every method definition, comprising a parameter definition $\bar{x}_k$ and a body s.

Appendix B: Implementation of Board and Queens

The following code implements the class structure as illustrated in Figure 11.7.

```
1  public class Board {
2      final int dim;
3
4      public Board(int dim) { this.dim = dim; }
5
6      public boolean isOnBoard(Queen q) {
7          if (q.x < 0) return false;
8          if (q.x > dim-1) return false;
9          if (q.y < 0) return false;
10         if (q.y > dim-1) return false;
11         return true; }
12
13     public boolean threatens(Queen p, Queen q) {
14         if (p.x == q.x) return true;
```

```
15          if (p.y == q.y) return true;
16          if (p.x - p.y == q.x - q.y) return true;
17          if (p.x + p.y == q.x + q.y) return true;
18          return false; } }
19
20  public class Queen { int x, y; }
```

References

[Dag19] Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming*. Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8.

[DK18] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DK19] Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

[DZZ08] A. V. Demakov, S. V. Zelenov and S. A. Zelenova. 'Using abstract models for the generation of test data with a complex structure'. In: *Programming and Computer Software* 34.6 (2008), pp. 341–350. ISSN: 03617688. DOI: 10.1134/S0361768808060054.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Berlin Heidelberg: Springer, 2003. ISBN: 978-3-642-08712-7.

[FB92] Bjorn N. Freeman-Benson and Alan Borning. 'Integrating Constraints With an Object-Oriented Language'. In: *ECOOP 92*. Vol. 615. 1992, pp. 268–286. ISBN: 978-3-540-55668-8. DOI: 10.1007/BFb0053042.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 03/05/2019).

[HKM95] Michael Hanus, Herbert Kuchen and Juan Jose Moreno-Navarro. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Hun18] John Hunt. *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. 2nd ed. Springer, 2018. ISBN: 978-3-319-75770-4.

[Kor90] Bogdan Korel. 'Automated Software Test Data Generation'. In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879. ISSN: 00985589. DOI: 10.1109/32.57624.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu and Willem Visser. 'Generalized Symbolic Execution for Model Checking and Testing'. In: *TACAS'03 Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems*. 2003, pp. 553–568.

[Kri+20] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau and Dierk Ehmke. 'Towards Constraint Logic Programming over Strings for Test Data Generation'. In: *Declarative Programming and Knowledge Management*. Vol. 12057. 2020, pp. 139–159. DOI: 10.1007/978-3-030-46714-2_10.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 1084-4309. DOI: 10.1145/785411.785416.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 03/05/2019).

[LLX17] Lian Li, Yi Lu and Jingling Xue. 'Dynamic symbolic execution for polymorphism'. In: *ACM International Conference Proceeding Series* (2017), pp. 120–130. DOI: 10.1145/3033019.3033029.

[Mic20] Microsoft. *Reference types (C# Reference)*. 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types (visited on 16/04/2020).

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/ (visited on 03/05/2019).

[Ost15] Ludwig Ostermayer. 'Seamless Cooperation of Java and Prolog for Rule-Based Software Development'. In: *Proceedings of RuleML 2015*. 2015. URL: http://ceur-ws.org/Vol-1417/paper2.pdf.

[The20] The OptaPlanner Team. *OptaPlanner User Guide, Version 7.32.0*. 2020.

[ZL07] Ruilian Zhao and Qing Li. 'Automatic Test Generation for Dynamic Data Structures'. In: *Fifth International Conference on Software Engineering Research, Management and Applications*. 2007, pp. 545–549. DOI: 10.1109/SERA.2007.64.

12

Structured Traversal of Search Trees in Constraint-Logic Object-Oriented Programming

Jan C. Dageförde* · **Finn Teegen**†

Citation Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13.

Abstract In this paper, we propose an explicit, non-strict representation of search trees in constraint-logic object-oriented programming. Our search tree representation includes both the non-deterministic and deterministic behaviours of executing an application. Introducing such a representation facilitates the use of various search strategies. In order to demonstrate the applicability of our approach, we incorporate explicit search trees into the virtual machine of the constraint-logic object-oriented programming language Muli. We then exemplarily implement three search algorithms that traverse the search tree on-demand: depth-first search, breadth-first search, and iterative deepening depth-first search. In particular, the last two strategies allow for a complete search, which is novel in constraint-logic object-oriented programming and highlights our main contribution. Finally, we compare the implemented strategies using several benchmarks.

Keywords Constraint-logic object-oriented programming · explicit search tree · complete search strategy · virtual machine implementation.

*University of Münster, Germany
†CAU Kiel, Germany

```
1 boolean flipCoin() {
2   int coin free;
3   if (coin == 0)
4     return false;
5   else
6     return true; }
```

Listing 12.1: A simple non-deterministic search region in Muli for the demonstration of constraint-logic object-oriented programming concepts.

```
1 boolean flipTwoCoins() {
2   int coin1 free, coin2 free;
3   if (coin1 == 0)
4     return false;
5   else if (coin2 == 0)
6     throw Muli.fail();
7   else
8     return true; }
```

Listing 12.2: Muli search region example that comprises two solutions and a failure.

12.1 Motivation

In constraint-logic object-oriented programming, combining imperative code with features from logic programming causes the runtime to execute parts of the imperative code non-deterministically ("don't know" non-determinism). To give an example, the program (or search region) depicted in Listing 12.1 has two solutions. The example is written using the Münster Logic-Imperative Language (Muli), which we explain in Section 12.2. The search region declares a boolean logic variable `coin`. Subsequently, evaluating the `if` statement causes the runtime environment to take and implement a decision regarding the potential value of `coin`, thus introducing non-determinism. Consequently, implementing the decision selects a single branch of execution, eventually resulting in one of the two outcomes.

Non-deterministic execution is useful for applications involving search, i. e., an application would usually cause the runtime environment to evaluate more than one branch. To that end, the runtime environment systematically evaluates multiple alternative branches in sequence. Non-deterministic branching dynamically creates an implicit search tree that represents the various execution paths that lead to alternative outcomes of a program. The goal of the present work is to make this search tree explicit at runtime. It encodes

the various execution paths of a program, the choices encountered along every path, and every path's outcome (i. e., solution or failure). As there can be paths of infinite length, our search tree representation is non-strict. Our search tree then serves as a basis for structured traversal by arbitrary search algorithms, including iterative deepening depth-first search. Furthermore, by making the search tree explicit, it is possible to inspect the search tree at any given point in time, e. g., after search or even at an intermediate stage. This way, the search tree aids in effective debugging.

This paper provides the following contributions:

- A general *search tree structure* for constraint-logic object-oriented programming that encapsulates execution state (Section 12.4).
- *Search algorithm implementations* that traverse the search tree structure for finding solutions to constraint-logic object-oriented programs (Section 12.5).
- A discussion of the implications of our work for executing object-oriented (imperative) programs non-deterministically (Section 12.6).

First of all, Section 12.2 introduces concepts of constraint-logic object-oriented programming, followed by an outline of the Muli virtual machine in Section 12.3.

12.2 Constraint-Logic Object-Oriented Programming

Constraint-logic object-oriented programming combines the flexibility of imperative and object-oriented programming with features from constraint-logic programming, namely logic variables, constraints, and search. Muli is a constraint-logic object-oriented programming language that is based on Java [DK19a].

In Muli, *logic variables* are declared in a way that is similar to declaring regular variables. As indicated in Listing 12.1,

```
int coin free;
```

declares a logic variable of a primitive (integer) type. Instead of assigning a constant value, the `free` keyword specifies that `coin` is a logic variable. A logic integer variable can be used interchangeably with other integer variables, i. e., they can become part of conditions or arithmetic expressions and can be passed to methods as parameters [DK18]. In contrast to regular variables, logic variables are used symbolically. Recent work is looking into support for reference-type logic variables [Dag19], but here we focus on logic variables of primitive types.

Constraints are defined as relational expressions, (typically) involving logic variables. For simplicity, Muli does not provide a dedicated language feature for imposing con-

straints. Instead, a constraint is imposed whenever the flow of execution branches, such as when a branching condition is evaluated. Therefore, constraints are derived from boolean expressions. For instance, in Listing 12.1

```
if (coin == 0) { s1 } else { s2 }
```

`coin` occurs in the condition and is not sufficiently constrained, so that the condition can be evaluated to either true or false. As a result, the evaluation of the condition creates a *choice*, from which alternatives are evaluated non-deterministically. The runtime environment selects an alternative by imposing the corresponding constraint. In our example, by imposing $coin \neq 0$ the runtime environment can proceed with the evaluation of s_2. The runtime environment leverages a constraint solver for finding solutions as well as for cutting execution branches early as soon as their constraint system becomes inconsistent.

Search transparently performs non-deterministic evaluation in combination with backtracking until a solution is found. Implicitly, following a sequence of choices (and taking decisions at each choice) produces a (conceptual) search tree that represents the order of execution. In such a search tree, inner nodes are choices and leaves represent alternative ends of execution paths. In Muli, an execution path ends with a *solution* (specified by either **return** or **throw**) or with a *failure*, e. g., if a path's constraint system is inconsistent. The full listing of our example in Listing 12.1 demonstrates how solutions are returned. After search completes, solutions of the example are **false** and **true** (in any given order).

Moreover, applications sometimes require an *explicit failure* denoting the end of an execution path without a solution. In Muli, an explicit failure is expressed by **throw** `Muli.fail()`. Nevertheless, executing that statement will not return an exception. Instead, the statement is specifically interpreted by the runtime environment, resulting in backtracking. Listing 12.2 provides a slightly extended search region with three execution paths, one of which ends in a failure.

The main program is executed deterministically, whereas all non-deterministic search is *encapsulated*. Encapsulation gives application developers control over search. In addition to coarse-grained control (i. e., requesting either a single solution or an array comprising all solutions), Muli offers fine-grained control by returning a Java stream that evaluates solutions non-strictly. `Muli.muli()` accepts a `Supplier` and returns a stream of `Solution` objects. In Java (and, therefore, in Muli), a `Supplier` denotes either a lambda expression or a method reference (both without arguments). We refer to the method that is passed as an argument as a *search region*, as it will be executed non-deterministically and therefore describes the constraint-logic object-oriented problem. Following the

principles of the Java Stream API, solutions can be retrieved from the stream individually on demand [DK19b]. For instance, considering Listing 12.1, a stream is initialised using

```
Stream<Solution<Boolean>> stream = Muli.muli(self::flipCoin),
```

and the actual search starts as soon as the first solution is requested from the stream.

12.3 Muli Logic Virtual Machine

The Muli Logic Virtual Machine (MLVM) is a runtime environment for Muli. The MLVM is a custom Java Virtual Machine (JVM) that complies with the JVM Specification (see [Lin+15]) for deterministic execution. Moreover, it adds modifications that support Muli-specific extensions, particularly symbolic execution and non-deterministic execution [DK19a]. As in a regular JVM, execution state is represented in the MLVM by a combination of *program counter (PC)*, a *heap*, a stack of executed method frames (*frame stack*), and an *operand stack* per frame. Additional state serves the purpose of supporting non-deterministic execution and constraints. In particular, this includes the constraint stack and the trail.

The *constraint stack* maintains the active constraint system, i. e., the conjunction of all constraints on the stack [DK19a]. Representing the constraint system in a stack structure is beneficial as constraints are added dynamically during execution. Consequently, on backtracking, only the most recently added constraints need to be removed from the stack. Moreover, the *trail* records changes that are made to the virtual machine (VM) state during execution. On backtracking, the information on the trail can be used to revert to a previous execution state. More precisely, using the trail, backtracking achieves the specific state of the choice at which the next decision can be made. In fact, the trail is therefore split up into incremental trails, one per choice, each describing how to backtrack towards the next choice. In addition, in order to be able to not only backtrack to a choice (upwards along a search tree) but to achieve an arbitrary previous state (including downward navigation), the MLVM maintains two trails per choice, one being the inverse of the other [DK19b]. In the following, we call the trail for backtracking *backward trail*, as opposed to the *forward trail* that is used to navigate downwards.

Like a regular JVM, the MLVM reads applications from bytecode and executes bytecode instead of the original source. Muli's bytecode format is compatible with that described in [Lin+15], merely adding custom attributes in order to represent logic variables [DK19a]. For instance, the example application from Listing 12.2 compiles to the bytecode instructions in Listing 12.3. Some bytecode instructions exhibit non-deterministic behaviour. For

```
1   0: iload_1              // coin1
2   1: iconst_0
3   2: if_icmpne (7)        // coin1 != 0
4   5: iconst_0
5   6: ireturn              // return false
6   7: iload_2              // coin2
7   8: iconst_0
8   9: if_icmpne (16)       // coin2 != 0
9   12: invokestatic  #91   // fail()
10  15: athrow
11  16: iconst_1
12  17: ireturn             // return true
```

Listing 12.3: Bytecode generated by the Muli compiler for the program in Listing 12.2.

Triggering bytecode instruction	Type of choice	No. of decisions
`If<cond>`, `If_icmp<cond>`	**if** instruction, integer comp.	2
`FCmpg`, `FCmpl`, `DCmpg`, `DCmpl`	floating point comparison	2
`LCmp`	long comparison	3
`Lookupswitch`, `Tableswitch`	**switch** instruction	1 per case + 1

Table 12.1: Bytecode instructions that may cause non-deterministic branching upon execution. `<cond>` is a placeholder for specific comparisons, e. g., `eq` for equality.

instance, `if_icmpne` in Listing 12.3 jumps to the specified instruction if the two integer operands on the operand stack are not equal. Otherwise, execution continues linearly with the following instruction. If one or both operands are logic variables, both *jumping* and *not jumping* are feasible alternatives. As logic variables are used in the current example, the execution of `if_icmpne` instructions creates choice points that offer two decision alternatives. While **if** instructions always provide two alternatives (i. e., jumping to the **else** branch or not), **switch** instructions result in alternatives according to the number of **case**s plus one for the **default** case, each jumping to instructions accordingly. Table 12.1 provides a reference of instructions that may exhibit non-deterministic behaviour and counts the decision alternatives from which the MLVM chooses.

Executing a bytecode instruction with non-deterministic branching creates a choice point in the MLVM [DK19a]. Prior to this work, the implementation of the choice point itself was responsible for managing the execution of its branches. More specifically, executing a bytecode instruction created a choice point representation in the MLVM.

Consequently, the created choice point contained information about applicable branches, but also implemented the behaviour of search. That is, upon creation, the choice point representation immediately selected the first decision alternative and applied it, thus committing to a specific branch. The created choice point representations are stored in a stack of choice points. The MLVM referred to the choice point stack during backtracking. Starting from the top, it popped choice points until reaching one with an alternative that had not been evaluated yet. It then immediately committed to this alternative by adding its constraint and following its path.[23] As a consequence, the runtime environment never actually stored an explicit representation of the search tree. Instead, the choice point stack merely maintained a single path through the (implicit) search tree. Therefore, diverting from the currently executed path was not possible, effectively restricting the search capabilities of the MLVM to depth-first search. All things considered, the previous MLVM used a complex, tangled mixture of responsibilities in which bytecode-instruction implementations, choice-point implementations, and the VM realise non-deterministic search in combination.

In a cleaner architecture,

- declaratively executing a bytecode instruction creates choice objects and just returns them to the MLVM (instead of performing a decision right away), and
- choice objects only hold information about available decision alternatives (but no implementation for taking decisions).

As a consequence, the MLVM is the only element that is allowed to change execution state by committing to decisions, instead of sharing this permission with choice objects or instruction implementations. The search tree structure that we discuss subsequently facilitates an explicit representation that holds a declarative representation of choices and of the alternatives that each choice provides. Overall, the structure serves as a clean basis for following arbitrary execution paths through the tree.

12.4 Search Trees

A declarative, explicit search tree representation lays the groundwork for following arbitrary execution paths instead of limiting execution to depth-first search only. We first explain the conceptual representation, outlining the intuition of the elements that constitute the search tree. Afterwards, we describe how a search tree is constructed

[23] Provided that the constraint system was still consistent. Otherwise, backtracking occurred until the next choice point that offered an unevaluated, feasible alternative.

dynamically during the execution of a Muli application. Last, we abstractly describe navigation through the search tree as the basis for search.

12.4.1 Representation

Conceptually, our explicit search tree comprises five distinct node types. There are node types for returned values, thrown exceptions, choices between non-deterministic branches, failed computations, and yet unevaluated search trees. Figure 12.1 shows a class diagram for our search tree representation. Basically, this representation corresponds to an algebraic data type and therefore does not implement any decision-taking in contrast to the previously used choice points.

As solutions of a search region, a `Value` node holds the value returned by a computation while an `Exception` node does the same with an exception that has been thrown. A `Fail` node represents either an explicit failure or branches whose constraint system is inconsistent. As a consequence, it does not hold any values. Furthermore, `Choice` nodes store a list of subtrees which, in turn, reference their `parent` choice. Having an explicit reference to each node's parent allows for easy and direct navigation through the search tree. For the root node of a search tree, the `parent` attribute is **`null`**. Finally, `UnevaluatedST` serves as a proxy for subtrees that have not been evaluated yet, facilitating non-strict usage.

Moreover, each node in the search tree stores fields that prepare for later execution. The `frame` and `pc` fields represent a reference to the (mutable) stack frame and the value of the PC at which the node has been created. Each node holds an optional constraint expression that has to be satisfied in order to reach this node, e. g., as a consequence of non-deterministic branching. Additionally, the backward trail stores the changes to the VM state that were made in order to reach this node (thus preparing for backtracking), whereas the forward trail stores changes that are needed in order to return to this node afterwards. In combination, these fields are used to properly manipulate the state of the MLVM during the traversal of the search tree, which is discussed in detail in Subsection 12.4.3.

12.4.2 Construction

The actual search tree is constructed during search. A search strategy is responsible for determining the order in which the search tree is traversed. Regardless of the order, a search strategy evaluates `UnevaluatedST` nodes as long as there are such nodes left and the

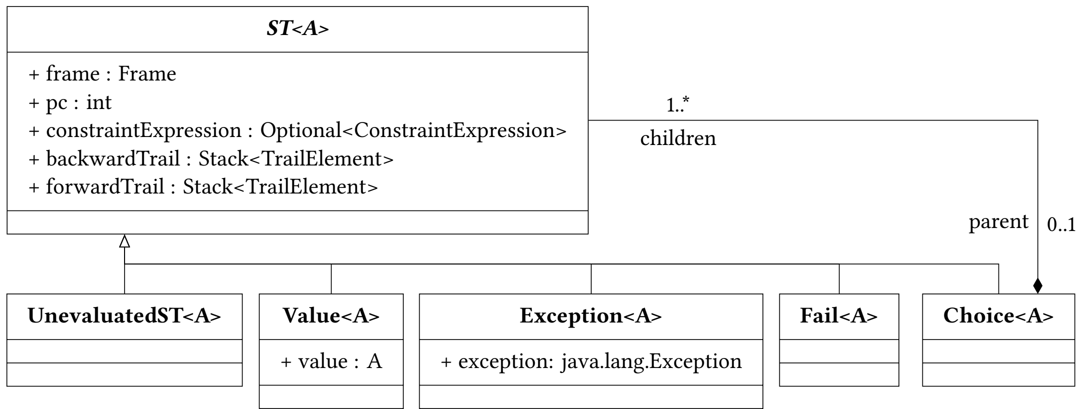


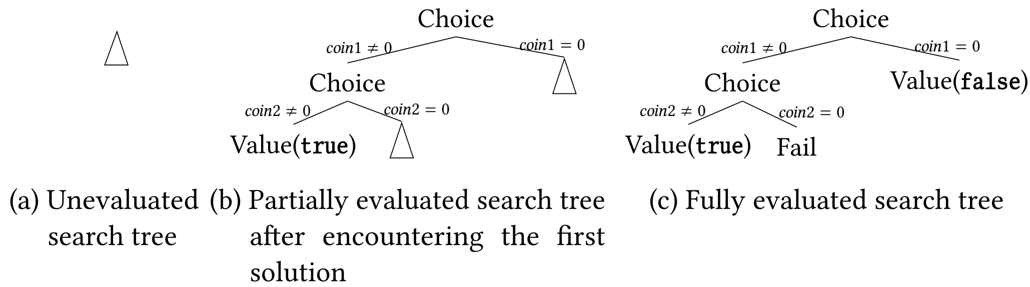
Figure 12.1: Class diagram for the representation of search trees.

encapsulating program demands additional solutions. In general, the MLVM evaluates an `UnevaluatedST` node by imposing the node's constraint and executing the bytecode of the search region starting from the PC, which the node points to, until either of the following situations occurs.

- The computation in the search region returns with a value,
- an uncaught exception occurs during execution,
- the method `Muli.fail()` signals a failed computation, or
- one of the instructions in Table 12.1 is executed, which results in the creation of a `Choice` object.

In any case, the `UnevaluatedST` node in the search tree is replaced by its evaluated counterpart, i. e., by a `Value`, `Exception`, `Fail`, or `Choice` node. Note that all children of a newly created `Choice` node are unevaluated search trees initially. Furthermore, state changes that were made during this evaluation are received from the MLVM and stored within the new node as its backward trail.

At the beginning of search, the search tree is unknown and therefore initially represented by a single `UnevaluatedST` node. The PC of that node points to the start of the search region, and the optional constraint expression is left empty since no constraints apply to the start of a search region. Similarly, the trails are empty as this node has not yet been evaluated. Figure 12.2 exemplarily shows three search trees for the program from Listing 12.2 that all are evaluated to a different degree, and thus illustrate various intermediate evaluation stages that can occur during a search. The illustration assumes a depth-first search strategy; therefore, other search strategies will result in different intermediate stages.



(a) Unevaluated search tree

(b) Partially evaluated search tree after encountering the first solution

(c) Fully evaluated search tree

Figure 12.2: Different evaluation stages of the search tree corresponding to the search region in Listing 12.2. The constraint of each subtree is noted at the respective edge.

12.4.3 Traversal

The implementation of any search algorithm requires to be able to navigate through the search tree in any direction, i. e., upwards and downwards. For example, if a branch of a search tree has been fully evaluated, search continues elsewhere. While navigating through the search tree, it is vital to ensure that the MLVM remains in a consistent state. A node's forward and backward trail together with its frame and PC are used for that purpose. In general, navigation takes place from an already evaluated node to another evaluated node, since only evaluated nodes have a trail (see Subsection 12.4.2). More specifically, a `Choice` node is always the target node or source node when navigating upwards or downwards.

We navigate upwards in a search tree by following references to the parents until we reach the target node (e. g., the root), backtracking the VM state in the process. In doing so, we remove previously imposed constraints from the constraint stack and undo the changes to VM state by processing the backward trails of nodes along the path. At the same time, the backward trails are converted into forward trails so that a node from which we navigate away can be reached again later when navigating downwards, e. g., for the evaluation of another subtree of that node. Last but not least, the frame and PC of the VM are set accordingly, using the information that was recorded at each node when it was created.

Navigating downwards is slightly more complicated as we first need to determine how to reach a target node from the current (source) node. However, we always have a reference to the target. Therefore, we can use the target's parents in order to find the path to the source. Afterwards, we process the path in reverse order, thus getting from the source node to the target node. We basically do the opposite of what is done in upwards navigation: For each node, we set the frame and PC to what is recorded in

```
1  void navigateUpwards(ST from, Choice to) {
2    while (from != to) {
3      if (from.constraintExpression.isPresent())
4        constraintStack.pop();
5      vm.processTrail(from.backwardTrail, from.forwardTrail);
6      vm.setFrame(from.frame); vm.setPc(from.pc);
7      from = from.parent; } }
8
9  void navigateDownwards(Choice from, ST to) {
10   Stack<ST> nodes = new Stack<>();
11   while (to != from)
12     nodes.put(to); to = to.parent;
13   while (!nodes.empty()) {
14     to = nodes.pop();
15     vm.setFrame(to.frame); vm.setPc(to.pc);
16     vm.processTrail(to.forwardTrail, to.backwardTrail);
17     if (to.constraintExpression.isPresent())
18       constraintStack.push(to.constraintExpression.get()); } }
```

Listing 12.4: Methods for navigating upwards and downwards in a search tree.

the node, apply the forward trail to reapply changes to the execution state, and impose a node's constraint if present. Simultaneously to processing the forward trail, we convert it again into a backward trail to be later able to navigate upwards. For clarity, Listing 12.4 shows simplified implementations for navigating upwards and downwards, respectively. Subsequently, these general navigation methods serve as primitives for traversal.

12.5 Search Strategies

As a demonstration of how the explicit search tree representation can be employed for the implementation of search strategies, we outline the implementations of three particular ones.

Depth-first Search

The implementation of depth-first search maintains a stack of unevaluated subtrees from the search tree. At the beginning of the search, the initial node (see Subsection 12.4.2) is pushed to the stack. Then, depth-first search repeatedly pops an unevaluated search tree node from the stack and tries to evaluate it. If its evaluation results in a `Choice` node, its

```
1 Choice findCommonAncestor(ST a, ST b) {
2   initialise empty set;
3   while (b != null) {
4     add b to set;
5     b = b.parent; }
6   while (!set.contains(a))
7     a = a.parent;
8   return a; }
```

Listing 12.5: Algorithm for finding the first common ancestor of two nodes.

children are pushed to the stack and search continues by popping the next node from the stack (i. e., a local subtree). Otherwise, if a `Value` or `Exception` node is encountered, the search strategy must be able to return the result to the encapsulating program. To that end, it reverts execution state to the state from the beginning of search using `navigateUpwards`. When search is picked up again, the search strategy uses `navigateDownwards` in order to evaluate the next node from the stack. Finally, if the node at hand is evaluated to a `Fail` node, local backtracking is performed, i. e., we navigate upwards to the nearest parent that has at least one unevaluated subtree.

Breadth-first Search

Instead of a stack, a FIFO queue keeps track of unevaluated subtrees. Beginning or resuming search dequeues nodes from the head of the queue. In contrast, when a `Choice` node is encountered, its children are enqueued at the end. Another difference is the fact that breadth-first search requires navigating between arbitrary nodes within the search tree. While it is, of course, possible to go over the root node, it is more efficient to navigate along a path going over the first common ancestor of the two involved nodes. Listing 12.5 shows a simple algorithm that determines the first common ancestor of two nodes in the search tree. Once the first common ancestor is found, search combines `navigateUpwards` (to the found ancestor) and `navigateDownwards` in order to efficiently navigate between two arbitrary nodes.

Iterative Deepening Depth-first Search

Our search tree can also be used to implement an exciting variant of iterative deepening search. Iterative deepening provides the strength of depth-first search while ensuring that solutions can be found even if there are execution paths of infinite length. In iterative

deepening, search is bounded by a constant maximum depth. Search proceeds in a depth-first manner until nodes are reached that are at the maximum depth. In that case, search first evaluates other nodes up to that depth, thus assuming breadth-first search behaviour. Only if additional solutions are required, search increases the bound, again by a constant, and so on. In Muli, aided by the inverse trails, when the bound is increased, the runtime environment does not need to restart computation at the root, which usually leads to a reevaluation of known execution paths (and solutions). Instead, it leverages the (partial) search tree and the recorded inverse trails in order to restart computation from known states that provide further alternatives.

12.6 Discussion

The implementation of our search tree structure in the MLVM facilitates the non-deterministic execution of imperative (object-oriented) programs in novel ways, using search strategies that could not be implemented without an explicit structure. The existing depth-first search strategy has been reimplemented and is now based on the explicit search tree structure as well. In order to ensure that the required changes do not adversely affect the performance of depth-first search, we first compare the runtime behaviour before discussing novel aspects of search. Note that we measure only performance, not memory consumption. Obviously, maintaining the search tree requires more memory than merely storing the current execution path. However, a possible memory optimisation would be to discard search tree nodes that belong to exhaustively evaluated subtrees — especially in depth-first search strategies.

We are interested in comparing the performance of depth-first search in the new search-tree-based and old choice-point-stack-based implementations. To that end, a set of experiments is conducted in a modified MLVM that contains our search-tree structure as well as in an MLVM without modifications, each executed by OpenJDK 1.8.0_212.[24] Since the MLVM is executed by a JVM, we drop the first 15 executions in order to account for effects caused by just-in-time compilation and take the performance values of subsequent executions. In total, we aggregate performance values of 500 executions per experiment, tackling classic search problems. The first experiment calculates a solution to the 3-partition problem for a fixed set of integer values using a depth-first search strategy. Until the first solution is found, search passes 374 choices. The second finds a solution to the Send More Money puzzle. For reference, we also execute corresponding

[24] Ubuntu 18.04.2 with 4.15.0 x86_64 GNU/Linux kernel; Intel Core i5-5200U CPU.

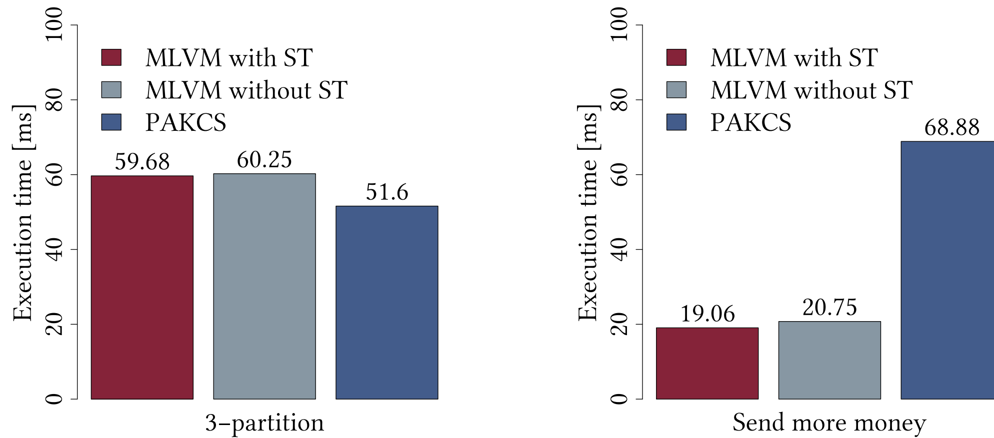


Figure 12.3: Comparison of execution times in MLVM with or without explicit search trees, both using depth-first search. Execution times in PAKCS for reference.

Curry implementations on PAKCS 2.1.1 using depth-first search. Figure 12.3 features the average execution times. Our experiment indicates that the implementation and use of an explicit search tree do not negatively affect depth-first search performance. Moreover, the comparison to PAKCS is encouraging, seeing that Muli search regions offer competitive performance while providing support for using side-effects during non-deterministic execution.

Since the use of explicit search trees does not add visible overhead to execution times, we can focus on the benefits of using a search tree representation at runtime. The MLVM now features additional search algorithms beyond depth-first search that all leverage the search tree structure. In particular, using breadth-first search is novel to the non-deterministic execution of imperative programs that have side-effects.

Consider the search region from Listing 12.6. For lack of a termination condition, there is one infinite execution path. Therefore, it is impossible to evaluate the search tree (or the application) strictly. In our depth-first search implementation, the infinite execution path is the leftmost one. As a result of this structure, depth-first search is unable to compute a single solution. In contrast, several solutions can be returned using a breadth-first or iterative deepening strategy, even though the tree can never be evaluated in full. As a more sophisticated example, we have implemented a search region that finds solutions to the Water jugs problem. Here the MLVM is unable to evaluate a full search tree as there are cyclic execution paths that result in valid solutions or failures. We have executed these programs using the available strategies 500 times for up to ten seconds each and indicate the average number of solutions in Table 12.2.

```
1 private static boolean nonTerminatingCoin() {
2   int coin free;
3   if (coin == 0)
4     return true;
5   else
6     return nonTerminatingCoin(); }
```

Listing 12.6: Muli search region featuring an infinite amount of execution paths.

	DFS	BFS	ID-DFS
Simple infinite recursion	0	1469.7	1555.2
Water jug problem	0	29.5	34.4

Table 12.2: Comparison of search strategies w. r. t. the number of solutions that are returned within ten seconds.

Note that the results do not imply that depth-first search is generally a bad strategy. On the contrary, the combination of increased memory requirements and the time needed for changing VM state using the trail still speaks against using breadth-first search by default. Iterative deepening shares this disadvantage in case that additional levels of the search tree need to be evaluated (but is as efficient as depth-first search if the initial depth is sufficient). Consequently, the results indicate that iterative deepening depth-first search is a good trade-off, if not a better strategy. Further evidence is needed to conclusively argue that iterative deepening is a superior strategy in general. In any case, both are useful strategies in certain situations in which depth-first search falls short.

The search tree structure that is presented in this paper is conceptually similar to the ST structure known from the KiCS2 compiler for Curry [HPR12]. However, Curry search trees only encode evaluation alternatives of an expression. In contrast, search trees for constraint-logic object-oriented programming need to encode the execution behaviour, i. e. VM state changes, that results from different alternatives. Consequently, the state changes are recorded on the corresponding paths that lead to solutions, so that the VM can change state depending on the alternative that is being evaluated. In our current work, we do this by maintaining the forward and backward trails on edges of the search tree.

Prior to our work, the execution state of constraint-logic object-oriented programming in Muli was represented by the PC, frame stack, operand stacks, constraint stack, trail, and choice point stack. Our work results in a slightly altered definition of execution state.

What previously was a choice point stack is now replaced by the search tree and a pointer to the current search tree node that is under evaluation. In addition, a search algorithm is responsible for maintaining a suitable data structure that keeps track of the progress of traversing the search tree, e. g., a stack of not-yet-evaluated choices in depth-first search algorithms.

Moreover, the explicit search tree structure is useful for the development of constraint-logic object-oriented programs, as it can be helpful to visualise the structure of search. Specifically, we can visualise at which points different kinds of choices are introduced and which solutions are encountered by the runtime environment. During the development of the MLVM, the search tree structure is useful for ensuring that non-deterministic branching and search algorithms are implemented correctly. In contrast, the structure of the previous approach impeded the diagnosis of problems with non-deterministic execution, as only the current execution path was represented. Consequently, relevant information about previously encountered choices and solutions was lost, whereas this information is adequately represented in the explicit search tree. All in all, the discussed benefits of an explicit search tree structure outweigh the increased memory requirements.

12.7 Related Work

For software testing, symbolic execution trees describe possible execution paths of an imperative program under test [Kin76; MK09]. Similar to our search tree, a symbolic execution tree represents choice points where execution branches and collects path constraints. However, a symbolic execution tree usually describes the entire execution of an application. In contrast, our search tree for constraint-logic object-oriented programming describes the execution of specific application parts, namely the non-deterministic execution of a search region. Its leaf node types are tailored to describing the result (i. e., solutions or failures) of execution paths. Moreover, a symbolic execution tree is the result of performing depth-first search, whereas the dual trails of our search tree specifically support arbitrary traversal.

The idea of using an explicit data structure for non-deterministic computations in order to facilitate different search strategies is extensively used in functional logic programming [BHH04; HPR12]. In functional logic programming, search trees cover non-determinism of expressions, i. e., they encode alternatives for the values to which a pure expression can be evaluated. In contrast to that, constraint-logic object-oriented programming is non-deterministic in its execution behaviour, which includes side-effects

incurring during execution. Therefore, the present search tree structure has to encode alternative behaviour, including side-effects, in addition to final results. In addition to the representation usually used in functional logic programming, our representation includes node types for exceptions (as a different kind of solution) and unevaluated search trees. The latter types are a prerequisite for the on-demand construction of the search tree during search, which is innately given with the non-strict evaluation in functional logic programming.

An explicit data structure for representing a search tree structure has also been used in a monadic definition of constraint programming [SSW09]. In contrast to our work, it abstracts from side effects and asserts an ordering of subtrees. Another explicit search tree is used for implementing a domain-specific language (DSL) for probabilistic programming in OCaml [KS09]. As OCaml is strict, the on-demand characteristic of the search tree is modelled explicitly using lambda functions. Although OCaml is not purely functional, the authors disregard backtracking w. r. t. behaviour, modelling only non-deterministic results of pure expressions.

As an alternative to using an explicit search tree, the interface of the probabilistic DSL in OCaml has also been implemented by using continuation passing style and by using delimited continuations, i. e., using `shift` and `reset` [DF90]. Using continuations provides an implementation in a direct style and removes the run-time overhead of the search tree data structure. Therefore, implementing Muli by means of `shift` and `reset` is an interesting option for future work. In this case, however, monadic reflection (i. e., inspecting the search tree) is expensive, and its efficient implementation requires additional techniques [PK15].

The concept of trails has initially been adapted from the trail described for the Warren Abstract Machine (WAM) [War83] and has been extended towards dual trails for arbitrary execution state in [DK19b]. Dual trails facilitate their use for backtracking upwards along a search tree as well as for descending towards nodes that have been (partially) evaluated. For their duality, the two trails were originally termed trail and inverse trail. Here we call them backward trail and forward trail, respectively, in order to improve clarity regarding the direction in which they are used. Extending previous work, the present paper leverages dual trails for the implementations of search strategies other than depth-first search.

12.8 Conclusion and Future Work

Our search tree structure represents the paths of non-deterministic execution of a search region. A runtime environment of a constraint-logic object-oriented language can construct the search tree non-strictly while executing a search region, thus encoding the solutions that are found as well as the execution behaviour of imperative code that leads to solutions or intermediate choices. As a result, the explicit search tree representation can serve several purposes. First, it provides a structure that arbitrary search strategies utilise for traversing the search tree. Furthermore, we found it to make debugging of non-deterministic execution behaviour more productive by allowing developers who use a debugger to introspect intermediate state at breakpoints. More opportunities for utilising the search tree in constraint-logic object-oriented programming will be part of future work.

We also extend Muli's runtime environment, the MLVM, to implement depth-first search, breadth-first search, and iterative deepening depth-first search. Even though they are well-known as search algorithms for tree traversal, they are of particular interest in the context of constraint-logic object-oriented programming where the search tree is not (entirely) known before the program that it represents has been executed in its entirety. The MLVM already supported depth-first search using the previous, unstructured approach, but our evaluation demonstrates that using a structured approach does not add any overhead. On the contrary, the explicit representation provides opportunities for novel search algorithms that could not be used for executing constraint-logic object-oriented programs prior to our work. The modifications have already been integrated into the open source MLVM and are available at `https://github.com/wwu-pi/muli`.

The current work is the basis for future endeavours. The search tree structure could be used for implementing an interactive search strategy in which a developer could manually decide how to explore the search space when a choice is encountered. This interactivity could be an additional aid for debugging. Moreover, it is interesting to explore alternatives to explicit search trees, such as the use of delimited continuations for the implementation of non-deterministic execution.

Acknowledgements

The initial ideas that led to this work were conceived during the first author's visit to the University of Kiel. The authors appreciate the valuable input of those that participated in the discussions; in particular, Sandra Dylus, Jan Christiansen, Jan Rasmus Tikovsky, and Michael Hanus.

References

[BHH04] Bernd Braßel, Michael Hanus and Frank Huch. 'Encapsulating Non-Determinism in Functional Logic Computations'. en. In: *Journal of Functional and Logic Programming* (2004), p. 28.

[Dag19] Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *WFLP 2018*. Ed. by J. Silva. Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8.

[DF90] Olivier Danvy and Andrzej Filinski. 'Abstracting Control'. en. In: *Proceedings of the 1990 ACM Conf. on LISP and Functional Programming - LFP '90*. ACM Press, 1990, pp. 151–160.

[DK18] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declare 2017*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DK19a] Jan C Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

[DK19b] Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proc. 34th SAC*. Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912.

[HPR12] M. Hanus, B. Peemöller and F. Reck. 'Search Strategies for Functional Logic Programming'. In: *Proc. ATPS'12*. GI LNI 199, 2012, pp. 61–74.

[Kin76] James C. King. 'Symbolic execution and program testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[KS09] Oleg Kiselyov and Chung-chieh Shan. 'Embedded Probabilistic Programming'. en. In: *Domain-Specific Languages*. Ed. by Walid Mohamed Taha. Vol. 5658. Springer, 2009, pp. 360–384.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 03/05/2019).

[MK09] Tim A. Majchrzak and Herbert Kuchen. 'Automated Test Case Generation Based on Coverage Analysis'. In: *TASE 2009*. IEEE, 2009. ISBN: 978-0-7695-3757-3. DOI: 10.1109/TASE.2009.33.

[PK15] Atze van der Ploeg and Oleg Kiselyov. 'Reflection without Remorse: Revealing a Hidden Sequence to Speed up Monadic Reflection'. In: *ACM SIGPLAN Notices* 49.12 (2015), pp. 133–144.

[SSW09] T. Schrijvers, P. Stuckey and P. Wadler. 'Monadic constraint programming'. In: *JFP* 19.6 (2009), pp. 663–697. ISSN: 0956-7968. DOI: 10.1017/s0956796809990086.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set*. Tech. rep. Menlo Park: SRI International, 1983.

13

Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space

Jan C. Dageförde* · Herbert Kuchen*

Citation Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing*. Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912.

Abstract Constraint-logic object-oriented programming facilitates the development of applications that occasionally solve search problems. To that end it is possible to define constraints imperatively on-the-fly, using conditions over free variables. As a result, the search space is defined dynamically at runtime, based on the evaluation of such conditions in control structures. Consequently, the search space can – intentionally or by accident – become infinitely large if recursion and/or loops are involved. It is therefore desirable that the corresponding encapsulated search can be controlled by application developers. To that end, we discuss mechanisms for encapsulating such a search space and retrieving solutions from it. Our implementation for Muli is based on an appropriately extended symbolic Java virtual machine and it leverages the Java Stream API to provide a control mechanism that integrates well with object-oriented programs. Moreover, we show how our language Muli behaves for some example programs.

Keywords Infinite search space · dynamic solution retrieval · constraint-logic object-oriented programming · multi-paradigm languages · virtual machine implementation.

*University of Münster, Germany

```
public static void powersOfTwo() {
  Solution[] powers = Muli.getAllSolutions(() -> {
    return twoPower(0); });
  for (int i = 0; i < 10; i++)
    System.out.println(powers[i].value); }
public static int twoPower(int y) {
  boolean coin free;
  if (coin) return Math.pow(2, y);
  else return twoPower(y+1); }
```

Listing 13.1: Muli search region generating 2^y for all integer $y \geq 0$.

13.1 Motivation

Muli (Münster Logic-imperative Language) augments the object-oriented (OO) programming language Java by features from constraint-logic programming, namely logic variables, constraints, and encapsulated search. By running Muli applications on a specialised symbolic Java virtual machine (SJVM), Muli facilitates the development of business applications that occasionally require search involving constraints that are dynamically added at runtime, e. g. as found in logistics. In Muli, search problems are defined in so-called *search regions.* These are methods (also implemented in Muli, i. e. augmented Java) which are symbolically and non-deterministically executed by the SJVM [DK18a], thus exploring the search space.

However, as a result of symbolic execution of an OO (imperative) program, the complete search space is only known after fully evaluating all alternative execution paths [Kin76]. Consequently, it is possible that the search space comprises an infinite number of solutions. In contrast, a typical program requires only a few solutions. For example, consider the program in Listing 13.1 that intends to use the first ten solutions of a constraint-logic program that computes 2^y for all $y \geq 0$ that are integers.[25] To that end, the runtime environment first has to find all solutions, which is not possible as the search space is infinite. But even if the number of solutions were finite, it could still be practically infeasible to explore the entire search space of a search region if its search space is large. As a result, a more sophisticated approach is required for performing non-deterministic search in constraint-logic OO programming.

So far, no such approaches exist. Therefore, our research sets out to develop methods that facilitate the retrieval of individual solutions in constraint-logic OO programming,

[25]The program is explained in detail in Section 13.3.

even given an infinite search space. This paper presents the results as follows. Section 13.2 explains the syntax and core concepts of Muli. Subsequently, Section 13.3 details the problem at hand and Section 13.4 discusses methods that implement search for non-deterministic constraint-logic OO programs. Section 13.5 describes the implementation of one of these methods in the Muli runtime environment. The changed runtime environment is then evaluated in Section 13.6. Section 13.7 relates our work to existing literature, followed by concluding remarks in Section 13.8.

13.2 Constraint-Logic Object-Oriented Programming

Muli is an extension of Java, integrating features of constraint-logic programming into the language. To that end, it allows defining *logic variables* and fields using the `free` keyword [DK18a], e. g.,

`int x free`.

Constraints are defined by formulating control structure conditions that involve logic variables, such as

`if (x > 5)`,

where `x` is a logic variable. A specialised runtime, the SJVM, performs symbolic evaluation of expressions and statements. If conditions of control structures involve variables that are (at runtime) free or insufficiently constrained to decide whether a condition evaluates to `true` or `false`, all applicable execution branches are executed non-deterministically [DK18b]. To that end, one branch is selected and a corresponding constraint is imposed on the SJVM's constraint store. That branch is then executed. Subsequently, backtracking occurs in order to ensure that all applicable branches are executed systematically.

In order to make the effects of non-deterministic execution manageable in a Java-oriented context, non-deterministic execution only happens within *encapsulated search.* Outside encapsulated search, execution is deterministic, but may involve the construction of symbolic expressions. Encapsulated search executes a *search region*, i. e. a method that formulates the search problem. The implementation of search regions as methods allows arbitrarily mixing imperative statements with manipulations of the constraint store, and facilitates their reuse across the program. Furthermore, encapsulation of search ensures that the overall application terminates in a single consistent exit state. Otherwise, if non-determinism were allowed outside of encapsulation, the program would yield one exit state per leaf of the symbolic execution tree. Instead, a symbolic execution tree only

exists for the encapsulated execution of search regions. There, every leaf of the symbolic execution tree corresponds to one *solution* of encapsulated search.

Found solutions are returned to the (deterministic) invoking program so that it can work on them. Encapsulated search is started by invoking operators that accept a search region and define the number of solutions to be returned. Specifically, `getOneSolution` terminates non-deterministic search after the first solution is found and returns that, whereas `getAllSolutions` first explores the entire search space and collects all solutions afterwards. For instance, the program in Listing 13.1 uses `getAllSolutions`, causing Muli to collect solutions in an array. Execution of the invoking program continues after the search space has been explored exhaustively.

13.3 Infinite or Large Search Spaces

As an example for a search region with an infinite number of solutions, consider the program `powersOfTwo()` in Listing 13.1 that generates all powers of two. `powersOfTwo()` creates a search region by defining a lambda expression that invokes `twoPower(int y)`, passing the constant 0 as the initial parameter value. `twoPower()` declares a free boolean variable `coin`. By branching over the variable `coin` in the condition of `if`, the search region simulates a coin flip as the condition can be both `true` or `false`. In the `true` branch, 2^y is computed and returned; otherwise `twoPower()` is called recursively, with its parameter incremented by one. Encapsulated search is started by invoking `getAllSolutions` with the search region as a parameter, which causes the SJVM to evaluate the search region non-deterministically. Afterwards, solutions will be returned by the SJVM as an array of type `Solution[]`, of which each element encodes one solution. This array is used by the remaining (deterministic) program.

At each recursive invocation, a fresh local `coin` variable is created. Consequently, execution is branched at every invocation. The result of non-deterministic evaluation of this search region is an infinite symbolic execution tree (cf. Figure 13.1; subtree of infinite height abbreviated by '...'), corresponding to an infinite amount of solutions. Effectively, `getAllSolutions` will never terminate for this search region, except if memory is exhausted (which will terminate the entire application in an error state).

Prior to this work, if more than the first solution was needed by a program, the SJVM was required to compute all solutions of a search region before returning them to the invoking program for two reasons. First, the entire search space is not known before the symbolic execution tree is fully evaluated [Kin76]. Second, backtracking is local [DK18a].

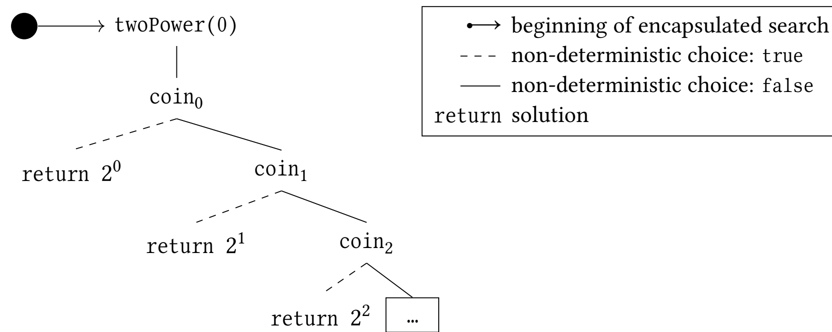


Figure 13.1: Simplified symbolic execution tree for the program in Listing 13.1 with conditional branching.

Consequently, `getAllSolutions` only terminates if the search region has a finite number of solutions. As a result, the `getAllSolutions` operator falls short for search regions that exhibit an infinite search space, particularly if a program intends to examine only a few solutions from encapsulated search, e. g. using an external system (and only resumes search to obtain *additional* solutions if the previous ones are found to be insufficient). This behaviour is consistent with Prolog [Wie03], whose `findall` predicate will not terminate if there is an infinite number of solutions to a goal. Similarly, Curry provides `allValues` methods to traverse a search tree. However, as the Curry language is lazy, the `allValues` methods work well even with large or infinite search spaces even though no mechanism is implemented to handle this problem specifically. However, constraint-logic OO programming is not lazy per se. Therefore, a mechanism is needed to achieve retrieving individual solutions lazily from a constraint-logic OO problem.

Prior to our work, the Muli API as used in Listing 13.1 defined the set of all solutions to be returned as an array and was therefore not suited, unless a program actually required all solutions (and the set of solutions is finite). Alternatively, Muli offers `getOneSolution` which explicitly aborts encapsulated search after one solution and returns that. Invoking `getOneSolution` a second time on a search region, however, will yield the same solution again, as the SJVM does not maintain state and is therefore unable to recognise a solution as being found already. Generally speaking, calculating only the first *n* solutions is trivial, as search can be terminated after *n* solutions have been obtained (or earlier, if the search space contains less than *n* solutions), but resuming search for further solutions is hard. Even though it would be possible to re-start search from the beginning and dropping the first *n* (previously found) solutions, the resulting re-computation of these *n* solutions is unnecessary and may also incur unwanted (repeated) side effects.

Choice point type	Triggering bytecode instruction
Floating point comparison	`FCmpg`, `FCmpl`, `DCmpg`, `DCmpl`
Long comparison	`LCmp`
`if` instruction, integer comp.	`If<cond>`, `If_icmp<cond>`
`switch` instruction	`Lookupswitch`, `Tableswitch`

Adapted from [DK18a]

Table 13.1: Subset of bytecode instructions that may introduce non-determinism if they involve logic variables. `<cond>` is one of `eq`, `ne`, `lt`, `le`, `gt`, or `ge`.

Since search regions are executed on the same virtual machine as program parts outside of them, a suitable mechanism is required for resuming search, i. e. for lazily retrieving individual solutions from encapsulated search, while at the same time retaining deterministic execution behaviour of the surrounding program parts. The subsequent section presents and discusses possible methods for traversal along the symbolic execution tree to facilitate non-deterministic search, accounting for the possibility that the symbolic execution tree may contain an infinite number of paths and that the program needs to be able to retrieve individual solutions from encapsulated search regardless.

13.4 Individual Retrieval of Solutions from Encapsulated Search

Traversing the symbolic execution tree requires a backtracking mechanism that is able to restore a previous state if the entire SJVM, so that it can resume execution from there. Similar to a regular JVM [Lin+15], the SJVM maintains a program counter and a stack of frames, where each frame maintains the values of its local variables and an operand stack [DK18a]. Additionally, to support constraint-logic OO programming, the SJVM maintains a constraint store, as well as a stack of choice points. Each choice point tracks where non-determinism was introduced, which choices have been taken, and the constraints that were added. On backtracking to a previous state, all these data structures (in the following subsumed as *SJVM state*) need to be reverted to their previous contents. Choice points are created whenever execution branches non-deterministically, i. e. by evaluation of only a few bytecode instructions that are depicted in Table 13.1.

Approaches that come to mind for achieving restoration of previous states are outlined in the following. They roughly fall into two fundamentally different categories, distinguished by their basic concept for describing the SJVM state(s). The trail-based

approaches leverage a trail data structure, adapted from the identically-named concept used in the Warren Abstract Machine [War83], describing individual state changes that have been applied to the SJVM during execution. In contrast, the copy-based approaches store copies of the entire state at pre-defined points during the execution, instead of tracking individual changes.

13.4.1 Copy-Based Backtracking

Serialisation One option is to introduce explicit savepoints at specific points during application execution, similar to transaction management of relational database management systems. In the *serialisation* approach, whenever a savepoint is reached, the SJVM state data structures are serialised and stored in a location separate from SJVM state, e. g. in files on disk. These stored data structures are referred to as *snapshots* in the following.

Savepoints need to be introduced at the beginning of encapsulated search, and wherever non-deterministic choice occurs. For example, for the application in Listing 13.1 savepoints s_1, s_2, ... need to be added when the conditional expression is evaluated, which will introduce non-determinism as it involves a free variable. Another savepoint s_0 is added at the beginning of encapsulated search, i. e. the *root*. Figure 13.2 augments the symbolic execution tree from Figure 13.1 with savepoints accordingly.

The implementations of bytecode instructions that introduce non-determinism (cf. Table 13.1) would need to be changed in the SJVM in order to support savepoints. To that end, the implementations of each of these bytecode instructions would first create a snapshot of the current state and store it with the corresponding savepoint for later retrieval, together with the selected choice. Afterwards, the instruction's effect is executed on the SJVM as before, followed by subsequent instructions of the program.

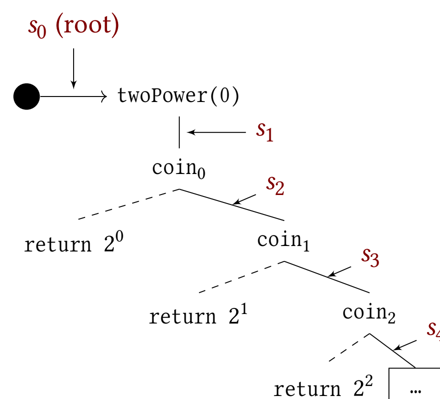


Figure 13.2: Placement of savepoints within the symbolic execution tree.

Once the SJVM reaches a leaf of the symbolic execution tree, i. e. finds a solution, the return value (or exception) of the search region is wrapped in a `Solution` object and backtracking to the first savepoint s_0 occurs by deserialising the object structures corresponding to its state, thus discarding the current SJVM state. As a result, the SJVM is now in the state from when encapsulated search has begun, without the effects from the executed search region. However, instead of executing the next branch as before, encapsulated search terminates and returns the found solution to the surrounding application.

When the next solution is required, the SJVM creates a new savepoint s_c at the current point in the deterministic execution (not visible in the illustrations, as this can occur at any later time during deterministic control flow) and stores a snapshot. After that, the state of the SJVM is restored from a snapshot that corresponds to the most recent choice point that still has another choice. Based on the recorded choice that has been taken before, the next choice is selected. The effect of the new choice is then executed by executing the subsequent instructions. If the latest choice point of that search region does not offer a choice that has not been selected yet, the SJVM tries that choice point's parent choice point, and so on. Once the root is reached, that indicates that all choice points in the symbolic execution tree have been exhaustively tried, and that there is no further solution. In either case, the state from savepoint s_c is restored and, if there has been another solution, that is yielded to the program.

Overall, this approach is rather straightforward to understand, as well as to implement. However, there are issues with how serialisation is generally performed in OO languages. All classes have the opportunity to override deserialisation of objects of their type so that, for instance, no new instances can be created for singleton classes on deserialisation. As intentional as that may be for regular applications, this implies that the state of singleton classes is not reverted to a previous state on deserialisation; instead the deserialisation method might return the corresponding singleton object in its most recent, now invalid, state. As a result, the resulting SJVM state becomes inconsistent on backtracking if such classes or similar overrides to their deserialisation method are involved, thus rendering this approach unreliable. Moreover, this approach falls short if the SJVM state's object graph involves classes that are not serialisable, which is likely the case for arbitrary applications. Another major disadvantage is that each savepoint requires a full snapshot of the entire SJVM state, regardless of how much of it is changed during snapshots, and of whether a snapshot of a savepoint is actually used at a later point in time. Therefore, the amount of memory required depends on the data structures managed by the application,

multiplied with the number of choice points times their respective choices. This results in high memory requirements for this approach.

Object cloning Another option relies on the same savepoints as the serialisation approach. The *object cloning* approach is generally similar, however in contrast to that, the state is not serialised. Instead, snapshots are created by generating clones of the SJVM state data whenever a savepoint is encountered. The snapshots are kept in a map, recording snapshots for each savepoint. That map is used on backtracking in order to obtain an earlier snapshot of the SJVM state. The SJVM needs to ensure that the map of SJVM state clones is not stored as part of the SJVM state itself, as otherwise it would be modified on backtracking as well.

A virtual machine implementation in Java can leverage the fact that every class in Java inherits a `clone()` method from `java.lang.Object` that it can override. However, the default implementation only creates a shallow copy of an object [Ora18a]. This is problematic because, if an object that references another (or more) object(s) is cloned using that implementation, the clone will reference the exact same object. In contrast, a deep copy has to be created in order to copy the state consistently, i. e. the entire object graph needs to be copied in order to ensure that snapshot clones are not changed by later evaluations.

Consequently, all objects that occur in the object graph need to override the `clone()` method accordingly so that a deep copy can be obtained. This is unlikely, therefore serialisation followed by immediate deserialisation can be used as a workaround to achieve a deep copy. Alternatively, a virtual machine could create a deep copy using a reflection-based approach [Kou18]. However, this is not supported by all classes either, e. g., creating a clone of a file stream will crash the JVM.

As a result, this approach falls short in several ways, depending on the chosen implementation alternative. Some disadvantages are shared with the serialisation approach. Classes might implement the clone, serialisation, and/or deserialisation methods in ways that prevent deep copies from being created. Therefore, changes made to objects during current execution might have unexpected side effects on snapshots. Alternatively, if reflection is used the use of classes that do not support deep copying via reflection is forbidden implicitly. As cloning such classes results in a crash of the JVM, there is no opportunity to recover from that. Moreover, this approach potentially requires a lot of memory for maintaining all snapshots. In fact, that problem might be even more relevant here, as snapshots are now stored in main memory.

Trail element	Affects state of	Parameters	Inverse element
`PCChange`	Frame	pc	`PCChange`
`Restore`	Frame	variable index, value	`Restore`
`FrameChange`	Frame stack	frame	`FrameChange`
`VmPop`	Frame stack		`VmPush`
`VmPush`	Frame stack	value	`VmPop`
`Pop`	Operand stack		`Push`
`Push`	Operand stack	value	`Pop`
`FieldPut`	Heap	instance, field, value	`FieldPut`

Table 13.2: Trail elements, representing inverse operations to reverse previous SJVM state changes in trail-based backtracking approaches.

13.4.2 Trail-Based Backtracking

The Muli SJVM maintains a stack structure for each choice point, the so-called trail, on which operations are recorded that need to be performed in order to reverse changes that instructions effected on the SJVM state. During encapsulated search, whenever an instruction is executed that changes the SJVM state, an object representation of the inverse of that change is instantiated, which is referred to as the *trail element*. A choice point only contains trail elements collected since a choice was made that results from that choice point, whereas previously collected elements are stored with parent choice points.

A trail element comprises an operation on virtual machine state and, if applicable, parameters that specify details of the operation, such as a value or the name of an affected field. All possible trail elements and their parameters are listed in Table 13.2. As an example, executing the `iadd` bytecode instruction pops two elements from the current operand stack and pushes the result, either the constant sum [Lin+15] or a corresponding symbolic expression [DK18a], onto the operand stack. As an additional result, three trail elements are generated that correspond to the original operations changing the SJVM state. Two `Push` trail elements with the values that were on the stack originally are generated, as well as one `Pop` element to remove the result. Since constraints are only added whenever choice points make a choice, there is no explicit trail element for changes to the constraint store. Instead, when a choice point is backtracked, its constraint is removed from the constraint store implicitly.

On backtracking, trail elements are popped from the trail and the corresponding changes that they represent are effected on the SJVM, thus reverting the SJVM to its state prior to making the last choice. One trail is maintained per choice point, so that all

actions taken after making a choice can be reverted by looking at that choice point's trail. As soon as the trail of a choice point is empty and its constraint is removed, the next choice can be made by imposing the new constraint and setting the virtual machine's program counter for the next control flow branch.[26] Otherwise, the next choice of that choice point will be imposed or, if no more changes are left, further backtracking to its parent choice point occurs, now using the parent's trail.

Partial backtracking So far, Muli worked under the assumption that a program will require either one or all solutions. For that reason, backtracking in the original Muli implementation occurred locally. As an example, consider the symbolic execution tree in Figure 13.1. After a solution has been found in one of the `return` leaves, backtracking to the last choice point needs to occur to get to the next alternative path. To that end, the trail of that particular choice point is unrolled so that the changes that have lead to this leaf are reversed. Afterwards, the SJVM is in the exact state needed for following the next path. If that choice point does not provide another choice, its parent choice point's trail is unrolled analogously. Otherwise, the next path is evaluated. Still, backtracking happens only as far as required until the next choice can be made.

We refer to this behaviour as *partial backtracking*, because after backtracking the SJVM is still in a state corresponding to an inner node of the symbolic execution tree. Since encapsulated search relies on the execution of a method, that method's frame (and all frames directly or indirectly created by it) could be popped from the frame stack in order to return to deterministic control flow of the surrounding application. This way, the SJVM could end encapsulated search immediately and return an individual solution to the surrounding application.

However, this mechanism is only useful if search is not expected to be continued at a later point in time. Particularly, this generalises badly under the assumption that an arbitrary number of solutions will be queried. The underlying problem is that it is not possible to resume search in a state in which the next solution could be computed. After frames corresponding to search are popped from the stack, their former state is lost. As an ad-hoc mitigation, frames could be stored before popping them from the frame stack. However, the mechanism for storing frame state would be serialisation or a different form of deep copy. Therefore, this mitigation suffers from the same deficiencies as the copy-based approaches, so partial backtracking is insufficient for the problem at hand.

[26] In fact, after making a choice and imposing its respective constraint, Muli checks the resulting constraint system for consistency. As a result, Muli only follows the corresponding execution path if all imposed constraints are satisfiable [DK18b].

Full backtracking The problems of partial backtracking can be resolved by extending the approach. Instead of performing backtracking locally by using the trail of the current choice point only, *full backtracking* reverses all effects that encapsulated search has had on the state of the SJVM using all trails of the current choice point and its parents, until reaching the very beginning of encapsulated search. This way, the SJVM can cleanly return from encapsulated search and pass the value to the surrounding (deterministic) application, thus allowing regular execution to continue. Consequently, in order to be able to evaluate a subsequent choice later, the SJVM has to be able to restore the state that is valid right before making that choice, i. e. the state that the SJVM would be in after partial backtracking reaches a choice point that offers another choice. To that end, the SJVM is augmented by two additional data structures.

The first is a *pseudo choice point* that marks the root of a symbolic execution tree. We call it pseudo, because it always offers only one 'choice'. Instead, its purpose is to record a trail for all SJVM state changes performed from the beginning of encapsulated search to the first actual choice caused by non-deterministic evaluation. This is necessary because the SJVM maintains trails per choice point. Consequently, the SJVM will be able to reverse the entire effects of evaluation within encapsulated search, so that the SJVM is in a clean state to continue evaluating the invoking application.

The second data structure is an *inverse trail*. Like the trail, the inverse trail is a stack structure that records operations necessary to restore a given SJVM state. However, it is not used for backtracking the effects of encapsulated search, but for restoring these effects in order to be able to resume search, i. e. to undo backtracking. Consequently, elements are not added to the inverse trail during execution of a search region. In fact, the inverse trail remains empty during encapsulated search. Elements are only added to the inverse trail during backtracking, directly corresponding to elements that are removed from the trail. Therefore, after backtracking the effects that followed making a choice, that choice point's trail stack is empty, whereas its inverse trail stack may be full. There is a bijective mapping of trail elements to their inverses (cf. Table 13.2), thus ensuring that for every trail the SJVM is able to create a corresponding inverse trail. For example, backtracking the `iadd` instruction is performed by first executing the `Pop` element from a choice point's trail, thus removing the last operand e from the operand stack. Simultaneously, the corresponding inverse trail element `Push` is created, holding the operand e. The inverse element is pushed to the inverse trail.

Leveraging these additional data structures, the trail-based full backtracking method works as follows. After making a choice at a choice point, executing subsequent instructions results in changed SJVM state, as well as in elements recorded on the trail

that describe how to revert to the previous state. Upon finding a solution, trails of all choice points are used in order to obtain the SJVM state from before search has begun (in contrast to partial backtracking, which uses the most recent choice point's trail only). The trail of the most recent choice point is only relevant for the solution that has just been found, so that the SJVM will not need to be able to revert to that state anymore as no additional solution can follow from that state. Therefore, during backtracking of this choice point, no elements are added to the inverse trail. Instead, the choice point is marked to proceed with the next choice when it is encountered next. For the remaining choice points until the root of the symbolic execution tree, elements are added to their respective inverse trails during backtracking: For each element popped from the trail stack, the corresponding effect is performed on the SJVM, and an inverse trail element that describes how to reverse this effect is pushed on the inverse trail stack.

Maintaining an inverse trail ensures that, when effects of search are later restored using the inverse trail, the underlying bytecode does not have to be executed again to reach a specific state. Instead, the relevant consequences of its computations (results as well as side effects) are already captured by the (inverse) trail elements and can be obtained from there. This is illustrated in Figure 13.3, where backtracking pops a symbolic expression $5+y$ from the operand stack, while creating a corresponding `Push` trail element on the inverse trail in order to prepare for restoring the former state later on. As a result, the use of the inverse trail enables the SJVM to undo the effects of backtracking until reaching the next relevant choice point. At the same time, after backtracking the effects of encapsulated search and before resuming search for a search region by unrolling its inverse trail, the SJVM is in a clean state that allows deterministic computation to examine the found solutions or to perform unrelated tasks.

Advantages of the full backtracking method include that an established structure of the Muli SJVM, the trail, can be re-used for implementing the inverse trail. Moreover, the operations that are needed to restore state using the inverse trail are identical to those used by the regular trail. There is a bijective mapping from a trail element to its inverse, thus ensuring that for every trail there is a unique inverse trail and vice versa.

In order to maintain the inverse trail, the full backtracking method requires memory corresponding to the number of executed instructions within a search region. However, the required memory does not exceed the amount of memory that was required for the regular trail in the first place, since the inverse trail only holds elements that are inverses to those that have been on the trail prior to backtracking. Furthermore, only one sequence of trails and inverse trails representing a single path through the symbolic execution tree is required per search region in order to be able to reach previous states from which

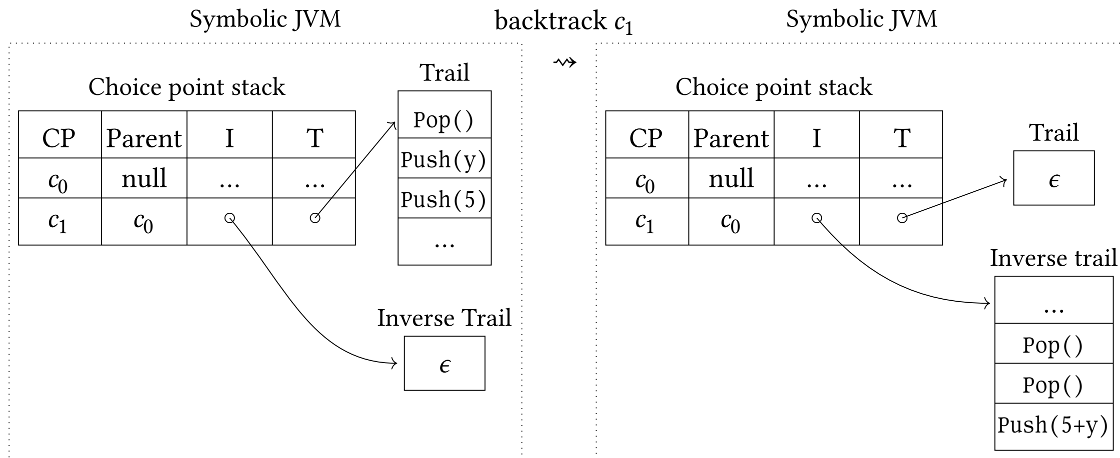


Figure 13.3: Effects on trail and inverse trail resulting from backtracking a choice point.

execution can continue. In contrast, the copy-based approaches require maintaining multiple copies corresponding to several savepoints along the symbolic execution tree.

13.5 Full Backtracking in the SJVM Using an Inverse Trail

For the Muli SJVM, the full backtracking method is preferable over the copy-based approaches as it is a straightforward extension of the existing Muli SJVM while providing a balance of memory requirements and reliability. Moreover, for the problem at hand it is more suited than the partial backtracking method, given that it facilitates fine-granular control of encapsulated search, allowing programs to compute and retrieve individual solutions for a given search region, regardless of the size of the search space. Therefore, we extend the Muli SJVM [DK18a] to implement a prototype of the full backtracking method, thus achieving support for the retrieval of individual solutions from encapsulated search, as well as offering an interface to resume search.

As a basis for defining a new interface to Muli's encapsulated search, the Java Stream API introduced in Java 8 comes to mind. In the Stream API, a stream may be a (potentially) infinite sequence of elements that are not evaluated unless consumed individually [Ora18b]. Each element of a stream can be consumed exactly once, and may be computed on an on-demand basis, i. e. as soon as an element is required by a terminal operation. Consequently, the stream is allowed to comprise an infinite number of elements. Moreover, as soon as an element is consumed, the stream is not expected to be able to produce it again. Additionally, the API does not prescribe how stream elements are

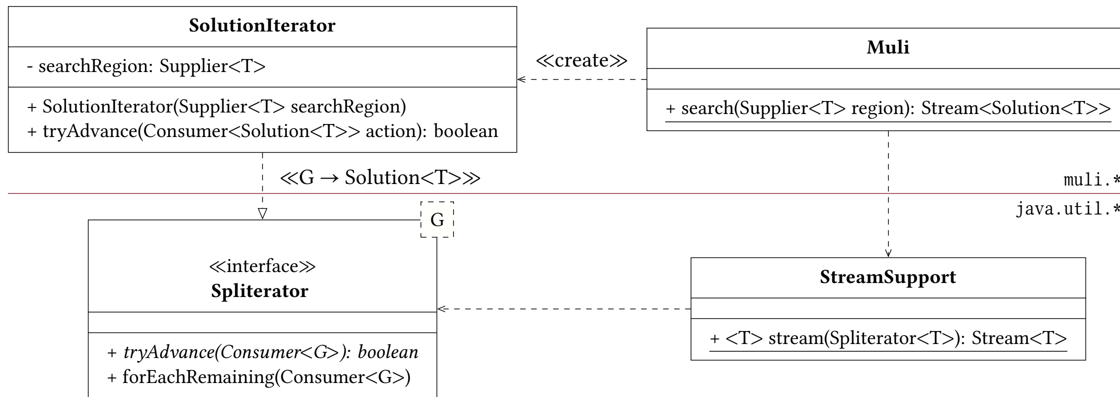


Figure 13.4: UML class diagram conceptualising the relationships between Java Stream API and the Muli implementation for retrieving individual solutions from encapsulated search.

obtained, i. e. a stream implements the iterator pattern [Gam+95]. All things considered, this interface is very suitable for accessing the full backtracking method for encapsulated search. In order to accommodate for the Java Stream API, the goal is to find a suitable implementation for a new encapsulated search operator `search()` that accepts a search region as a parameter and returns a stream of `Solution` objects.

The search region is passed to `search()` as an implementation of the functional interface type `Supplier<T>`, where `T` represents the type of individual solutions returned by the search region (cf. Figure 13.4). A search region implicitly implements this functional interface type if it is in the form of a lambda expression or a method reference, as long as the lambda or the referenced method do not accept parameters and return a value of type `T`. The runtime environment wraps a found value (or an exception that occurred at runtime) in objects of type `Solution<T>`. The return type of `search()` is `Stream<Solution<T>>`, i. e. an object representation of the stream whose elements are lazily computed by encapsulated search as soon as they are required by a terminal operation.

For the purpose of constructing the stream, `search()` uses a helper method in `java.util.stream.StreamSupport` that accepts an implementation of an iterator and converts it into a stream. This iterator must implement the generic interface `java.util.Spliterator<G>`, where `G` represents the type of returned elements, here `Solution<T>`. This interface is implemented in Muli by the class `SolutionIterator<T>`, which is responsible for controlling encapsulated search in the SJVM. Control logic that starts encapsulated search and (on subsequent invocations on the same search region) reverts SJVM state is implemented in the method `tryAdvance` of `SolutionIterator`, which is called indirectly from a consuming stream operation.

Adhering to the interface prescribed by `Spliterator`, the implementation of `tryAdvance` returns **`false`** if no additional solution has been computed, i. e. if no choice point of the corresponding search region offers another choice, or **`true`** if a solution has been computed. In case of the latter, the found solution is passed to the `accept` method of the consumer that needs to be passed to `tryAdvance`. More precisely, `tryAdvance` performs the following steps, given a consumer `c` and a search region `r`. First, it changes the SJVM's execution mode so that non-deterministic computations are allowed and sets this iterator's search region to be the active one in the SJVM. This is required to assign trail elements created during execution to the correct choice point. Second, the method either (before computing the first solution) creates a pseudo choice point to mark the root of encapsulated search and invokes `r` to compute the first solution, or (before computing subsequent solutions) instructs the SJVM to revert the state to that of the current choice point using the inverse trail and to calculate an additional solution. Third, as soon as a solution is found, `tryAdvance` instructs the SJVM to wrap it into a `Solution` object `s` and to perform full backtracking. Fourth, it reverts the SJVM's execution mode and active search region to those prior to this search. Finally, it invokes `c.accept(s)` in order to pass the found solution on to the next operation of the stream.

Inside the SJVM, the existing backtracking mechanism is modified to accommodate for the inverse trail. So far, the mechanism for backtracking operated locally, i. e. only on a choice point's trail stack, iterating over its elements and thus reversing the changes to the SJVM state. The new implementation extends this mechanism to operate on both trail and inverse trail. As a result, backtracking can now be executed in three modes: First, `SimpleRestore` effectively uses the former backtracking mechanism, i. e. it processes the trail without creating inverse elements. This is used when processing the trail to backtrack to the most recent choice point. Second, `TrailToInverse` processes the regular trail and pushes corresponding inverse elements to the inverse trail. This mode is applied for backtracking to the remaining choice points until the root of symbolic execution, i. e. the pseudo choice point, is reached. Third, `InverseToTrail` is used for restoring the effects of encapsulated search from the inverse trail. Analogously, it restores a previous, mid-search state of the SJVM and pushes corresponding inverses to the regular trail, thus preparing for later backtracking.

Prior to this work, that mechanism was only invoked to process the most recent choice point, as it implemented the partial backtracking method. We extend the scope to process all choice points as follows. Backtracking begins in `SimpleRestore` mode with the most recent choice points until a choice point is reached that offers another choice. This is the state that needs to be restored when resuming search later. From that choice point

```
1 public static void powersOfTwo() {
2   Stream<Solution<Integer>> powers =
3     Muli.search(() -> {
4       return twoPower(0); });
5   powers.limit(10).forEach(System.out::println); }
```

Listing 13.2: Modified `powersOfTwo` method using the new interface. Guaranteed to terminate after computing at most 10 solutions.

on, backtracking continues in `TrailToInverse` mode until reaching the root choice point. This mechanism is invoked by `tryAdvance` after a found solution is wrapped. Moreover, we implement an additional method in the SJVM that is invoked by `tryAdvance` to resume search. It uses the same backtracking mechanism, but in the `InverseToTrail` mode.

Since the new mechanism leverages Java's Stream API by offering an interface that implements `java.util.stream.Stream`, Muli programs can now control encapsulated search in the same way as they use other stream-based iterators on arbitrary data structures. In particular, this means that programs can invoke methods on the solution stream generated by encapsulated search using the new `search()` operator, such as `count`, `filter`, `skip`, `limit`, `map`, and `forEach`. These methods can be combined using the Stream API's fluent interface style. For example, Listing 13.2 exhibits a modified version of the program from Listing 13.1 that uses Muli's new API in combination with standard stream operations from Java. The modified example creates the solution stream by invoking `search()` using a search region, but search is not actually started at this point. Subsequently, the resulting stream is limited to at most 10 elements using the `limit` method. Actual elements are calculated as soon as the terminal operation `forEach` is invoked, which retrieves solutions from the stream individually.

13.6 Evaluation

Search problems tackled by Muli programs may exhibit a search space containing an infinite number of solutions. As in lazy functional languages, such a search space may have been created intentionally in order to avoid adding imperative, deterministic checks for termination conditions that would be detrimental to a rather declarative style. In any case, the runtime environment is not able to detect whether full execution can terminate and it must therefore be capable of retrieving individual solutions from the search space. Prior to our work, the runtime environment of Muli did not cope well

with such a situation, as it can either return the first solution of a specified search region only, or return all its solutions. In the latter case, the application would not terminate for search regions with an infinite search space, since the SJVM would not return from search before all solutions have been computed, which is trivially impossible in this situation. Our novel approach and implementation provide support for this. Applications are now able to evaluate a search region in a way that individual solutions can be retrieved, by facilitating to resume search in order to obtain additional solutions at any later point during runtime.

A direct comparison of the implementations in Listing 13.1 and Listing 13.2 demonstrates that the programming style that follows from using the Stream API is beneficial to the understanding of Muli source code. The example exhibits how post-processing of individual solutions can now be expressed elegantly using a functional style.

We have performed a set of experiments in order to conduct a quantitative evaluation of the modified runtime as well. Muli applications are executed on the Muli SJVM running in an OpenJDK JVM, version 1.8.0_171. Constraint solving is performed using a solver component in the SJVM that employs the JaCoP finite domain solver [Kuc03]. The experiments are conducted using Ubuntu 18.04 with a 4.15.0 x86_64 GNU/Linux kernel on an Intel Core i5-5200U CPU. Each experiment is executed 510 times in a row, from which the first 10 results are dropped in order to exclude results from before JIT compilation.

In a first experiment, we have evaluated how well the new approach supports the retrieval of individual solutions from encapsulated search of a problem with an infinite search space. To that end, we compared the execution of the program depicted in Listing 13.1 on the *SJVM prior to our work* to the execution of the program in Listing 13.2 on our *modified SJVM*. Both search regions solve the same problem and generate the same set of constraints on the SJVM during search. In both cases, we try to obtain 10 solutions from the search region and print them to the command line. As expected, using the old SJVM implementation the program never terminated within a pre-defined time limit of 5 seconds and did not print a single solution. Using the new SJVM implementation, the respective program terminated after 0.66 milliseconds on average, successfully printing the first 10 solutions.

In a second set of experiments, we have let our modified Muli SJVM solve three classic search problems.[27] We compare their execution times to those of implementations of identical problems in Prolog in two variations: Each problem is implemented once using the CLP(FD) package [Tri12], and once using pure Prolog only [Wie03]. The Prolog

[27]The source code of the example applications is available at `https://github.com/wwu-pi/muli/tree/master/examples/sac19`.

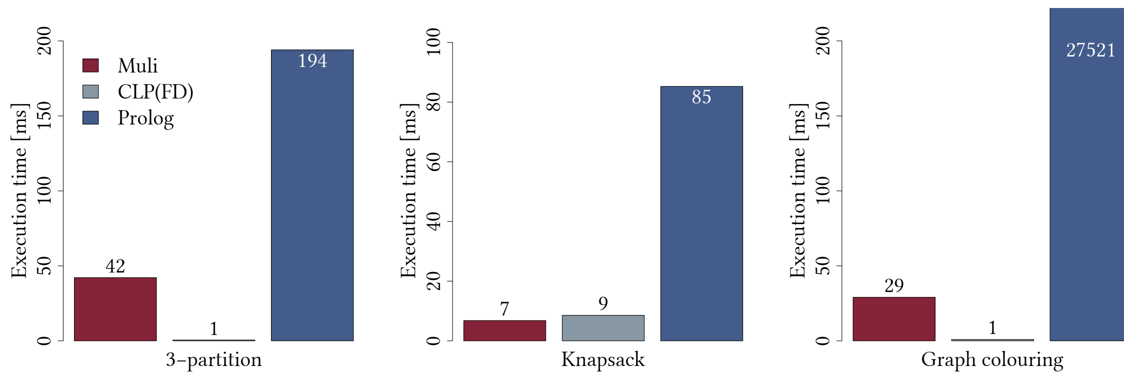


Figure 13.5: Comparison of execution time needed to solve search problems, each averaged over 500 executions.

applications are each executed 500 times using the 64-bit version of SWI-Prolog 7.6.4, on the same machine as the Muli applications.

For all three applications, the Muli implementations exhibit a strong advantage over the pure Prolog implementations, as execution takes drastically shorter (cf. Figure 13.5). This is likely the result of the constraint propagation abilities of the solver employed by the Muli SJVM, which allow the SJVM to rule out execution paths early if their sets of constraints are not satisfiable. In contrast, for most experiments the Muli application performs slightly worse than its CLP(FD) counterpart in Prolog. That was to be expected, given that our prototypical implementation of the SJVM has been implemented in Java, leading to one virtual machine running nested into another. This costs us about an order of magnitude. This overhead can be avoided by a low-level implementation once the concepts of Muli are stable and have shown to be useful. Having this in mind, the comparison to CLP(FD) is encouraging nonetheless. After all, in contrast to Prolog programs that use CLP(FD), Muli allows greater flexibility for expressing programs that contain search problems, with the option of mixing the OO and constraint-logic programming paradigms as needed. Moreover, in the case of the Knapsack problem, Muli is even better than CLP(FD). Here, encapsulated search is used to find a valid solution. This search is included in a loop which looks for better solutions in each iteration, thus combining an imperative style for defining the exit condition with the constraint-logic OO search for solutions. This behaviour can be elegantly implemented in Muli, whereas this is harder to achieve in Prolog (with or without CLP(FD)).

In addition to using these applications for execution time comparisons, the Muli applications exhibit some implementation details that demonstrate how constraint-logic OO programming can be useful to developers coming from an OO programming background. For example, the Knapsack application manages three arrays of the same length (`weights`,

`benefits`, and `amounts`), where each array element describes a different property of an entity that is consistently described by its index. Muli facilitates initialisation of these properties in an imperative style using **for** loops, e. g. for declaring each element of the `amounts` array free. Furthermore, developers are able to access and modify individual elements without requiring a recursive predicate iterating over all of them. While that may come natural to a Prolog programmer, it is strange for developers who are used to imperative programming. Additionally, the Graph colouring application demonstrates how constraint-logic OO search can include object representations by declaring a class `Edge` and using it inside the search region.

13.7 Related Work

Closest to Muli are approaches that extend OO programming or imperative programming with concepts from constraint-logic programming. Such approaches are presented in e. g. [DM03; MK11; TH08]. Neither of these approaches provides encapsulated search and hence the possibility to cleanly separate search from other computations. Moreover, none of the mentioned approaches integrate OO programming and constraint-logic programming as smoothly as Muli.

An alternative to using an integrated constraint-logic OO language is to call constraint-solver libraries from an OO language. Constraint solvers for Java are e. g. JaCoP [Kuc03] and OptaPlanner [The17]. Clearly, a seamless integration of OO and constraint-logic programming cannot be achieved this way. For instance, a close interaction between search and search control will not be possible.

Moreover, there are several approaches which add OO features such as inheritance to a (constraint) logic programming language, mostly to Prolog, e. g., Visual Prolog [Sco10], the work of McCabe [McC92], the approach by Shapiro and Takeuchi [ST83], Prolog++ [Mos94], and Mozart/Oz [Van+03]. All of these languages are *declarative* languages, which just provide syntactic sugar for OO concepts but do not integrate them directly. Their flavour is completely different from that of Muli. Assignments and state changes are not supported. Moreover, these languages do not provide encapsulated search.

tuProlog achieves the integration of Prolog and Java by providing a Prolog implementation written in Java [DOR05]. However, this results in applications implemented using two different languages. CAPJa combines Java and Prolog by facilitating the mapping of Java objects to Prolog terms, but distinct code in each language is required never-

theless [Ost15]. Consequently, both approaches offer neither seamless integration nor encapsulated search.

There are also approaches such as Scala [Ode+17] integrating functional and OO programming. Also, Java itself now offers support for writing lambda abstractions [UFM14]. Although functional programming is another declarative programming paradigm, these approaches are rather different from Muli as they do not provide built-in search mechanisms. However, these and other approaches demonstrate how software development benefits from mixed-paradigm languages.

The encapsulated search of Muli is similar to the corresponding concept provided by the functional-logic programming language Curry [AJ16; Bra+11; HKM95; LK99]. However, our implementation works differently, since the Curry approach (particularly the `try` operator) would not work properly in the presence of side effects. Particularly, when encountering non-deterministic choice during evaluation (cf. choice points in Muli), Curry's `try` operator stops evaluation at that point and returns a list of lambda abstractions representing partially evaluated results, where each element of the list represents one choice. As subsequent branches might rely on a given state (i. e. the one prior to making a particular choice), this approach only works if it is safe to assume that no side effects are incurred during non-deterministic evaluation. However, this assumption does not hold in Muli due to the nature of OO (imperative) programming, thus requiring the SJVM to evaluate an individual solution in full.

13.8 Conclusions and Future Work

We have provided a set of contributions. First, we have integrated encapsulated search into a constraint-logic object-oriented language and presented approaches to traverse the solutions of a search problem, in particular if that problem – expectedly or unexpectedly – has an infinite number of solutions. Second, we contribute a prototypical implementation of a suitable approach in the Muli runtime environment. The prototypical implementation is able to undo the effects of trail-based backtracking, thus facilitating the retrieval of individual solutions from encapsulated search over search regions, while allowing for arbitrary computations outside of encapsulation. As a result, we show that it is possible to lazily compute a stream of solutions in constraint-logic OO programming.

The trail-based full backtracking method allows runtime environment to compute solutions of search regions in encapsulated search, while allowing applications to retrieve individual solutions. As a result, solutions to given problems are computed on an on-

demand basis. This improves performance if only a subset of solutions is required (for search regions with finite search spaces) and allows encapsulated search to terminate even in the case of infinite search spaces. For example, this can be used in order to first obtain a small set of solutions and to evaluate them deterministically according to a separate target function, e. g. one that is supplied by an external system. If these solutions are considered insufficient according to that target function, encapsulated search can resume in order to determine further solutions.

To that end, the method augments Muli's trail concept used for backtracking by adding an inverse trail. On backtracking, this inverse trail is used to record operations that are necessary for restoring the state of the SJVM, i. e. to undo backtracking. That way, effects on the state resulting from non-deterministic computation can be reversed after one (or more) solutions have been found, thus facilitating deterministic computation, while still maintaining the ability to re-enter encapsulated search in its previous state in order to resume search for further solution This method has been found to perform well in our evaluations using a Muli runtime environment that was modified accordingly.

The presented method is prototypically implemented in the Muli SJVM, enabling Muli application developers to control encapsulated search and to obtain individual solutions. The resulting modified Muli SJVM is available on GitHub as free software.[28]

As a complement to our prototype, future work can consider implementing an alternative virtual machine using a language that offers immutable data structures in combination with copy-on-write, as this might be a way to achieve a reliable and efficient implementation of a copy-based method. For example, this could be done using Scala, Haskell, or Curry. Moreover, once concepts of Muli are stable, a low-level implementation of the runtime environment should be considered to improve execution performance. Furthermore, the Java Stream API is designed to facilitate parallel computation of stream elements, by splitting a stream and delegating the consumption of the resulting partial streams to multiple threads. We have disregarded (and disabled) parallel computation for now, because this feature is not mandatory for custom streams and because the SJVM is not capable of parallel execution. Nevertheless, future work could tackle increasing the performance of search by evaluating non-deterministic branches in parallel.

[28]Available at `https://github.com/wwu-pi/muli`.

References

[AJ16] Sergio Antoy and Andy Jost. 'A New Functional-Logic Compiler for Curry: Sprite'. In: *LOPSTR 2016*. 2016. DOI: 10.1007/978-3-319-63139-4_6.

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller and Fabian Reck. 'KiCS2: A New Compiler from Curry to Haskell'. In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: 10.1007/978-3-642-22531-4.

[DK18a] Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*. ACM, 2018, pp. 1185–1194. ISBN: 978-1-4503-5191-1/18/04. DOI: 10.1145/3167132.3167260.

[DK18b] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DM03] J. Doyle and C. Meudec. 'IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming'. In: *2nd PPPJ*. 2003, pp. 55–58. DOI: 10.1145/957289.957307.

[DOR05] Enrico Denti, Andrea Omicini and Alessandro Ricci. 'Multi-paradigm Java-Prolog integration in tuProlog'. In: *Science of Computer Programming* 57.2 (2005), pp. 217–250. ISSN: 01676423. DOI: 10.1016/j.scico.2005.02.001.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[HKM95] Michael Hanus, Herbert Kuchen and Juan Jose Moreno-Navarro. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Kin76] James C. King. 'Symbolic execution and program testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[Kou18] Kostas Kougios. *cloning: deep clone java objects*. 2018. URL: https://github.com/kostaskougios/cloning (visited on 06/12/2018).

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 10844309. DOI: 10.1145/785411.785416.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 09/06/2017).

[LK99] Wolfgang Lux and Herbert Kuchen. 'An Efficient Abstract Machine for Curry'. In: *Informatik '99*. Ed. by K Beiersdörfer, G Engels and W Schäfer. Springer Verlag, 1999, pp. 390–399.

[McC92] F. G. McCabe. *Logic and Objects*. Prentice-Hall international series in computer science. Prentice Hall, 1992. ISBN: 9780135360798.

[MK11] Tim A Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7.

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/ (visited on 13/06/2017).

[Ora18a] Oracle. *Class Object*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html (visited on 06/05/2018).

[Ora18b] Oracle. *Interface Stream<T>*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html (visited on 05/05/2018).

[Ost15] Ludwig Ostermayer. 'Seamless Cooperation of Java and Prolog for Rule-Based Software Development'. In: *Proceedings of RuleML 2015*. 2015. URL: http://ceur-ws.org/Vol-1417/paper2.pdf.

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010. ISBN: 9781432749361.

[ST83] Ehud Shapiro and Akikazu Takeuchi. 'Object oriented programming in Concurrent Prolog'. In: *New Generation Computing* 1.1 (1983), pp. 25–48. ISSN: 02883635. DOI: 10.1007/BF03037020.

[TH08] Nikolai Tillmann and Jonathan de Halleux. 'Pex: White Box Test Generation for .NET'. In: *2nd International Conference on Tests and Proofs*. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9.

[The17] The OptaPlanner Team. *OptaPlanner User Guide, Version 7.0.0.* JBoss. 2017. URL: https://docs.optaplanner.org/7.0.0.Final/optaplanner-docs/html%7B%5C_%7Dsingle/index.html.

[Tri12] Markus Triska. 'The Finite Domain Constraint Solver of SWI-Prolog'. In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316. DOI: 10.1007/978-3-642-29822-6_24.

[UFM14] Raoul-Gabriel Urma, Mario Fusco and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT: Manning Publications Co., 2014. ISBN: 9781617291999.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. DOI: 10.1017/S1471068403001741.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set.* Tech. rep. Menlo Park: SRI International, 1983.

[Wie03] Jan Wielemaker. 'An Overview of the SWI-Prolog Programming Environment'. In: *Workshop on LP Environments*. 2003.

14

Reference Type Logic Variables in Constraint-logic Object-oriented Programming

Jan C. Dageförde*

Citation Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming*. Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8.

Abstract Constraint-logic object-oriented programming, for example using Muli, facilitates the integrated development of business software that occasionally involves finding solutions to constraint-logic problems. The availability of object-oriented features calls for the option to use objects as logic variables as well, as opposed to being limited to primitive type logic variables. The present work contributes a concept for reference type logic variables in constraint-logic object-oriented programming that takes arbitrary class hierarchies of programs written in object-oriented languages into account. The concept discusses interactions between constraint-logic object-oriented programs and reference type logic variables, particularly invocations on and access to logic variables, type operations, and equality. Furthermore, it proposes approaches as to how these interactions can be handled by a corresponding execution environment.

Keywords constraint-logic object-oriented programming · multi-paradigm languages · free objects · object type constraints.

*University of Münster, Germany

14.1 Motivation

Constraint-logic object-oriented programming can be used to develop business software that involves finding solutions to constraint-logic problems in an integrated way, particularly for applications that add constraints dynamically during runtime. The mixed paradigm leverages benefits of well-known object-oriented programming languages as well as of constraint-logic programming. For example, the constraint-logic object-oriented programming language Muli augments Java with logic variables, symbolic execution, constraints, and encapsulated search using a customised symbolic Java virtual machine (SJVM) [DK18a].

So far, symbolic expressions in Muli can involve logic variables of any type, but constraints can only be defined over (logic) variables of *primitive* types [DK18b]. While those variables may be fields of objects, thus proving useful in an imperative context as well as in an object-oriented one, such constraints are not applicable to entire objects. Similarly, the semantics of further interactions (particularly invocations and field accesses) with unbound reference type logic variables is not defined yet. After all, objects in object-oriented languages usually do not just encapsulate data, but behaviour as well. As a result, such interactions lead to interesting behaviour, e. g., when methods are invoked on unbound logic variables or objects are compared for equality. In order to realise the benefits of an integrated programming language, the expected behaviour of such interactions needs to be defined and implemented.

Consider the following case that will be used as a running example. We have an object-oriented representation of shapes, namely `Rectangle` and `Square` that both implement an interface `Shape` (cf. Figure 14.1), assuming integer edge lengths in millimetres. Implementations of `Shape` provide an appropriate method `getArea()` that calculates the area from field values of an object, as well as a method `toString()` that outputs the object's field values in a human-readable form.[29]

As a simple example, Listing 14.1 formulates a constraint to search for arbitrary shapes that have an area of 16 square millimetres. No specific instance is provided for `s`; instead, `s` is declared as a logic variable. On invocation of either `getArea()` or `toString()` on `s`, the execution environment has to consider that multiple implementations of these methods are applicable, as per the definitions depicted in Figure 14.1. In Muli, we expect the applicable alternatives to be evaluated non-deterministically until all alternatives are

[29]Even though `toString()` is not declared explicitly in the given interface, the Java language specification implicitly augments interfaces with abstract methods that correspond to every method that is declared in `java.lang.Object` [Gos+15, § 9.2]. Among others, this includes an implicit declaration of `toString()` that is consistent with the corresponding declaration in `Object`.

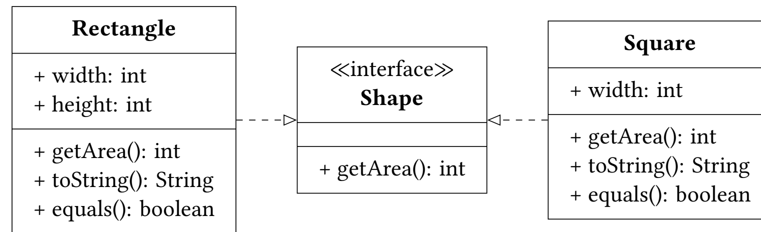


Figure 14.1: Class structure assumed for the running example.

```
1 Shape s free;
2 if (s.getArea() == 16) {
3   System.out.println(s.toString()); }
4 else { Muli.fail(); }
```

Listing 14.1: A constraint-logic object-oriented program that involves a free object.

considered [DK18a] ("don't know" non-determinism), here resulting in at least two output lines, namely one per actual type of s. Among other things, this paper will elaborate and discuss where exactly non-determinism can be introduced during the evaluation of this example and similar programs.

This paper contributes a concept for reference type logic variables in the context of constraint-logic object-oriented programming. To that end, all types of interactions of a program with reference type logic variables are discussed based on the example of Muli. This takes peculiarities of comparing equality of Java objects into account. For each possible interaction, this paper defines the expected behaviour and outlines approaches for handling it in the context of arbitrary object graphs. These approaches account for varying positions of objects' types in the class hierarchy that result from inheritance and implementation relations between classes.

This paper presents the contribution as follows. Section 14.2 provides a brief introduction to the constraint-logic object-oriented programming language Muli. Afterwards, Section 14.3 discusses interactions and explains how they can be handled. Furthermore, that section introduces constraints that are necessary to achieve these interactions. As this is a report on research in progress, Section 14.4 presents an initial implementation idea for a prototype that is going to be used for evaluation. Related research is outlined in Section 14.5. Finally, Section 14.6 summarises the contribution and provides an outlook.

```
int x free;
int i = 2, j = 3;
int y = i + j; // y == 5.
int z = x + y; // z == x + 5.
```

Listing 14.2: Arithmetic expressions containing bound or unbound variables.

14.2 Constraint-Logic Object-Oriented Programming with Muli

As a constraint-logic object-oriented language, Muli allows developers to use programming styles of object-oriented programming, while facilitating the specification of constraint-logic problems and finding solutions to them in the same language [DK18a]. Muli syntax is based on Java 8. The SJVM serves as the execution environment that supports logic variables by means of symbolic execution and leverages a constraint solver to solve constraint-logic problems. Compared to Java, the syntax extension is minimal and limited to the `free` keyword. It occurs in declaration statements to indicate an unbound ("free") variable:

```
int x free;
```

At runtime, free variables of primitive types are treated as logic variables to be used as part of symbolic expressions. Similarly, free objects can be defined, but their semantics is undefined and the execution environment does not provide an implementation for treating such variables yet. Therefore, the following code compiles but invoking the method in the second line will fail:

```
Object o free;
o.toString();
```

All variables, including unbound ones, can be used in boolean or arithmetic expressions in the same way as in Java. However, if an expression contains unbound variables, they cannot evaluate to a specific value. Therefore, the execution environment treats those variables symbolically and creates a symbolic expression [DK18b]. For instance, after executing Listing 14.2, `y` holds the constant value `5` (as expected in Java), whereas `z` holds the symbolic expression `x + 5`.

Ultimately, symbolic arithmetic expressions can evaluate to numeric constants (e. g., after labelling symbolic variables they contain). Therefore, an arithmetic expression that

contains only `int` (logic) variables and `int` constants can be used anywhere where an `int` expression is expected.

The behaviour described so far is deterministic. However, as soon as a symbolic expression is used as part of a condition that leads to branching (e. g., in an `if` statement), it is possible that the execution environment cannot decide on a unique outcome, e. g. whether a condition evaluates to `true` or `false`. When there is more than one choice, non-determinism is introduced, so that execution may continue with any of the possible branches [DK18b]. The execution environment makes a choice by selecting a branch, thus asserting a particular outcome (e. g., the condition shall be `false`). That assertion is maintained by imposing a corresponding constraint on the constraint store. After executing that branch, the execution environment backtracks state (constraint store, operand and frame stacks, program counter, and heap values) to the point where a choice was made, and then proceeds with the next choice. In Muli, this behaviour is referred to as search.

In order to limit the effects of non-deterministic execution, non-deterministic branching has to be encapsulated in the program. To that end, Muli offers encapsulation methods such as `getAllSolutions()` or `getOneSolution()` that take a lambda expression or a method reference as a parameter which is then executed non-deterministically. The result of non-deterministic branching is a symbolic execution tree [Kin76]. Solutions to a constraint-logic problem correspond to the leaves of that tree, i. e. where execution ends, such as by throwing an exception or returning a value or expression. The encapsulation method collects the required solutions and returns them to the calling, deterministic program.

14.3 Reference Type Logic Variables (or Free Objects)

As Muli is based on Java, Muli distinguishes the same four kinds of reference types as Java [Gos+15, § 4.3]: class types, interface types, array types, and type variables. Type variables are fundamentally different from the other kinds, as they are substituted by a reference type. For example, `ArrayList<E>` contains the type variable `E` that is substituted by a reference type, e. g., `Object` or `String`. In contrast, the other kinds of reference types imply that they are instantiated at runtime with values that come from the heap, i. e. they point to data structures such as objects or arrays. Since type variables are that different, they are excluded from further considerations in this work, resulting in a definition of

reference types that is congruent to that of C# [Mic15].[30] Class and interface types exhibit an identical structure [Gos+15], whereas array types are interpreted differently. Even though array types are interesting as well, this work focuses on class and interface types for now. In the following, they are subsumed as *reference types* for improved legibility.

Due to the nature of Java (and, therefore, Muli), the reference types that this work focuses on are not limited to data encapsulation. They also encapsulate behaviour (via methods) that may change along the implementation hierarchy as a consequence of overriding. Therefore, when a variable that is declared by `Object o` is of type `Object`, `o` may hold an instance of `Object` or of its subclasses. This affects the typecasts that can (validly) be performed on `o` at runtime, as well as the behaviour that is expected from invoking methods on the object. This implies that interactions with a reference type logic variable declared by `Object o free` need to consider that `o` may represent instances of subclasses of `Object` as well.

Consequently, we first need to define at which point exactly non-determinism may be introduced when interacting with reference type logic variables. Options are either during declaration/initialisation of a reference type logic variable (i. e. at `Object o free`), or when a feature of a variable that is not sufficiently specified is required later during runtime (e. g., on invocation of `o.toString()` or on access to a field such as `square.width`). If non-determinism were already introduced at declaration/initialisation time, this would introduce many branches that are potentially irrelevant, because the SJVM cannot determine how many choices will be required. Therefore, aiming to reduce the state space, Muli creates choices only if discriminating behaviour is expected, e. g., when control flow branches. For reference type logic variables, discriminating behaviour is not expected at the declaration of a logic variable (which can be done deterministically) but can be expected when one of its fields is accessed or its methods are invoked. Hence, we propose that non-determinism is incurred when a feature of a logic variable v is required, where v is not sufficiently specified to be handled deterministically. As a result, this allows search to focus on branches relevant to the respective access, thus effectively reducing the state space. Note that these considerations are similar to those regarding the *Label* reduction rule from [DK18b] that is used for substituting primitive type logic variables with their potential values. Similar to the present case, *Label* is suggested to be used only as a last resort if no other rule can be applied as its application results in

[30] Note that only the standalone use of type variables is disregarded here. Consequently, the reference types that we consider in the following may still make use of type variables as part of parameterised (generic) types.

```
1 class Demo {
2   public static void main(String[] args) {
3     A a = new A();
4     A b = new B(); } }
5 class A { public int i = 2; }
6 class B extends A { private int i = 1; }
```

Listing 14.3: Fields are only hidden, but not overridden.

one branch per potential value, which usually are a lot. If this is done too early during evaluation, this increases the state space unnecessarily [DK18b].

With this in mind, there are six different kinds of interactions between a program and a reference type logic variable that need to be examined in the following as they potentially result in non-determinism. First, accesses to fields of an object by a program, followed by invocations of methods. Moreover, the program can compare equality, which occurs in two forms in Java (and therefore in Muli), i. e. comparing reference equality or value equality, which are the third and fourth kind, respectively. Fifth, a program can perform operations on the type of a variable. Last but not least, as a novel kind of interaction, programmers may expect to be able to compare objects for structural equality, i. e. equality based on objects' field values instead of the entire object. This is similar to unification of constructor terms, which is common in logic programming but not in object-oriented programming languages.

14.3.1 Accessing a Field of a Free Object

In Muli and Java, fields are accessed using a dot notation, e. g., `square.width`. In contrast to methods, fields of a Java class cannot be overridden by subclasses. Although subclasses can declare fields with names identical to those in superclasses, this merely results in the original field being hidden from the overriding class, but not from the original one. Consider an artificial Java example in Listing 14.3. Accesses to `i` in both cases `a.i` and `b.i` result in the same value 2 because `a` and `b` are accessed via variables of type `A`. Of course, if `b` were stored in a variable of type `B`, that would not be the case. Muli shares this semantics with Java.

As a result, accesses to fields of free objects do not need to consider the class hierarchy of the object's type, but only the type of the reference type logic variable through which access takes place (here, `A`). Since a free object is uninitialised, in its initial state all its fields are to be treated as logic variables as well. Therefore, accessing a field of a free

object is a deterministic operation. Its result is the logic variable that is the field of the object. For instance, in the running example accessing `square.width` yields the logic variable of type **`int`** that is stored at that field.

14.3.2 Invoking a Method on a Free Object

For a variable `Shape s` **`free`**, consider the statement `s.getArea()` as seen in Listing 14.1. As `s` is declared free, this causes the execution to evaluate the method `getArea()`. `Shape` is merely an abstract supertype, so all the subtypes need to be taken into consideration, as they provide implementations for `getArea()`. Similarly, even in the deterministic nature of Java, the method that is actually invoked depends on the type of the referenced instance, not on that of the variable. Consequently, in order to determine which actual implementation is going to be invoked, the statement `s.getArea()` causes the SJVM to discover the set S of non-abstract subtypes that extend `Shape`.[31] If the supertype can be instantiated as well, the set of relevant types then is $S' = S \cup \{\texttt{X}\}$ for a non-abstract supertype `X`. Otherwise, the set of relevant types is just $S' = S$. For the running example, $S' = S = \{\texttt{Square}, \texttt{Rectangle}\}$, as the supertype is an interface type and is therefore abstract.

In general, the set of relevant subtypes can be restricted further, thus reducing the number of non-deterministic branches that the SJVM needs to evaluate. After all, we are only interested in those branches that potentially exhibit distinct behaviour. Therefore, the SJVM needs to discover $S'' \subseteq S'$, comprising only those classes that provide their own implementations of `getArea()`, thus omitting all types that merely inherit an implementation from their supertype. Afterwards, the SJVM only needs to evaluate one branch per element of S''. If S'' holds exactly one type, execution continues deterministically by invoking that type's implementation on *s*. Otherwise, evaluation creates a choice point in order to execute all `((t)s).getArea()`, where $\texttt{t} \in S''$. As a result, the number of choices that this choice point provides is equal to the cardinality of S''.

Looking at the running example from Listing 14.1, S' cannot be reduced as all subtypes provide their own implementations, i. e. $S'' = S' = \{\texttt{Square}, \texttt{Rectangle}\}$. For this reason the `System.out.println` statement is expected to be executed twice, as indicated in Section 14.1; once per type in S''. To discuss a different example with a more detailed implementation hierarchy, consider the classes depicted in Figure 14.2. For a logic variable `A a` **`free`**, invoking `a.m()` results in discovering the subtypes $S = \{\texttt{B}, \texttt{C}, \texttt{D}\}$ first. The

[31] In general, this includes parameterised (generic) types that remain in parameterised form (e. g., `ArrayList<E>`). Therefore, this set is finite.

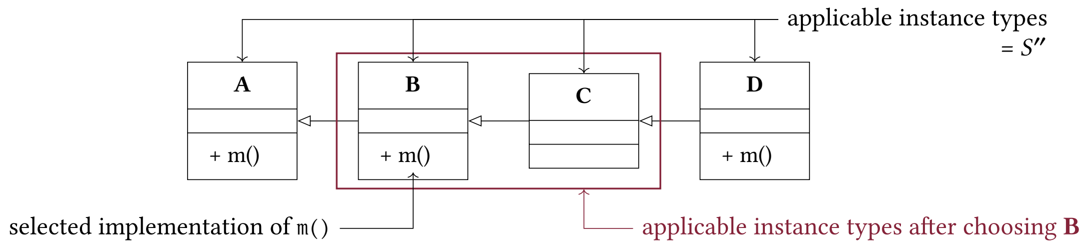


Figure 14.2: Applicable instance types for a given object `A a free` before and after choosing a particular subtype.

supertype `A` is non-abstract, therefore $S' = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$. However, since `C` does not provide its own implementation of `m()` and relies on that of `B` instead, the set is reduced further to $S'' = \{\texttt{A}, \texttt{B}, \texttt{D}\}$. The SJVM then continues the evaluation based on S''.

After making a choice for a type $t \in S''$ whose method implementation is used, the actual type of the instance that the method is invoked on can be an arbitrary one from a set of types. Specifically, either the determined type or any of its subtypes. However, the set of allowed types is restricted further, as it may not contain subtypes that provide their own implementation (as *their* implementation would need to be invoked otherwise). This is illustrated in Figure 14.2, where the set of types is constrained only to `B` and `C`. Even though `D` is a subtype, it provides an own implementation of `m()` and would therefore conflict with having chosen `B`'s implementation.

As a result of choosing an implementation, the SJVM needs to add a constraint to its constraint store that restricts the type of s according to the above description. This ensures that later interactions with that object do not make conflicting assumptions regarding the type of s, i. e. to avoid assuming s to be of a type that is not in the reduced set of applicable types. Similarly, a type t cannot be assumed for s if that would violate a previously imposed constraint, so the corresponding branch must not be evaluated. Consequently, the constraint that restricts an instance's type is a set-based constraint. This type of constraint is novel to Muli, as existing constraints are only of arithmetic nature.

14.3.3 Comparing Reference Equality of Reference Type Logic Variables

In Muli and Java, objects are typically compared by one of two means, either reference equality or value equality. First, let us focus on the former. Based on the program in

```
1  Object o free;
2  Object p = new Object();
3  Object q free;
```

Listing 14.4: Declaration of a set of reference type variables.

Listing 14.4, consider the conditional control flow statements `if (o == p)` and `if (o == q)` that compare references of reference type (logic) variables.

As `o` and `q` are declared free, it needs to be discussed whether the constraint created by evaluating reference equality should result in the SJVM unifying their references upon evaluation of the condition, i. e. result in `o` pointing to the instance referenced by `p` (for `o == p`), or to the same reference as the other logic variable `q` (for `o == q`). Arguably, this should not be the case. Listing 14.4 expressly declares the three variables to be three different instances, unlike an assignment, such as `Object w = p`, which would explicitly make `w` assume the same reference as `p`.

Therefore, the evaluation of a condition comparing reference equality is a deterministic operation even for reference type logic variables that yields `true` iff two variables reference the same free object, which is consistent with the Java semantics of comparing reference equality. No implicit unification is performed.

14.3.4 Comparing Value Equality of Reference Type Logic Variables

In addition to the means described in Subsection 14.3.3, Java (or Muli) code can also compare objects in terms of value equality, e. g., by `if (o.equals(p))` or `if (p.equals(o))` (after an initialisation as depicted in Listing 14.4). This presents another opportunity for unifying objects if free objects are involved.

As `equals()` is a method that every class can implement individually, the interpretations of these two examples are fundamentally different. In `p.equals(o)`, `p` is a concrete instance of `Object`, so `Object`'s default implementation is invoked deterministically, effectively checking for reference equality. Other implementations might compare instances by accessing fields of the free object `o`, thus resorting to the case described in Subsection 14.3.1. In contrast, `o.equals(p)` is an invocation of `equals()` on the logic variable `o`. As a result, this case reduces to the invocation of methods (cf. Subsection 14.3.2), resorting to specific implementations of `equals()`, e. g., of `Square` and `Rectangle`. Consequently, `equals()` is not commutative.

As a result, Muli does not need to handle value equality comparisons specifically, as they are implicitly covered by other considerations regarding reference type logic variables.

14.3.5 Performing Type Operations on a Free Object

The (super-) type of a logic variable is determined by its declaration, but initially the corresponding instance may be of that type or of its subtypes (cf. the definition of S in Subsection 14.3.2). This affects operations that operate on the type of a free object; namely **instanceof** and typecasts. For example, the set of allowed types for the instance is reduced by (successful or failed) typecasts. Considering Listing 14.4 again, a program might try to cast a reference type logic variable to a subtype, e. g., `(Square)o`. In that case, given that this is a valid cast, the actual type of `o` can be `Square` or any of its subtypes.

Typecasts can be either valid or invalid at runtime. Invalid typecasts are those that violate the class hierarchy, such as casting an object of type `Square` to `Rectangle`. This deterministically yields a `ClassCastException` and therefore does not result in a choice point. The result of evaluating **instanceof** statements in a similarly invalid contexts is deterministically **false**.

In contrast, performing a valid typecast results in two choices as to how execution can continue. Either the cast is successful (unless a contradictory constraint exists in the constraint store at runtime), so a new constraint can be imposed narrowing the logic variable's type; or the cast is not successful. In regular Java, the latter case is not caught by a compiler and results in a runtime exception (`ClassCastException`). Similarly, Muli can handle this case by imposing a corresponding constraint and throwing that exception. Therefore, a valid typecast of a reference type logic variable results in a choice point with at most two options, depending on existing constraints in the constraint store. Similarly, using **instanceof** in a valid context results in non-deterministic execution that imposes the same constraints as successful or unsuccessful typecasts.

To support non-deterministic branching, a constraint is needed that is imposed when a choice is made for a branch that corresponds to a type operation. This constraint reduces the set of possible instance types. The set-based constraint from Subsection 14.3.2 can be re-used, but the sets are computed differently. Given that S describes the set of applicable types prior to imposing a constraint and U describes the set of types comprising the cast target types and all of its subtypes, on a successful cast, the set of applicable types is narrowed to the intersection $V = S \cap U$, whereas for a failed typecast all remaining types are applicable, i. e. the type is constrained to the set difference $V' = S \setminus U$. The

resulting sets of types are used to impose the corresponding constraints, i. e. V for the constraint that is added to the constraint store when making the choice that the typecast is successful, and V' for the other choice.

14.3.6 Imposing a Constraint for Structural Equality between Two Objects

The cases discussed so far refer to the interpretation of object-oriented concepts against the background of a constraint-logic object-oriented language. In addition to that, Muli creates a novel opportunity regarding unification of objects that cannot exist in plain object-oriented languages without symbolic execution, namely comparing (free) objects for structural equality (in combination with constraints that enforce it).

Value equality relies on the `equals()` method that a class can implement individually (cf. Subsection 14.3.4), for example so that equality depends only on a specific field. In contrast, we use the term structural equality to refer to a situation in which all fields of two (free) objects of the same type either share identical values (for fields of primitive types) or are structurally equal again (for reference-type fields), i. e. the following recursive definition applies: $o_1 \odot o_2 \Leftrightarrow type(o_1) = type(o_2) \land ((o_1.x \text{ primitive} \land o_1.x = o_2.x) \lor (o_1.x \text{ not primitive} \land o_1.x \odot o_2.x))\ \forall x \in fields(o_1)$,[32] where $type(o)$ is the type of an object o and $fields(o)$ is the set of its fields. For example, given two free objects `Rectangle r1 free, r2 free`, imposing structural equality `r1 ⊙ r2` implies that `r1.width == r2.width` and `r1.height == r2.height` in addition to sharing their type. Similarly, if `r2` were an initialised object of type `Rectangle`, the values of `r1`'s fields are unified with those of the corresponding fields in `r2`. As a result, `r1 ⊙ r2` ⇔ `r2 ⊙ r1`, i. e. structural equality is commutative.

A new operator is needed to denote the structural equality constraint $\odot$ in source code. For that purpose, I introduce the symbol `#=` to be used as a boolean, binary operator in conditions in order to add this constraint to the constraint store at runtime. It evaluates to **`true`** if fields of two objects are unifiable as described above, and to **`false`** if they are not. In both cases, a corresponding constraint is added to the constraint store that maintains this equality.

[32] Note that here $fields(o_1) = fields(o_2)$ since $type(o_1) = type(o_2)$, so $fields(o_2)$ could be used just as well.

14.4 Implementation

The considerations in Section 14.3 require modifications to the Muli SJVM in terms of additional constraints and choice point types. This results in changes that need to be made to the SJVM's solver component and its choice point generator (cf. [DK18a]).

The *applicable type constraint* is a set-based constraint that restricts possible types for a free object. It maintains a reference to the free object that it affects, and a set of fully qualified names of types that the object may assume. This set is defined prior to instantiation of that constraint. In the solver manager, a constraint is imposed in conjunction with all other constraints in the constraint store. Therefore, the solver manager can verify consistency of a constraint store by collecting all imposed applicable type constraints involving a free object and checking that the intersection of the sets of types is non-empty for each object, i. e. there is at least one type that any object can assume; in addition to verifying consistency of the remaining constraints.

Additionally, the *structural equality constraint* translates into a conjunction of arithmetic equality and type equality constraints as specified in Subsection 14.3.6, hence it does not need to be represented on its own. The *type equality constraint* references two involved objects that need to be of the same type. A constraint store comprising a type equality constraint is consistent if both objects are trivially of the same type (such as for regular objects) or if there is a type that is among the applicable types of both objects.

At runtime, evaluations of bytecode instructions that incur non-determinism result in the creation of choice points. These are responsible for controlling search and, hence, for imposing constraints and removing them afterwards [DK18a]. Therefore, the support for type operations on logic reference type variables requires a corresponding choice point. It offers choices according to the description in Subsection 14.3.5 and imposes an appropriate instance of the applicable type constraint for each choice. Similarly, a choice point for invoking a method according to Subsection 14.3.2 is required. Both choice point implementations require the implementation of new helper methods that discover sets of available types. The method `Type[] getSubtypes(Type)` discovers, for a given type, all of its subtypes from the loaded classpath. A further method `Type[] getImplementations (Type[], Method)` is required that filters a list of types such that it returns only those types that can be instantiated and that provide an own implementation of a particular method, thus supporting the case from Subsection 14.3.2.

Last but not least, another choice point is generated if free objects are compared for structural equality as specified in Subsection 14.3.6. It comprises two choices. One choice

represents that equality is maintained, resulting in the corresponding constraint being imposed. The other one corresponds to imposing the negation of that constraint.

14.5 Related Work

Several approaches intend to integrate elements from object-oriented programming into declarative languages, mostly based on Prolog. For example, tuProlog provides a Prolog engine implemented in Java, offering access to Java features from Prolog [DOR05]. However, referring to Java types is done rather artificially by means of string literals which cannot be checked by a compiler, and free objects and accessing their fields are not considered. As a non-Prolog-based example, Oz is a constraint language that offers OO features, but does not seem to support constraints involving logic objects [Van+03]. Despite their integration, the mentioned programming languages follow a declarative style, which might not be as accessible for developers who are used to imperative languages.

CAPJa intends to seamlessly integrate Prolog search into Java programs, e. g. by providing a Java-based abstraction layer from Prolog [Ost15]. The integration supports a mapping of data structures from Java to Prolog and vice-versa, but focuses on logic objects used for encapsulating data. It does not consider free (unbound) objects in terms of method invocations and field accesses, which become relevant if we consider that objects also encapsulate behaviour, which is expected in object-oriented programming. As another example, the library `heya-unify` facilitates unification of data structures in JavaScript [Laz14], particularly in order to compare object contents or to perform pattern matching on them. However, it does not support defining entire objects as logic variables and is limited to comparing structural equality on weakly typed objects and arrays.

The type unification algorithm presented by [Plü09] can be used for Java type inference. Although their work emanates from a different standpoint, the type unification could be re-used for formulating the subtype relations for the constraints in this work.

Other work demonstrates that the use of languages integrating multiple paradigms is beneficial, most notably the Java Stream API [UFM14] and Scala [Ode+17], which integrate object-oriented programming with functional programming on the JVM. LINQ offers a similar integration, but for languages on the .NET CLR [MBB06]. A very relevant integration of logic and functional programming is Curry [Han+95], which incorporates logic programming into a language with Haskell syntax. Muli lends and adapts some ideas from Curry, such as encapsulated search and constraint definition via boolean equalities

[AH15]. However, the adaptation of these concepts to constraint-logic *object-oriented* programming results in fundamentally different considerations and implementations.

14.6 Concluding Remarks

This work contributes a concept for reference type logic variables in constraint-logic object-oriented languages. It details interactions of programs with reference type logic variables and discusses approaches for handling such interactions, on the basis of the programming language Muli. As a result, there now is a concept for invocations on free objects and accesses to their fields, comparisons of different kinds of equality, and type operations in constraint-logic object-oriented programming.

The discussed approaches efficiently introduce non-determinism where it is specifically required and take class hierarchies into account. This requires a novel constraint that restricts types of free objects to support these approaches. Since the constraints previously supported by Muli were of a purely arithmetic nature, this work also contributes a set-based constraint to restrict the possible types of free objects.

The contribution is helpful not just for Muli but for constraint-logic object-oriented programming in general, because it allows non-deterministic search to extend beyond logic variables of primitive types. For example, a constraint-logic object-oriented language based on C# could also make use of these approaches. Furthermore, it facilitates the usage of object-oriented features in combination with free objects.

Subsequently, the implementation of this approach in the Muli SJVM will be completed in order to evaluate its benefits. The resulting virtual machine implementation will be part of the open source distribution of Muli provided via GitHub.[33] It is also planned to provide an augmented formal semantics, incorporating the aspects discussed in this paper, thus yielding an integrated semantics for a constraint-logic *object-oriented* language. Future work will tackle the extension of these considerations towards further reference types, particularly array types.

References

[AH15] S Antoy and M Hanus. 'From Boolean Equalities to Constraints'. In: *Logic-Based Program Synthesis and Transformation.* 2015, pp. 73–90.

[33] `https://github.com/wwu-pi/muli`.

[DK18a] Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *SAC 2018*. ACM, 2018, pp. 1185–1194. ISBN: 978-1-4503-5191-1/18/04. DOI: 10.1145/3167132.3167260.

[DK18b] Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DOR05] Enrico Denti, Andrea Omicini and Alessandro Ricci. 'Multi-paradigm Java-Prolog integration in tuProlog'. In: *Science of Computer Programming* 57.2 (2005), pp. 217–250. ISSN: 01676423. DOI: 10.1016/j.scico.2005.02.001.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 09/06/2017).

[Han+95] M Hanus, H Kuchen, J J Moreno-Navarro, JR Votano, M Parham and LH Hall. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Kin76] James C. King. 'Symbolic execution and program testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[Laz14] Eugene Lazutkin. *Unification for JS*. 2014. URL: http://www.lazutkin.com/blog/2014/05/18/unification-for-js/ (visited on 29/06/2018).

[MBB06] Erik Meijer, Brian Beckman and Gavin Bierman. 'LINQ: Reconciling Objects, Relations and XML in the .NET Framework'. In: *ACM SIGMOD International Conference on Management of data*. 2006, p. 706. ISBN: 1595932569. DOI: 10.1145/1142473.1142552.

[Mic15] Microsoft. *Reference Types (C# Reference)*. 2015. URL: https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/reference-types (visited on 27/06/2018).

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/ (visited on 13/06/2017).

[Ost15] Ludwig Ostermayer. 'Seamless Cooperation of Java and Prolog for Rule-Based Software Development'. In: *Proceedings of RuleML 2015*. 2015. URL: http://ceur-ws.org/Vol-1417/paper2.pdf.

[Plü09] Martin Plümicke. 'Java type unification with wildcards'. In: *Applications of Declarative Programming and Knowledge Management. INAP 2007, WLP 2007.* Springer, 2009. DOI: 10.1007/978-3-642-00675-3_15.

[UFM14] Raoul-Gabriel Urma, Mario Fusco and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT: Manning Publications Co., 2014. ISBN: 9781617291999.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. DOI: 10.1017/S1471068403001741.

15

A Constraint-Logic Object-Oriented Language

Jan C. Dageförde[*] · Herbert Kuchen[*]

Citation Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*. ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260.

Copyright notice Republished with permission of ACM (Association for Computing Machinery), from Proceedings of the 33rd Annual ACM Symposium on Applied Computing (2018); permission conveyed through Copyright Clearance Center, Inc.

Abstract Object-oriented (OO) programming languages prevail in the development of enterprise software, but they do not particularly support the implementation of software which includes solving complicated search problems with dynamically appearing constraints, e. g. as found in logistics. Such problems could be tackled by implementing the main business logic in e. g. Java and the search in a constraint-logic language. However, integrating both aspects is clumsy.

Thus, we propose the constraint-logic OO language Muli. It facilitates an integrated implementation of applications that use both aspects. Muli extends Java by logic variables and encapsulated search. Its implementation is based on a symbolic Java virtual machine and constraint solvers. Outside of search regions, Muli behaves just like Java.

We motivate the benefits of integrating object-oriented programming and constraint-logic programming and introduce concepts that are required to achieve a seamless integration. We also describe our implementation of these concepts and discuss our approach.

Keywords Programming paradigm integration · Java · symbolic execution · constraint-logic programming · virtual machine implementation.

[*]University of Münster, Germany

15.1 Motivation

In contemporary software development, object-oriented (OO) programming is prevalent, as languages such as Java and C# continue to dominate usage rankings [Sta17; TIO17]. Inheritance and encapsulation of behaviour and structure are examples of features that make them useful for most industry applications and provide reusability as well as maintainability [Lou93]. However, there are scenarios in which languages from other paradigms are more suitable.

Consider search problems: Constraint-logic programming languages, such as Prolog with the CLP(FD) package, allow for declarative specifications of the search space by means of variables and their constraints [Tri12]. As a result, finding solutions within the search space is performed implicitly by the runtime environment and the included constraint solver. In contrast, solving search problems in Java requires either manually implementing an imperative solver or importing non-standardised constraint-solver libraries. Self-made implementations of solvers are often highly specialised towards a given problem, which might be beneficial for performance but harms generalisability. Another option, using (e. g.) Prolog via the Java Native Interface (JNI), is tedious and error-prone due to the nature of the JNI [KO08].

In an effort to remedy this situation, we propose a novel approach to integrating constraint-logic and OO paradigms based on Java: the constraint-logic object-oriented programming language *Muli*. Instead of developing yet another constraint solver library, of which there are many (cf. e. g. [PFL16; Opt17; Kuc03]), our solution provides means for constraint-logic programming within Java programs by adding the concept of free variables directly to the language, i. e. variables that are not initialised to a particular value but to a symbolic value of a certain Java type. This is combined with symbolic execution by a specialised Java virtual machine (JVM) that adapts concepts from the Warren Abstract Machine [War83]. Within code parts that we refer to as *search regions*, execution becomes non-deterministic whenever branching conditions involve one or more free variables whose domains are insufficiently constrained. By backtracking, the JVM ensures that all applicable branches are executed.

Muli is particularly suited for enterprise applications where most of the business logic can be expressed adequately in Java, but which occasionally require the solution of search problems whose details have been assembled in previous inputs or calculations. An example of such an application is a truck scheduling system which dynamically adapts its schedule depending on traffic information and newly arriving orders, thus incrementally adding constraints.

This paper introduces Muli as a constraint-logic OO language and motivates concepts for language and runtime that are required to achieve this integrated paradigm. Furthermore, it explains our implementation of these concepts. To these ends, our paper is structured as follows. We start off by describing novel language elements and a corresponding compiler in Section 15.2. Section 15.3 presents a custom implementation of a symbolic JVM (SJVM) that supports search and backtracking, detailing structures and runtime concepts required for the execution of Muli applications. Using sample applications, we discuss our approach in Section 15.4, outlining its advantages but also its current weaknesses. We then summarize related work in Section 15.5. In Section 15.6, we conclude and point out future work.

15.2 Muli Language

Muli is a language extension to Java, with Java 8 as the reference language. Additions to the language are kept to a minimum and we entirely refrain from making modifications to existing Java concepts and features in order to minimise the burden on Java developers to understand Muli programs. As a result, Muli is a superset of Java, so that every Java program is also a Muli program that can be compiled and executed by Muli.

Our approach follows some design principles that we deem useful. First, we want to solve search problems supported by a custom-tailored SJVM. Second, we want search to be encapsulated. As a result, non-deterministic execution is only performed if explicitly required, whereas other parts of the program remain deterministic and cause no overhead w. r. t. Java. Third, we refrain from adding more special syntax than absolutely necessary, especially for defining constraints. For example, we do not want to add operators for constraints that can be expressed using relational Java operators. Fourth, since Java programs are not executed lazily, Muli should not be evaluated lazily either, in contrast to integrations of logic programming with other paradigms (cf. e. g. Curry [Han97]). Last but not least, Muli should be considered an extension of Java, as opposed to an entirely new language. This implies that functionality (and therefore understanding!) of Java constructs remains unchanged and performance of deterministic program parts should not be adversely affected.

We decided to use Java as the reference language as it is a ubiquitous programming language which is well-known and well-understood among most developers. Moreover, it comprises advantageous features of OO imperative languages, such as platform independence, inheritance, and encapsulation of data and operations [Lou93]. Furthermore,

although no official formal operational semantics exists, Java and its corresponding runtime are documented extensively in natural language [Gos+15] and [Lin+15], respectively), which facilitates both conceiving an extension and deriving implementations.

15.2.1 Language Concepts

Extending Java into a constraint-logic OO language requires a few concepts that are novel to Java. First, we need to add the concept of *logic variables.* Actually, given the SJVM, any Java variable can be considered a logic variable. However, regular Java enforces that every variable must be initialised to a particular value before it is used. In contrast to that, Muli introduces *free variables* using the **`free`** keyword, indicating that they are initialised, although not to a particular value.

Second, we add *encapsulated search*, adapting the identically named concept from the functional constraint-logic language Curry [Han+95], to provide an abstraction from non-deterministic execution. Within encapsulated search, non-deterministic execution can happen, whereas any program part outside encapsulation is deterministic, analogous to Java. We refer to a program part inside encapsulation as *search region.* An encapsulated search region is executed symbolically. *Constraints* are incrementally imposed whenever branching occurs that involves insufficiently constrained (logic) variables, thus introducing non-determinism. Once a valid branch is chosen, its branching condition is imposed as an additional constraint, and symbolic execution continues. Later, execution of the branch is backtracked and the next branch is chosen analogously.

The search region's return values, which can be multiple due to non-determinism, are considered *solutions* that the encapsulation collects and returns to its caller. Furthermore, we enable developers to cut execution branches, resulting in immediate backtracking without adding a solution.

Generally, we consider runtime exceptions that occur during execution of a search region as a kind of solution, as they are just another result of the execution. Although they do not represent a particular value, they may be of interest to the surrounding application. To facilitate control over this behaviour, we propose operators that configure encapsulated search and its return value. The most general case is that all solutions of a search region, including exceptions, are to be returned (`getAllSolutionsEx`). This general case can be modified to return the first solution (`getOneSolutionEx`), to discard exceptions (`getAllSolutions`), or in combination to return the first non-exception solution (`getOneSolution`).

```
1 public static void main(String[] args) {
2   int i = Muli.getOneSolution(() -> sqrt(5));
3   System.out.println(i); }
4 public static int sqrt(int y) {
5   int x free;
6   if (x == y/x) return x;
7   else throw Muli.fail(); } // not defined
```

Listing 15.1: Muli program that searches the (integer) square root of 5 and prints the result 2 (class header omitted).

As an introductory example, Listing 15.1 presents a simple Muli application that makes use of the constraint-logic OO programming style. The example application searches and prints the square root of a fixed number. We express this by the constraint $x == y/x$ (i. e. $x == \lfloor\sqrt{y}\rfloor$) over integer variables x and y, resorting to a constraint solver for finding x. As in Java, a method with the signature `public static void main(String[])` is used by the SJVM as the entry point. Computation remains deterministic, i. e. non-searching, until encapsulation begins. Assuming that we are interested in only one solution that should not be an exception, we use the `getOneSolution` operation that returns a single non-exception value. We use this operation in `main()` in order to create an encapsulated search region that calls the method `sqrt()`. For elegance, the search region is expressed as a lambda abstraction, but a method reference could be used instead, facilitating reuse of search regions across an application. By using a lambda abstraction rather than an expression as argument of `getOneSolution` (or other operators), we make sure that the search region is not immediately evaluated, but only under control of the encapsulated search mechanism.

In `sqrt()`, `x` is declared as a free variable which might later be bound to a value. The branching condition of the `if` statement cannot be evaluated to a single boolean value, as `x` is unconstrained. Therefore, the constraint `x == x/y` is added to the constraint store and the computation continues with the first branch of the `if` statement, namely `return x`. Since the `return` statement finishes the considered execution branch, the constraint solver searches and finds a solution satisfying the accumulated constraints (here consisting of a single constraint) and returns the obtained value for `x`, namely 2, as a result. Solutions not fulfilling the above constraint are cut off by the `fail()` operator.

An example to finding multiple solutions is printed in Listing 15.2. In this case, `getAllSolutionsEx()` is used to start encapsulated search. This operator returns multiple

```
1  public static void main(String[] args) {
2    Stream<Solution<Integer>> factorials =
3        Muli.getAllSolutionsEx(() -> {
4      int n free; return fact(n); } );
5    factorials.limit(100).forEach(i ->
6      System.out.println(i)); }
7  private static int fact(int n) {
8    if (n == 0) return 1;
9    else if (n >= 1) return n * fact(n - 1);
10   else throw Muli.fail(); } // not defined
```

Listing 15.2: Muli program that searches factorials non-deterministically and prints the first 100 of them (class header omitted).

solutions that may also include thrown exceptions, using a stream of `Solution` objects that each encode one solution.

In this example, the search region declares a free variable **int** n that is passed as an argument to a method `fact()` which is supposed to compute *n*!. Since **int** n is still unconstrained, all three execution branches remain possible and will therefore be executed non-deterministically. In our implementation, this means that all possible branches will be tried one after another by backtracking and that all found solutions will be delivered to the resulting stream. One branch is exempted from the overall solution using the **fail**() operator, whereas any other exception would be considered a solution. The first branch imposes the constraint `n==0` and returns the constant 1. The second branch imposes the constraint `n >= 1` and returns an arithmetic expressions over n and recursion. Since n is still not bound to a fixed integer value, the expression cannot be evaluated yet and will be represented symbolically. Note that, by recursion, further branching is introduced. These branches will bind n to 1, 2, 3, ..., respectively, such that the expression can then be evaluated to an integer and returned as a result. Note also that the finally resulting stream of factorials is infinite. This is no problem, as long as only a finite part of it is actually needed and computed, as in our example, where only the first 100 factorials are printed.

Muli realises additional potential in search problems that are not fully defined at once, but that add new, incremental constraints during execution. As an example, Listing 15.3 exhibits a program that iteratively adds constraints over a free variable x from user input and finds a solution for x. Consider also a variation that derives additional constraints

```
1  import de.wwu.muli.Muli;
2  class MuliIncrementalArgs {
3    public static void main(String[] args) {
4      int x = Muli.getOneSolution(() -> {
5        int x free; int i = 0;
6        while (i < args.length) {
7          if (!(x<Integer.parseInt(args[i++])
8             && x>Integer.parseInt(args[i++])))
9            break; }
10     });
11     System.out.println(x); } }
```

Listing 15.3: Iterative addition of constraints from user input in Muli.

from user input at runtime, e. g. via `BufferedReader.readLine()`, where the full constraint system cannot be known before the application starts.

Note that a solution can be a data structure containing the values of possibly several free variables, in case that these values are required after leaving encapsulation (cf. the `Assignment` structure in Listing 15.4).

In general, each solution can be a unique value if constraints over the involved variables are sufficiently restrictive. In other cases, the solution describes a search space, i. e. an expression accompanied by its relevant constraints. If a particular value is required from that search space, the developer needs to explicitly use `solve()` to *label* the variables, i. e. successively try specific values for them, as it is usual in constraint solving. We decided not to do this implicitly, as developers might want to refine search spaces by constraints in later search regions, which they would be unable to do if labelling occurred in the meantime.

15.2.2 Syntactic Extension of Java

Our examples suggest that only minimal language extensions are necessary in order to implement these concepts. Syntactically, they are limited to adding the **free** keyword. Given Java's EBNF rules for declaring a field (adapted from [Gos+15]):

FieldDeclaration ::= FieldModifier Type VariableDeclarator (, VariableDeclarator)*;*

VariableDeclarator ::= VariableDeclaratorId (= VariableInitializer)?;

we can add **free** as an alternative to initialisation by changing the *VariableDeclarator* rule to:

*VariableDeclarator ::= VariableDeclaratorId (***free** | *(= VariableInitializer))*;*

Free local variables are enabled by an analogous extension. Method parameters cannot be declared free as parameters are supplied by the caller. Free variables may be passed as a parameter, but that does not require special syntax.

Note that the keyword could even be avoided completely. In pure Java, using a `FreeVariable<T>` class with a generic type `T` or an `@FreeVariable` annotation come to mind as alternatives. However, using a class with generic type introduces much overhead, especially when considering boxing and unboxing for free variables of primitive types. Annotations also fall short, since annotations of local variables are not preserved until runtime, whereas only those of class fields remain. Using declarations without initialisation is also insufficient as this results in uninitialised local variables or, in case of fields, in implicitly initialised instance variables. Instead, a compiler can parse the **`free`** keyword and transform it into bytecode, which is then interpreted by a specialised runtime.

15.2.3 Muli Classpath

The remaining concepts do not require syntactic changes to the language. Therefore, no further additions to the compiler are necessary. Instead, we propose a small library that will be on the classpath during compilation and execution.

A class `Muli` implements encapsulated search operators and **`fail`**`()` as static methods changing the SJVM's runtime behaviour. `getAllSolutionsEx()` implements the most general encapsulated search operator, from which further operators are derived.

For reasons of readability we implemented the classpath library directly using Java as far as possible. However, the library has to be able to change the state of the VM to switch the execution mode around encapsulation, to record solutions, and to enforce backtracking. Methods with that purpose are declared **`private static native`** and therefore do not provide a direct Java implementation. Instead, the Muli runtime engine provides their actual implementation that performs state changes.

15.2.4 Implementing a Compiler for Muli

Variables and class fields that are declared free need to be represented in bytecode accordingly. The JVM specification provides a set of attribute structures that can be extended arbitrarily without breaking bytecode compatibility [Lin+15]: In bytecode, each attribute names its type. Every JVM implementation is required to read all attributes and silently skip attributes whose names it does not recognise. We leverage this by specifying custom attribute types that are ignored by regular JVM implementations but that will be

picked up by our own one. The benefit of staying bytecode compatible with the JVM is that existing parsers and metaprogramming tools are also usable for Muli.

The bytecode provides four attribute structures that allow custom attribute types [Lin+15]. For free fields, we add a `FreeField` attribute to the `field_info` structure. Since every field maintains its own structure, the existence of a `FreeField` attribute is sufficient to indicate that that field is free, whereas its absence implies a regular variable.

Java maintains a table of local variables in bytecode as part of the `method_info` structure. Extending the tabular attribute `LocalVariableTable` by a boolean flag (i. e. free/regular) is a feasible alternative but that would breaks bytecode compatibility. Therefore, we add a `FreeVariablesTable` next to the `LocalVariableTable` with one entry per free variable that references entries from the `LocalVariableTable` by ID.

We have constructed a compiler based on the extensible compiler framework ExtendJ (formerly JastAddJ) which facilitates to easily extend an existing language in an aspect-oriented way [EH07]. The Muli compiler imports abstract syntax tree (AST), parser, and bytecode generator from ExtendJ's Java 8 modules. In the frontend, the AST is extended by two declarator subtypes for free variables, that are instantiated by the parser when it encounters the **`free`** keyword. The parser is modified according to the EBNF specification above. The backend picks up instances of the new AST types and generates the aforementioned attributes into the classes' bytecode in a JVM-compatible way.

15.3 A Backtracking, Symbolic VM

In addition to using Java as the reference language for extension, we also chose Java as an implementation platform for the virtual machine runtime. Incidentally, this makes the resulting constraint-logic OO language just as platform-independent as Java. As a positive side effect, we are able to leverage the multitude of third-party libraries written for the JVM that are useful for our endeavour, constraint solvers in particular.

The heart of the Muli runtime environment is the SJVM, i. e. a custom virtual machine that symbolically executes Java programs. Figure 15.1 depicts relevant components of the runtime environment and their relations, including subcomponents of the SJVM.

Since Muli programs are compiled to JVM-compatible bytecode, a specialised bytecode parser is not required. Therefore, the runtime environment executes all applications, regardless of whether they were implemented in Java or Muli. Nevertheless, only Muli programs can specify logic variables that are picked up by the runtime. Java programs by themselves will only be executed deterministically. However, Muli programs are fully

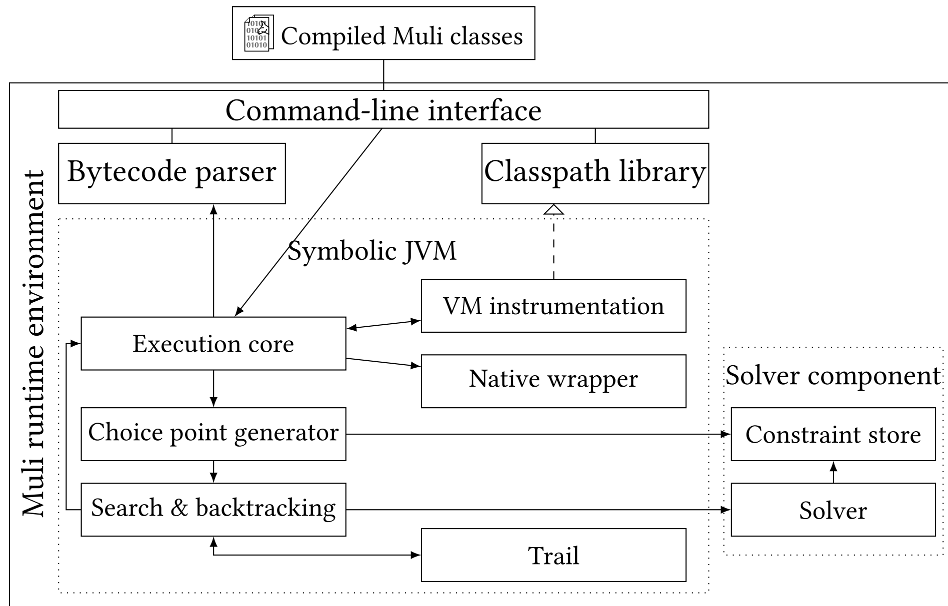


Figure 15.1: Components structure of the Muli runtime environment.

able to reference Java classes and interact with them non-deterministically. Consequently, it is possible that Java methods are executed symbolically, as long as they are invoked from a Muli program.

We explain the components and the concepts that influenced their implementation in the sequel. As Muli's SJVM is implemented in Java, it also runs on a JVM which will be referred to as the *enclosing JVM* for distinction.

Data structures

The Muli runtime environment implements all data structures that are required to execute Java programs, according to the JVM specification [Lin+15]. The most important one, the *Frame stack*, consists of frames, each representing an executed method call. The top of the stack corresponds to the method that is currently being executed. On return, the topmost element of the frame stack is popped and the SJVM continues execution of the new top. In a frame, the represented execution state comprises the program counter, values of local variables, and an *operand stack*. As in a regular JVM, most bytecode instructions operate on the operand stack, taking their input(s) from the top of the stack and pushing computation results. In contrast to a regular JVM, the operand stack does not only contain constant primitive values or addresses to objects. Instead, elements can also be symbolic representations of free variables or expressions. For simplicity, the

SJVM does not implement its own heap. Instead, it shares the heap with the enclosing JVM.

The aforementioned structures are required for a bytecode-compatible runtime that are modified to support symbolic execution and search. More importantly, the SJVM implements data structures derived from the Warren Abstract Machine (WAM) that is specifically designed to execute Prolog programs [War83].

In the WAM, the local stack comprises *environments*, which can be roughly compared to Muli's and Java's frame instances, and *choice points*, for which there is no corresponding structure on a regular JVM. Thus, the SJVM stores choice points in an additional *choice-point stack*. Each choice point maintains a *trail*, which is a concept also borrowed from the WAM [War83]. The trail is implemented as a stack as well. Each element represents an operation that must be performed on some component of the SJVM in order to undo a state change.

Additionally, the solver component of Muli maintains the *constraint store*. During execution of a program, constraints may be incrementally added to the store. Typically, the constraint store is described by a conjunction of atomic boolean expressions that correspond to branching criteria. Every choice point also maintains a references to constraints that where added by it, in order to remove them from the store on backtracking.

Last but not least, the SJVM has status flags that control its execution. Execution can either be *deterministic*, i. e. non-searching, or *non-deterministic*, i. e. searching. The latter state is only assumed during encapsulated search, ensuring deterministic execution outside the encapsulation. Further flags include e. g. the searching mode (currently only iterative deepening depth-first search is supported) and the requested logging level, which can provide helpful output during our development of the VM.

Symbolic types

Muli supports all types known from regular Java, including reference types and arrays. However, in order to accommodate for logic variables, additional types are introduced. According to the JVM specification, two basic kinds of types need to be distinguished: Primitive types and reference types [Lin+15]. However, we further split considerations of reference types into array reference types and object reference types due to their distinct structure.

Logic arrays are represented by instances of an `Arrayref` class, maintaining a logic array's element type (e. g. `int`), its dimensions, and its element values, which can be either regular values or logic variables. Similarly, logic objects are represented using `Objectref`

instances, each containing its class type and a map of its fields to their respective values, which can also be logic variables. A logic variable of numeric primitive type is an instance of class `NumericVariable`. It contains a flag indicating the particular primitive type. It does not have a value, but its domain is restricted by constraints in the constraint store. Analogously, logic boolean variables are described by `BooleanVariable` instances.

When encountering an instruction that performs operations on variables, the semantics of the SJVM does not distinguish between logic variable types and regular variable types. For example, primitive **`int`** variables are type-compatible to `NumericVariable` representations with integer type. Performing an `iadd` operation on an **`int`** variable and an `NumericVariable` will result in a symbolically expression representing the addition. The result is then pushed to the operand stack.

Solver component

The logic variable types are part of the *Solver component*, which is the runtime's abstraction layer from constraint solvers. It specifies types that are used to generate variables, expressions, and constraints during execution without having to consider particularities of constraint solvers. Before they are part of a constraint, expressions collected during symbolic execution are not mapped into a particular solver's object representation.

Currently, Muli integrates two constraint solvers from which a user can choose. Muconst is an SMT solver that was originally developed with automated glass-box test case generation in mind [Lem+04]. It supports linear and non-linear arithmetic theories as well as SAT solving. Its distinguishing advantage over other solvers is its handling of rounding errors in floating point solutions, ensuring that solutions still satisfy all constraints after rounding [EMK12]. Alternatively, the finite domain solver JaCoP is integrated, which is a free software constraint solver library for Java [Kuc03]. JaCoP's constraint propagation achieves early, computationally inexpensive detection of infeasible branches for applications that mostly involve finite domain numeric variables.

As the SJVM is able to work with multiple constraint solvers and abstractly defines constraints, we will treat the constraint solver as a black box in the remainder of this paper. Nevertheless, we formulate the following requirements for a constraint solver to be applicable: it needs to be able to label variables in order to find solutions where constraints are not sufficiently restrictive for finding solutions. Moreover, variables and labeling strategies for both finite domain problems and floating point problems need to be present and the constraint solver must be able to handle combinations of these problems.

Triggering bytecode instruction	Choice point type	Possible constraints
FCmpg, FCmpl, DCmpg, DCmpl	floating point comparison	$=, <, >$
LCmp	long comparison	$=, <, >$
If<cond>, If_icmp<cond>	**if** instruction, integer comp.	$=, \neq, <, \leq, >, \geq$
Lookupswitch, Tableswitch	**switch** instruction	$=, \notin$

Table 15.1: Bytecode instructions, resulting choice points, and applicable constraint types. <cond> is one of eq, ne, lt, le, gt, or ge.

Symbolic execution, encapsulated search, and choice points

Symbolic execution affects the interpretation of many Java bytecode instructions. For example, when a free variable is loaded by an instruction, a corresponding symbolic representation is pushed to the operand stack. Arithmetic operations that manipulate the operand stack involving symbolic representations push a symbolic representation of the result. Analogously, relational operators result in symbolic relational expressions.

Non-determinism may be introduced, if a branching condition contains logic variables. On the bytecode level, this corresponds to e. g. a conditional jump instruction such as ifne with two possible outcomes. Also, bytecode instructions, which may cause an exception (e. g. getfield, invokeinterface, invokevirtual, and checkcast), can cause branching. Depending on the branch that is chosen first, the branching condition or its negation will be pushed to the constraint stack and the execution will be continued at the corresponding instruction. The other branch will be considered after backtracking.

A selection of typical bytecode instructions, which may cause branching, as well as their corresponding choice-point types and possibly generated constraints are shown in Table 15.1.

Each choice point maintains a trail, a stack that, for every executed instruction, records the operation that will be necessary to reverse the effects of that instruction. For example, executing an instruction that takes two elements from the operand stack and pushes one leads to in three trail elements: one that will take one element from the operand stack and two that will push the values of the previous elements to the stack.

At the end of a symbolic execution path (i. e. when encountering the final **return** or a non-caught exception e. g. caused by **throw**), the SJVM will backtrack to the latest choice point and undo the corresponding changes recorded on the trail by replaying its trail stack. Moreover, the constraints imposed since the choice point will be removed from the constraint store. Afterwards, the next choice is realised by imposing its constraint and execution continues. Figure 15.2 illustrates how backtracking rolls back the trail, removes

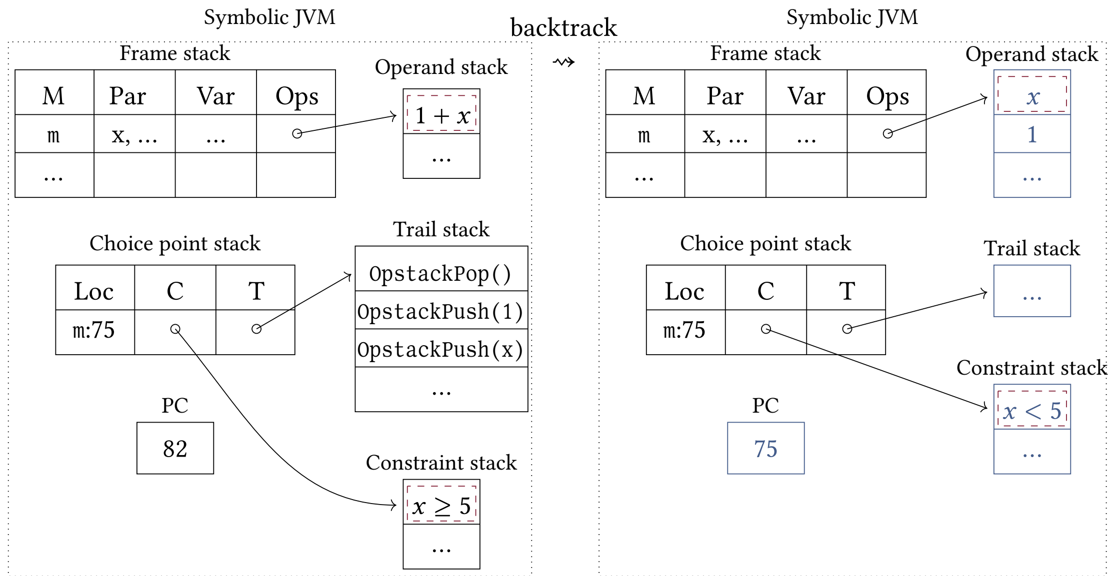


Figure 15.2: Effect of backtracking on execution state. Changes highlighted in blue. Operand stack elements with red dashed lines are on the heap, which we omitted for simplicity.

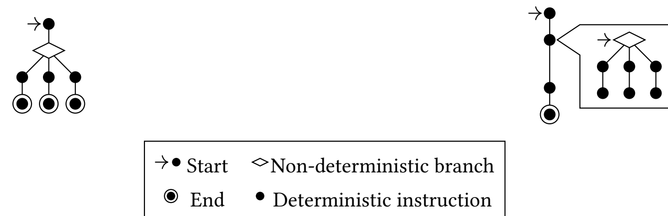


Figure 15.3: Unrestricted symbolic execution versus encapsulated symbolic execution.

a choice point's constraint, and imposes the next choice's constraint before continuing execution. When no further choice can be realised, the choice point is removed from the choice point stack and backtracking to the previous choice point occurs.

As we do not want to support non-determinism outside of encapsulated search regions, non-deterministic jumps are restricted to the searching mode of the SJVM. Therefore, encapsulation bounds the execution tree at the end of search regions, ensuring that effects of symbolic execution and backtracking remain local. Consequently, at the end of encapsulated search regions, the control is linearized again and the collected solutions (or solution spaces) are returned to the caller (see Figure 15.3). Encapsulated search can also be nested, thus achieving search hierarchies. If an unbound logic variable or a symbolically represented expression is accessed outside of a search region, an exception will be thrown.

It is debatable whether input/output should be disallowed in search regions, as this introduces side effects that cannot be backtracked by the SJVM. These issues are known from Prolog [Sco10]. On the other hand, such side effects may be wanted, as they facilitate printing (partial) solutions, logging, and asking end-users to supply additional data that may only be relevant in certain execution paths. Therefore, we decided not to forbid possible non-backtrackable side effects.

15.4 Discussion

Our approach towards an integration of constraint-logic and OO programming is useful for applications that are mainly programmed in Java while requiring a substantial amount of search. We expect it to be particularly suited for programs in which new constraints are discovered over time that are incrementally added to the existing set of constraints. The implementation of the constraint store facilitates such applications by reusing results from former searches when new constraints are added, thus preventing recomputation of partial solutions unless this is absolutely necessary. Further benefits are achieved by solvers with effective constraint propagation.

We demonstrate how Muli improves the programming style for search problems over pure Java using exemplary applications. Moreover, we quantify the performance of Muli programs in our runtime environment.

The example application in Listing 15.1 already demonstrates constraint solving, although with a problem that can be solved trivially in an imperative language as well. Less trivial is an application that solves the *Send More Money* Puzzle: Eight free integer variables s, e, n, d, m, o, r, and y need to be labelled with values from 0 to 9 such that every binding is different from the others, while fulfilling the constraint $1000s + 100e + 10n + d + 1000m + 100o + 10r + e = 10000m + 1000o + 100n + 10e + y$. For s and m, the value 0 is prohibited. This can be specified as a Muli search region as shown in Listing 15.4.

The helper method `diff` imposes the constraint that every variable be different from the others, while `domain` limits the variables' domains to $\{0, \dots, 9\}$. The `Assignment` class is a simple custom data structure comprising the eight variables, used to return all eight bindings.

We have compared the runtime of the application in Listing 15.4 with that of a corresponding pure Java application that attempts an imperative solution using eight **`for`** loops and backtracking. As an additional example application we implemented the Safe Lock

```
1  public static Assignment money() {
2    int s free, e free, n free, d free,
3        m free, o free, r free, y free;
4    if (domain(s,e,n,d,m,o,r,y) && s != 0 &&
5        m != 0 && diff(s,e,n,d,m,o,r,y)) {
6      if (        1000*s + 100*e + 10*n + d +
7                  1000*m + 100*o + 10*r + e ==
8        10000*m + 1000*o + 100*n + 10*e + y) {
9        Muli.solve(s,e,n,d,m,o,r,y);
10       return new Assignment(s,e,n,d,m,o,r,y);
11     } else throw Muli.fail();
12   } else throw Muli.fail(); }
```

Listing 15.4: Muli search region implementing the Send More Money Puzzle; class headers and helper functions omitted.

Application	Environment	Minimum	Maximum	Average	Median
Send More Money	Muli (on SJVM)	8.99	44.98	18.22	17.85
	Java (on JVM)	324.87	426.17	338.40	337.25
Safe Lock Key	Muli (on SJVM)	6.59	38.10	12.80	12.12
	Java (on JVM)	24.76	33.64	25.44	25.03

Table 15.2: Comparison of sample implementations by execution time (in milliseconds).

Key puzzle, also in each pure Java and Muli. In order to exclude possible overhead of the SJVM, we have executed the pure Java application on a regular OpenJDK JVM (version 1.8.0_131). The Muli application has been executed on our SJVM, using the JaCoP-based finite domain solver in the solver component. All experiments were executed on an Intel Core i5-5200U CPU, using Ubuntu 16.04.3 with a 4.10.0 x64 Kernel. Each application attempts to solve the puzzle 510 times. The first 10 results are dropped in order to disregard effects of Just-In-Time compilation of the enclosing JVM. The aggregated execution times are provided in Table 15.2.

The results indicate that, for Send More Money, the constraint-logic OO solution, while more elegant to write, is consistently more than an order of magnitude faster at finding a solution for the problem at hand. Similarly, except for the maximum runtime values, the Safe Lock Key search implementation is faster on Muli than on Java. The relative difference between minimum and maximum is higher for Muli, which could be due to increased need for garbage collection over constraint representations, whereas the regular Java version uses primitive values only. Nevertheless, we can see a significant

```
1 Muli.getAllSolutions(() -> { int n free;
2   int factorial = fact(n);
3   String test = "assertEquals(" + factorial
4     + ", fact("+n+"));");
5   return test; });
```

Listing 15.5: Modification of the factorials search region from Listing 15.2 to generate JUnit assertions for testing `fact()`.

improvement in performance for the Muli variant, that can be attributed to constraint propagation by the finite domain solver, allowing for more efficient search than a simple backtracking mechanism.

Another highly interesting application scenario for Muli is automated glass-box test case generation, as it is a prime example for incrementally added constraints [EMK12]: In order to generate JUnit assertions for a Java method, we need to create an appropriate output from free parameters and the method's results. For the factorial method from Listing 15.2, this can be done by changing the search region as demonstrated in Listing 15.5. Moreover, consider that Listing 15.5 could be extended to generating integration test cases by calling sequences or compositions of methods.

As `fact(n)` has an infinite search space, writing `test` into a file before returning it results in output in that file until execution is interrupted.

Furthermore, the example in Listing 15.3 demonstrates the simpler programming style that Muli facilitates. In contrast, writing a similar program in Java is much more challenging and requires three times as many lines of code, particularly for handling backtracking and negation in case that the constraint store is rendered inconsistent by a recent addition (Listing 15.6), thus introducing more potential for implementation mistakes. Moreover, the pure Java version requires a constraint solver-specific implementation for logic variables, constraint definition, and backtracking, so that the used constraint solver cannot easily be exchanged. In contrast, the Muli program makes use of implicit backtracking and negation by the SJVM. Established Java operators and the **`if`** control structure are used to add constraints, as well as the primitive **`int`** type to declare both free and non-free variables. Last but not least, the constraint solver is exchangeable by configuring the SJVM's solver component, thus facilitating later migrations to more advanced constraint solvers.

From the aforementioned examples, we conclude that constraint-logic OO programming is able to reduce runtime, at least compared to pure Java search applications.

```
1  import org.jacop.constraints.*;
2  import org.jacop.core.*;
3  class JacopIncrementalArgs {
4    public static void main(String[] args) {
5      Store store = new Store();
6      int i = 0;
7      IntVar x = new IntVar(store, ”x”,
8        -1000000, 1000000);
9      while (i < args.length) {
10       // Prepare for later backtracking.
11       int backtrackingLvl = store.level+1;
12       store.setLevel(backtrackingLvl);
13       // Create constraints from user input.
14       int c1 = Integer.parseInt(args[i++]);
15       int c2 = Integer.parseInt(args[i++]);
16       Constraint cons1 = new XltC(x, c1);
17       Constraint cons2 = new XgtC(x, c2);
18       store.impose(cons1);
19       store.impose(cons2);
20       if (!store.consistency()) {
21         // Backtrack and add negations.
22         store.removeLevel(backtrackingLvl);
23         store.setLevel(backtrackingLvl);
24         XgteqC cons1n = new XgteqC(x, c1);
25         XlteqC cons2n = new XlteqC(x, c2);
26         Or or = new Or(cons1n, cons2n);
27         store.impose(or);
28         store.consistency();
29         break; } }
30     System.out.println(x); } }
```

Listing 15.6: Implementation of adding constraints from user input incrementally and of manual backtracking requires more effort in Java (using the JaCoP solver) than in Muli.

Moreover, the constraint-logic OO programming style enables applications that interleave constraint-logic parts with imperative parts (such as requesting additional user input) during search, which is far less convenient and more error prone to achieve without an SJVM using pure Java with a solver library.

Having free variables creates a little overhead, since information about them is stored in bytecode and the SJVM has to accommodate symbolic types and expressions. However, this overhead is limited to logic variables and expressions that involve them, whereas deterministic computations that do not use these concepts are not handled symbolically. Therefore, no overhead is added for non-symbolic, deterministic applications.

While we are certain that the idea of constraint-logic OO programming is beneficial to a range of use cases, we acknowledge that some limitations apply to our results. Our current implementation supports constraints over primitive variable types only. Constraint-logic OO programming would benefit from support for solving constraints over object graphs or arrays. However, this is an endeavour that we will tackle in upcoming work. Nevertheless, recall that Muli does support constraint-logic programming using (primitive) object fields already, so Muli applications are not limited to non-OO features.

15.5 Related Work

There are many libraries that add constraint programming to Java (and, therefore, to JVM languages). Choco [PFL16] and OptaPlanner [Opt17] are examples that seem to have gained attraction from research and industry. However, their interfaces are non-standardised, so they can be unintuitive to use and hard to exchange. The finalised JSR 331 defines a standard for constraint programming and solving in Java, but efforts seem to have ceased since 2012 [Fel12]. Furthermore, they share the disadvantage that they always work somewhat separate from the Java program that leverages them. Thus, imperative or OO code are not seamlessly integrated. Instead, the imperative code can only invoke the constraint solver, but it cannot intervene with the search for a solution. In contrast, Muli allows to integrate search and an imperative style tightly, allowing to freely mix both paradigms in the most appropriate way to solve a given problem. Muli uses constraint solver libraries internally to check the consistency of branching constraints in order to skip infeasible paths, as well as to find specific solutions after sets of constraints have been collected.

Moreover, there are several approaches which add OO features such as inheritance to a (constraint) logic programming language, often to Prolog. For instance, Visual Prolog

extends Prolog by OO features, primarily aiming at artificial intelligence applications [Sco10]. Similarly, McCabe presents an OO language based on Prolog [McC92]. Another OO layer on top of Prolog is presented by Shapiro and Takeuchi, focussing on concurrency [ST83]. Similarly, Prolog++ adds OO features to Prolog [Mos94]. tuProlog approaches the integration of Prolog and Java differently, by providing a Prolog implementation written in Java [DOR05]. This enables Prolog programs to run on the JVM, thus facilitating integrated applications without a need for the JNI as well. Yet, this results in applications implemented using two different languages, running in two separate environments on the JVM. Therefore, non-deterministic program parts are separated from imperative program parts and cannot be mixed. As an example of a non-Prolog-based language, Mozart/Oz is a constraint language that, among concurrency and lazy evaluation, also offers OO features [Van+03]. Again, this approach has a declarative focus.

Compared to Muli, all these approaches have a different flavour and runtime behaviour. They are mainly declarative languages which simulate object-orientation. Assignments and state changes are not provided (natively). Although approaches adding OO features to a (constraint) logic programming language are interesting for declarative programmers, it is unlikely that they will receive much attention from mainstream OO developers due to the unfamiliar programming style.

Closest to Muli are approaches that extend OO programming with concepts from constraint-logic programming. All those that we are aware of are aimed at automated software testing. This includes glass-box test case generators, such as Muggl [MK09] and IBIS [DM03], that add symbolic execution and constraint programming to Java bytecode execution. Pex works similarly for the .NET intermediate language [TH08]. Quite recently we discovered Seer, an application that intends to add symbolic execution and constraint solving to the imperative programming language Rust [Ren17]. Although Seer is currently in an initial stage and also aimed towards software generation, development is ongoing and could be similarly useful in the context of Rust, as our approach is in that of Java.

Other work already attempted enabling logic programming in Java by extending Muggls symbolic VM into a self-contained runtime [MK11]. However, their approach falls short in the handling of logic variables, as only class fields can be declared free using annotations. Entire methods are declared either searching or non-searching by annotation, so defining search regions is tedious. The annotation is barely visible, thus harming effective understanding of an application.

There are also several integration approaches involving further programming paradigms. Contemporary OO languages have added features that originate in functional programming, making combinations of functional programming and OO programming

increasingly prominent among OO software developers. In Java, a combination of lambda abstractions and the Stream API enables development in a functional programming style where appropriate [UFM14], whereas LINQ delivers a similar approach to C# [MBB06]. Also, Scala integrates functional and OO programming and also runs on the JVM [Ode+17].

Functional and constraint-logic programming have also been integrated. For example, Curry combines both paradigms using a Haskell-based syntax extended by logic variables, non-determinism, and encapsulated search [Han+95; AJ16; Bra+11; LK99]. Encapsulated search and constraint definition concepts of Curry provided ideas for our constraint-logic OO language, although the implementation of these concepts in an imperative context differs from theirs.

15.6 Conclusions and Future Work

Muli achieves the integration of imperative OO programming with constraint-logic programming. Although Muli is a new programming language, it preserves the Java syntax and the functionality of Java concepts. Moreover, programs written in Muli compile to bytecode that could also be read by a regular JVM, albeit not being very useful. Instead, it is accompanied by a custom SJVM runtime environment that supports symbolic execution, encapsulated search, backtracking, and constraint solving within Muli programs. Incidentally, due to our choice of Java for the implementations of compiler and SJVM, our prototype is just as platform-independent as Java.

Compared to other attempts at adding constraint solving to Java, our approach is novel in that the entire runtime is capable of searching and backtracking (but only when requested explicitly, i. e. within encapsulated search). The Muli runtime environment implicitly traverses search spaces that are described by search regions in Muli programs and collects solutions, so that they can be re-used in later parts of the programs, including subsequent search regions. Implicit traversal gives developers the advantage to selectively mix declarative constraint definition and imperative control flows if appropriate, which is not possible with other approaches in Java.

All in all, we believe that our approach achieves a smooth integration of constraint-logic and OO paradigms, thus enabling Java developers to leverage the benefits of constraint-logic programming in a native Java style. In addition, developers are no longer required to manually invoke Java constraint solvers, or to bother with integrating external search applications, Prolog or otherwise, via JNI. This avoids a lot of programming effort and reduces potential for mistakes.

We made the resulting classpath library, compiler, and runtime available on GitHub as free software[34]. We welcome contributions of any kind, including reports of any issues that may arise when you are trying out our approach.

Our work results in further novel ideas that we have yet to tackle. The degree of freedom that imperative (OO) programming already offers makes it impossible to decide for a constraint-logic OO program whether its search regions create an infinite search space. Currently, this can cause encapsulated search to never terminate, which is undesirable. Future work will attempt to work on separating producer and consumer of solutions to achieve means to interrupting encapsulated search, to providing intermediate solutions, and to continue search.

Furthermore, future work will tackle solving for constraints involving non-primitive variables, such as solving of arrays, objects, and object graphs.

Our approach would be very useful in solving such search problems that are also optimisation problems. For now, an application would first have to compute all solutions and then iterate over all solutions to find the optimum. Here, support for optimisation problems during encapsulated search would be very convenient. However, this is not trivial since branching conditions are found dynamically, so that we never know the entire optimisation problem unless we execute it. We hope to find a sophisticated solution that integrates symbolic execution with optimisation problems.

References

[AJ16] Sergio Antoy and Andy Jost. ‘A New Functional-Logic Compiler for Curry: Sprite’. In: *LOPSTR*. 2016.

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller and Fabian Reck. ‘KiCS2: A New Compiler from Curry to Haskell’. In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: http://dx.doi.org/10.1007/978-3-642-22531-4_1.

[DM03] J. Doyle and C. Meudec. ‘IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming’. In: *2nd PPPJ* (2003), pp. 55–58.

[DOR05] Enrico Denti, Andrea Omicini and Alessandro Ricci. ‘Multi-paradigm Java-Prolog integration in tuProlog’. In: *Science of Computer Programming* 57.2 (2005), pp. 217–250. ISSN: 01676423. DOI: 10.1016/j.scico.2005.02.001.

[34] https://github.com/wwu-pi/muli

[EH07] Torbjörn Ekman and Görel Hedin. 'The JastAdd extensible Java compiler'. In: *OOPSLA*. 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297105.1297029.

[EMK12] Marko Ernsting, Tim A. Majchrzak and Herbert Kuchen. 'Test Case Generation and Dynamic Mixed-Integer Linear Arithmetic Constraint Solving'. In: *21st WFLP*. 2012.

[Fel12] Jacob Feldman. *JSR 331: Constraint Programming API*. 2012. URL: https://jcp.org/en/jsr/detail?id=331 (visited on 11/06/2017).

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 09/06/2017).

[Han+95] M Hanus, H Kuchen, J J Moreno-Navarro, JR Votano, M Parham and LH Hall. 'Curry: A Truly Functional Logic Language'. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Han97] Michael Hanus. 'A Unified Computation Model for Functional and Logic Programming'. In: *POPL 97*. 1997, pp. 80–93.

[KO08] Goh Kondoh and Tamiya Onodera. 'Finding bugs in Java Native Interface programs'. In: *ISSTA '08* (2008), p. 109. DOI: 10.1145/1390630.1390645.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 10844309. DOI: 10.1145/785411.785416.

[Lem+04] Christoph Lembeck, Rafael Caballero, Roger A. Müller and Herbert Kuchen. 'Constraint Solving for Generating Glass-Box Test Cases'. In: *Proceedings WFLP '04*. 2004, pp. 19–32.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 09/06/2017).

[LK99] Wolfgang Lux and Herbert Kuchen. 'An Efficient Abstract Machine for Curry'. In: *Informatik '99 - Informatik überwindet Grenzen*. Ed. by K Beiersdörfer, G Engels and W Schäfer. Springer Verlag, 1999, pp. 390–399.

[Lou93] Kenneth C Louden. *Programming Languages: Principles and Practice*. Ed. by Patricia Adams. Belmont, CA, USA: Wadsworth Publ. Co., 1993. ISBN: 0534932770.

[MBB06] Erik Meijer, Brian Beckman and Gavin Bierman. 'LINQ: Reconciling Objects, Relations and XML in the .NET Framework'. In: *ACM SIGMOD International Conference on Management of data*. 2006, p. 706. ISBN: 1595932569. DOI: 10.1145/1142473.1142552.

[McC92] F. G. McCabe. *Logic and Objects*. Prentice-Hall international series in computer science. Prentice Hall, 1992. ISBN: 9780135360798.

[MK09] T. A. Majchrzak and H. Kuchen. 'Automated Test Case Generation Based on Coverage Analysis'. In: *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. 2009, pp. 259–266. ISBN: 978-0-7695-3757-3. DOI: 10.1109/TASE.2009.33.

[MK11] Tim A Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7.

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/ (visited on 13/06/2017).

[Opt17] OptaPlanner Team. *OptaPlanner User Guide, Version 7.0.0*. 2017. URL: https://docs.optaplanner.org/7.0.0.Final/optaplanner-docs/html%7B%5C_%7Dsingle/.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes. 2016. URL: http://www.choco-solver.org.

[Ren17] David Renshaw. *Seer: Symbolic Execution Engine for Rust*. 2017. URL: https://github.com/dwrensha/seer (visited on 13/06/2017).

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010. ISBN: 9781432749361.

[ST83] Ehud Shapiro and Akikazu Takeuchi. 'Object oriented programming in Concurrent Prolog'. In: *New Generation Computing* 1.1 (1983), pp. 25–48. ISSN: 02883635. DOI: 10.1007/BF03037020.

[Sta17] Stack Overflow. *Developer Survey Results 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017%7B%5C#%7Dtechnology (visited on 11/06/2017).

[TH08] Nikolai Tillmann and Jonathan de Halleux. 'Pex: White Box Test Generation for .NET'. In: *2nd International Conference on Tests and Proofs*. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9.

[TIO17] TIOBE Software BV. *TIOBE Index*. 2017. URL: https://www.tiobe.com/tiobe-index/ (visited on 11/06/2017).

[Tri12] Markus Triska. 'The Finite Domain Constraint Solver of SWI-Prolog'. In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316.

[UFM14] Raoul-Gabriel Urma, Mario Fusco and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT: Manning Publications Co., 2014. ISBN: 1617291994, 9781617291999.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. ISSN: 14710684. DOI: 10.1017/S1471068403001741.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set*. Tech. rep. Menlo Park: SRI International, 1983.

16

An Operational Semantics for Constraint-Logic Imperative Programming

Jan C. Dageförde[*] · Herbert Kuchen[*]

Citation Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

Abstract Object-oriented (OO) languages such as Java are the dominating programming languages nowadays, among other reasons due to their ability to encapsulate data and operations working on them, as well as due to their support of inheritance. However, in contrast to constraint-logic languages, they are not particularly suited for solving search problems. During development of enterprise software, which occasionally requires some search, one option is to produce components in different languages and let them communicate. However, this can be clumsy.

As a remedy, we have developed the constraint-logic OO language Muli, which augments Java with logic variables and encapsulated search. Its implementation is based on a symbolic Java virtual machine that supports constraint solving and backtracking. In the present paper, we focus on the non-deterministic operational semantics of an imperative core language.

Keywords Java · operational semantics · encapsulated search · programming paradigm integration.

[*]University of Münster, Germany

16.1 Introduction

Contemporary software development is dominated by object-oriented (OO) programming. Its programming style benefits most industry applications by providing e. g. inheritance and encapsulation of structure and behaviour, since these concepts can positively contribute towards reusability and maintainability [Lou93]. Nevertheless, some industry applications require search, for which constraint-logic programming is more suited than OO (or imperative) programming. However, developing applications that integrate both worlds, e. g. a Java application using a Prolog search component via Java Native Interface (JNI), is tedious and error-prone [KO08].

For that reason, we propose the *Münster Logic-Imperative Programming Language (Muli)*, integrating constraint-logic programming with OO programming in a novel way. Based on Java, it adds logic variables and encapsulated search to the language, supported by constraint solvers and non-deterministic execution on a symbolic Java virtual machine (JVM). The symbolic JVM adapts concepts from the Warren Abstract Machine, such as choice points and trail [War83]. Muli's tight integration of both paradigms facilitates development of applications whose business logic is implemented in Java, but which also require occasional search, such as operations research applications [Hoo06].

In this paper, we describe a reduction semantics for a core subset of Muli. In particular, the interaction of imperative statements, free variables, and non-determinism is of interest. For simplicity, this core language abstracts from inheritance, multi-threading, and reflection, because those features do not exhibit interesting behaviour w. r. t. our semantics. The formulated semantics is helpful to get an understanding of the mechanics behind concepts that are novel to imperative and OO programming, and serves as a formal basis for implementing the symbolic JVM. It can also be used for reasoning about applications developed in Muli.

To that end, our paper is structured as follows. We provide an overview of the new language and its concepts in Section 16.2. Section 16.3 formalises the operational semantics of the core language. An example evaluation using this semantics is shown in Section 16.4. Section 16.5 presents a discussion of our concepts. Related work is outlined in Section 16.6. We then conclude in Section 16.7 and provide an outlook towards further research.

16.2 Language Concepts

The Muli language is derived from Java 8. We do not change existing concepts and features of Java, so that Muli also benefits from Java's well-known and well-received

features, such as OO and managed memory. Instead, the language is defined by its additions to Java, i. e. Muli is a superset of Java.

Muli adds the concept of *free variables*, i. e. variables that are declared and instantiated, but not to a particular value. Instead, they are treated symbolically and can be used in statements and expressions. *Constraints* on symbolic variables and expressions are imposed during symbolic execution of conditional statements. For example, an `if` statement with a condition that involves insufficiently constrained variables results in multiple branches that can be evaluated. Conceptually, we can non-deterministically choose a branch and evaluate it. Our implementation considers all these branches using backtracking and a (complete!) iterative deepening depth-first search strategy. This is supported by a specialised symbolic JVM that records choice points for each non-deterministic branch.

Furthermore, we enforce that non-determinism only takes place inside *encapsulated search* regions, whereas code outside encapsulation is executed deterministically. This ensures that non-determinism is not introduced by accident, intending not to harm the understanding of known Java concepts. Furthermore, this ensures that the overall application exits in a single state. In contrast, unencapsulated symbolic execution could result in multiple exit states, which could cause difficulties on the side of the caller. Encapsulation is expressed by using either of the `getAllSolutions` and `getOneSolution` operators. The logic of encapsulated search is described by *search regions* that are implemented as methods, e. g. as lambda abstractions, in order to defer their evaluation until encapsulation begins.

Solutions of encapsulated search are defined by values or expressions returned from search regions. Due to non-determinism, multiple solutions can be returned from search. Additionally, we introduce the special statement **`fail`**`;`, whose evaluation results in immediate backtracking in the symbolic JVM without recording a solution for the current branch.

From a syntactic perspective, these concepts extend Java only minimally. The resulting syntax of Muli can best be demonstrated using an example. Listing 16.1 exhibits a Muli method `log()` that searches for the logarithm of a number x to the base 2 using a free variable y and a method `pow` that calculates b^y imperatively, which is constrained to be equal to x.

Let us assume that the considered search region consists of a call to `log`, e.g. `log(4)`. When calling `log` with a given x, the free variable y is created and then passed to `pow` that calculates the power b^y symbolically, as y is free. Therefore, it returns a value that is accompanied by a set of accumulated constraints from which this particular value

```
1  int log(int x) {
2    int y free;
3    if (pow(2,y) == x) return y;
4    else fail; }
5  int pow(int b, int y) {
6    int i; int r; i = 0; r = 1;
7    while (i < y) {
8      r = r * b; i = i + 1; }
9    return r; }
```

Listing 16.1: Non-deterministic computation of the logarithm of a number to the base 2 using (core) Muli.

follows.[35] Consequently, `log` computes the logarithm by defining a constraint system using an imperative method that calculates the power.

If the variables involved in a branching condition (of **if** or **while** in Listing 16.1) are not sufficiently constrained, one of the feasible branches is chosen non-deterministically. Actually, our symbolic JVM would try them systematically one after the other, aided by a backtracking mechanism. When selecting a branch, the corresponding condition is added to the constraint store and consistency is checked. For example, `while (i < y)` can be either true or false as y is a free variable. As a result, one branch assumes the condition to be true and therefore adds the constraint $i < y$ to the constraint store by imposing a conjunction of the existing store and the new constraint. In contrast, the second branch assumes it to be false and therefore adds the negated condition as a constraint. If an added constraint renders the store inconsistent, backtracking occurs, i. e. that branch is pruned and execution continues with a subsequent branch. Similarly, backtracking occurs when a solution is found so that the next branch can be evaluated to find further solutions. Muli's encapsulated search operators use lazy streams to return collected solutions to the surrounding deterministic computation, such that the surrounding computation can decide how many solutions it wants to obtain.

[35]In other problems the return value could be a symbolic expressions if the accumulated constraints do not reduce the return value's domain to a concrete value.

16.3 A Non-Deterministic Operational Semantics of Muli

Muli is an extension to Java and therefore intends to fully support all Java functionality. In fact, all Muli programs even compile to regular JVM bytecode that can be parsed and executed by a regular JVM (but incorrectly), and all Java programs can be executed correctly by Muli's symbolic JVM. Outside of encapsulated search, execution in Muli is deterministic and replicates the behaviour of a standard JVM [Lin+15]. Inside encapsulation, search regions are executed non-deterministically. This changes the semantics of Java and adds subtleties that need to be explicated, particularly regarding the interaction of imperative statements, free variables, and non-determinism. Therefore, we formally define the semantics for non-deterministic evaluation of search regions.

For the purpose of describing a (non-deterministic) operational semantics of Muli, we focus on an imperative, procedural subset of Java (and Muli). This concise subset allows us to focus on the interaction between imperative and constraint-logic programming. It therefore abstracts from some features that are expected from Java but that would not contribute to the discussion in the present paper, such as inheritance.[36] Furthermore, this semantics abstracts from the execution of deterministic program parts and therefore does not prescribe an implementation for the encapsulation operators, `getAllSolutions` and `getOneSolution`.

Let us first describe the syntax of our core language. We will use variables taken from a finite set $Var = \{x_1, \dots, x_m\}$, for simplicity all of type integer ($m \in \mathbb{N}$). Also let $Op = AOp \cup BOp \cup ROp = \{+, -, *, /\} \cup \{\&\&, ||\} \cup \{==, !=, <=, >=, <, >\}$ be a finite set of arithmetic, boolean, and relational operation symbols, respectively. We focus on binary operation symbols. Furthermore, $\mathcal{M}$ is a finite set of methods.[37]

The syntax of arithmetic expressions and boolean expressions as well as statements can be described by the following grammar. *AExpr*, *BExpr*, and *Stat* denote the sets of all arithmetic expressions, boolean expressions, and statements, respectively, which can be constructed by the rules of this grammar.

$$e ::= c \mid x \mid e_1 \oplus e_2 \mid m(e_1, \dots, e_k)$$
$$\text{where } c \in \mathbb{Z},\ x \in Var,\ e_1, \dots, e_k \in AExpr,\ \oplus \in AOp,\ m \in \mathcal{M},\ k \in \mathbb{N},$$

$$b ::= e_1 \odot e_2 \mid b_1 \otimes b_2 \mid \texttt{true} \mid \texttt{false}$$

[36] Nevertheless, Muli's symbolic JVM supports these features exactly according to the JVM specification [Lin+15] (but does not add interesting details w. r. t. non-determinism).

[37] In fact they are functions, since we ignore object-orientation in this presentation.

where $e_1, e_2 \in AExpr,\ b_1, b_2 \in BExpr,\ \odot \in ROp,\ \otimes \in BOp,$

$s ::= \texttt{;}\ \mid\ \texttt{int}\ x\texttt{;}\ \mid\ \texttt{int}\ x\ \texttt{free;}\ \mid\ x = e\texttt{;}\ \mid\ e\texttt{;}\ \mid\ \{s\}\ \mid\ s_1\ s_2\ \mid$
$\texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2\ \mid\ \texttt{while}\ (b)\ s\ \mid\ \texttt{return}\ e\texttt{;}\ \mid\ \texttt{fail;}$

where $x \in Var, e \in AExpr,\ b \in BExpr,\ s, s_1, s_2 \in Stat.$

Note, in particular, the possibility to create free logic variables by `int` x `free;`.

After describing the syntax of the core language, let us now define its semantics. In the sequel, let $\mathscr{A} = \{\alpha_0, \dots, \alpha_n\}$ be a finite set of memory addresses ($n \in \mathbb{N}$). Moreover, let

$$Tree(\mathscr{A}, \mathbb{Z}) = \mathscr{A} \cup \mathbb{Z} \cup \{\oplus(t_1, t_2) \mid t_1, t_2 \in Tree(\mathscr{A}, \mathbb{Z}), \oplus \in Op\}$$

be the set of all symbolic expression trees with addresses and integer constants as leaves and operation symbols as internal nodes.

We provide a reduction semantics, where the computations depend on an environment, a state, and a constraint store. Let $Env = (Var \cup \mathscr{M}) \rightarrow (\mathscr{A} \cup (Var^* \times Stat))$ be the set of all environments, mapping each variable to an address and each function to a representation $((x_1, \dots, x_k), s)$ that describes its parameters and code, with the additional restriction that elements of Env may neither map variables to parameters and code nor functions to addresses. We consider functions to be in global scope and define a special initial environment $\rho_0 \in Env$ that maps functions to their respective parameters and code. Moreover, let $\Sigma = \mathscr{A} \rightarrow (\{\bot\} \cup Tree(\mathscr{A}, \mathbb{Z}))$ be the set of all possible memory states. In $\sigma \in \Sigma$, a special address α_0 with $\sigma(\alpha_0) = \bot$ is reserved for holding return values of method invocations. Furthermore, $CS = \{\texttt{true}\} \cup Tree(\mathscr{A}, \mathbb{Z})$ is the set of all possible constraint store states. Since constraints are specific boolean expressions, only conjunctions and relational operation symbols such as == and > will appear at the root of such a tree.

In the sequel, $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$; if needed, we will also add discriminating indices. We will use the notation $a[x/d]$ when modifying a state or environment a, meaning

$$a[x/d](b) = \begin{cases} d & \text{, if } b = x \\ a(b) & \text{, otherwise.} \end{cases}$$

A free variable is represented by a reference to its own location in memory. Consequently, $\sigma(\rho(x)) = \rho(x)$ if x is a free variable. Initially, a constraint store γ is empty, i. e. it is initialised with `true`. During execution of a program, constraints may incrementally be added to the store. This is done by imposing a conjunction of the existing constraints and a new constraint, thus replacing the constraint store by the new conjunction. As a result, the constraint store is typically described by a conjunction of atomic boolean

expressions. We treat the constraint solver as a black box. In our implementation, we use the external constraint solver JaCoP [Kuc03] in its most recent version 4.4. In fact, the constraint solver is exchangeable and any solver implementation fulfilling our requirements (particularly incremental adding/removal of constraints) can be used. [38]

Note that our definition of functions does not fully cover the concept of methods in object-oriented languages, since we abstract from classes and, therefore, inheritance. However, a function in our semantics can be compared to a static method, since a function in this semantics can access and modify its own arguments and variables, but not instance variables of an object. Static fields could be modelled as global variables, i. e. further entries in ρ_0.

Since classes, inheritance, instance variables, and static variables have little influence on the interaction between imperative statements, free variables, and non-determinism, object orientation can be considered (almost) orthogonal to our work.

16.3.1 Semantics of Expressions

Let us start with the semantics of expressions. The semantics of expressions is described by a relation $\rightarrow \ \subset (Expr \times Env \times \Sigma \times CS) \times ((\mathbb{B} \cup Tree(\mathscr{A}, \mathbb{Z})) \times \Sigma \times CS)$, which we use in infix notation. Note that evaluating an expression can, in general, change state and constraint store as a side effect, although only the Invoke rule actively does so. We will point out expressions that make use of this, whereas the others merely propagate changes (if any) resulting from the evaluation of subexpressions.

The treatment of constants and variables is trivial.

$$\langle c, \rho, \sigma, \gamma \rangle \rightarrow (c, \sigma, \gamma), \text{ if } c \in \mathbb{Z} \cup \mathbb{B} \tag{Con}$$

$$\langle x, \rho, \sigma, \gamma \rangle \rightarrow (\sigma(\rho(x)), \sigma, \gamma) \tag{Var}$$

Nested arithmetic expressions without free variables are evaluated directly, whereas expressions comprising free variables result in a (deterministic) unevaluated (!) symbolic expression ($\in Tree(\mathscr{A}, \mathbb{Z})$).

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2), \quad v_1, v_2, v = v_1 \oplus v_2 \in \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_2, \gamma_2)} \tag{AOp1}$$

[38] A very simple constraint solver could just take equality constraints into account. In this case, $\gamma \models x == v$, if $\gamma = b_1 \wedge \ldots \wedge b_k$ and for some $j \in \{1, \ldots, k\}$ $b_k = (x == v)$.

$$\frac{\langle e_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1\rangle \to (v_2, \sigma_2, \gamma_2),\ \ \{v_1, v_2\} \nsubseteq \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma\rangle \to (\oplus(v_1, v_2), \sigma_2, \gamma_2)} \quad \text{(AOp2)}$$

A boolean expression of the form $e_1 \odot e_2$ is evaluated analogously.

Coherent with Java, conjunctions of boolean expressions are evaluated non-strictly. The rules for the non-strict boolean disjunction operator `||` are defined analogously to the following rules for `&&`.

$$\frac{\langle b_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1),\ \ \gamma \vDash \neg v_1}{\langle b_1\ \texttt{\&\&}\ b_2, \rho, \sigma, \gamma\rangle \to (\texttt{false}, \sigma_1, \gamma_1)} \quad \text{(And1)}$$

$$\frac{\langle b_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1),\ \ \gamma \nvDash \neg v_1,\ \ (b_2, \sigma_1, \gamma_1) \to (v_2, \sigma_2, \gamma_2)}{\langle b_1\ \texttt{\&\&}\ b_2, \rho, \sigma, \gamma\rangle \to (\wedge(v_1, v_2), \sigma_2, \gamma_2)} \quad \text{(And2)}$$

We consider a function invocation to be an expression as well, as the caller can use its result in a surrounding expression. Evaluation of the function is likely to result in a state change as well as in additions to the constraint store. Invoking m implies that its description $\rho(m)$ is looked up and corresponding fresh addresses $\alpha_1, \dots, \alpha_k$, one for each of its k parameters, are created. The corresponding memory locations are initialised by the caller. Note that the respective values can contain free variables. $\sigma_{k+1}(\alpha_0)$ will contain the return value from evaluating the `return` statement in the body, whose semantics will be defined later (cf. rule Ret). As the compiler enforces the presence of a `return` statement, we can safely assume that $\sigma_{k+1}(\alpha_0)$ holds a value after reducing s. Invoke resets that value to $\perp$ for further evaluations within the calling method. We use the shorthand notation $\bar{a}_k = (a_1, \dots, a_k)$ for vectors of k elements.

$$\frac{\begin{array}{r}\langle e_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1),\ \ \langle e_2, \rho, \sigma_1, \gamma_1\rangle \to (v_2, \sigma_2, \gamma_2), \dots,\\ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1}\rangle \to (v_k, \sigma_k, \gamma_k),\ \ \rho(m) = (\bar{x}_k,\ s),\\ \langle s, \rho_0[{}^{\bar{x}_k}/{}_{\bar{\alpha}_k}], \sigma_k[{}^{\bar{\alpha}_k}/{}_{\bar{v}_k}], \gamma_k\rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}),\ \ \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle m(e_1, \dots, e_k), \rho, \sigma, \gamma\rangle\ \to (r, \sigma_{k+1}[\alpha_0/\perp], \gamma_{k+1})} \quad \text{(Invoke)}$$

16.3.2 Semantics of Statements

Next, we describe the semantics of statements by a relation $\rightsquigarrow\ \subset (\mathit{Stat} \times \mathit{Env} \times \Sigma \times \mathit{CS}) \times (\mathit{Env} \times \Sigma \times \mathit{CS})$, which we also use in infix notation.

A variable declaration changes the environment by reserving a fresh memory location α for that variable. A free variable is represented by a reference to its own location. Enclosing declarations in a block ensures that changes of the environment stay local.

$$\langle \texttt{int}\ x;, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha], \sigma, \gamma) \quad \text{(Decl)}$$

$$\langle \texttt{int } x \texttt{ free;}, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho[x/\alpha], \sigma[\alpha/\alpha], \gamma) \tag{Free}$$

$$\frac{\langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle \{ s \}, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1)} \tag{Block}$$

As a particularity of a constraint-logic OO language, an assignment `x = e` cannot just overwrite a location in memory corresponding to `x`, since this might have an unwanted side effect on constraints that involve `x` and refer to its former value. This side effect might turn such constraints unsatisfiable after they have been imposed and checked, thus leaving a currently executed branch in an inconsistent state. We avoid this by assigning a new memory address α_1 to the variable on the left-hand side. At the new address, we store the result from evaluating the right-hand side. Consequently, old constraints or expressions that involve the former value of `x` are deliberately left untouched by the assignment. In contrast, later uses of the variable refer to its new value. The environment is updated to achieve this behaviour.

$$\frac{\langle e, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1)}{\langle x = e, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho[x/\alpha_1], \sigma_1[\alpha_1/v], \gamma_1)} \tag{Assign}$$

Since the syntax does not enforce that no statements follow a `return` statement, we provide sequence rules that take into account that the state may hold a value in α_0 (indicating a preceding `return`) or not ($\perp$). Further statements are executed iff the latter is the case. Otherwise, further statements are discarded as a preceding `return` has already provided a result in α_0.

$$\frac{\langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) == \perp, \quad \langle s_2, \rho_1, \sigma_1, \gamma_1 \rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)}{\langle s_1 \ \ s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)} \tag{Seq}$$

$$\frac{\langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) \neq \perp}{\langle s_1 \ \ s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \tag{SeqFin}$$

The two following rules for if-statements introduce non-determinism in case that the constraints neither entail the branching condition nor its negation.[39]

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \not\vdash \neg v, \quad \langle s_1, \rho, \sigma_1, \gamma_1 \wedge v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if } (b) \ s_1 \texttt{ else } s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \tag{If$_t$}$$

[39]In the implementation, the applicability of these rules will depend on the constraint propagation abilities of the employed constraint solver. We discuss the implications in Section 16.5.

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \to (v, \sigma_1, \gamma_1),\ \ \gamma_1 \nvDash v,\ \ \langle s_2, \rho, \sigma_1, \gamma_1 \wedge \neg v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if } (b)\ s_1 \texttt{ else } s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \quad (\mathrm{If}_f)$$

As with `if`, `while` can also behave non-deterministically.

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \to (v, \sigma_1, \gamma_1),\ \ \gamma_1 \nvDash \neg v,\ \ \langle s, \rho, \sigma_1, \gamma_1 \wedge v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2),\ \langle \texttt{while } (b)\ s, \rho_1, \sigma_2, \gamma_2 \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)}{\langle \texttt{while } (b)\ s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)} \quad (\mathrm{Wh}_t)$$

$$\frac{\langle b, \rho, \sigma, \gamma \rangle \to (v, \sigma_1, \gamma_1),\ \ \gamma_1 \nvDash v}{\langle \texttt{while } (b)\ s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1 \wedge \neg v)} \quad (\mathrm{Wh}_f)$$

All branching rules If_f, If_t, Wh_f, and Wh_t could be accompanied by more efficient ones that deterministically choose a branch if its condition does not involve free variables, i. e. without having to consult the constraint store. We omit these rules in an effort to keep our definitions concise, as the provided ones can also handle these cases.

We assume that the code of a user-defined function is terminated by a `return` statement, i. e. its existence has to be ensured by the compiler. The corresponding return value is supplied to the caller by storing it in α_0, causing remaining statements of the function to be skipped (cf. rule SeqFin), and letting the caller extract the result from α_0 (cf. rule Invoke). The `return` statement is handled as follows:

$$\frac{\langle e, \rho, \sigma, \gamma \rangle \to (v, \sigma_1, \gamma_1)}{\langle \texttt{return } e, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1[\alpha_0/v], \gamma_1)} \quad (\mathrm{Ret})$$

Furthermore, we do not define an evaluation rule involving a `fail` statement. This is intentional, as the evaluation of such a statement leads to a computation that fails immediately.

The following (optional) substitution rule allows to simplify expressions and results.

$$\frac{\gamma \vDash \gamma(\alpha) == v,\ \ \langle s, \rho, \sigma[\alpha/v], \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \quad (\mathrm{Subst})$$

When variables are not sufficiently constrained to concrete values, *labeling* can be used to substitute variables for values that satisfy the imposed constraints [FA03]. This non-deterministic rule is applied with the least priority, i. e. it should only be used if no other rule can be applied. Otherwise, it would result in a lot of non-deterministic branching, thus preventing the constraint solver from an efficient reduction of the search space by constraint propagation.

$$\frac{\gamma \nvDash \sigma(\alpha) \neq v,\ \ \langle s, \rho, \sigma[\alpha/v], \gamma \wedge (\sigma(\alpha) == v) \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \quad (\mathrm{Label})$$

16.4 Example Evaluation

We demonstrate the use of the reduction rules defined in Section 16.3 by computing one possible result of the logarithm program from Listing 16.1 that will be invoked by an additional method `int main() { return log(1); }`. Other possible results can be computed analogously. We abbreviate the code of `log` and `pow` by s_1 and s_3, respectively, to improve readability. The substatement s_2 is included in s_1, while s_3 includes the substatements s_4, s_5, and s_6. Moreover, we use the infix notation for nested expressions, e. g. we write $n \geq 1$ instead of $\geq (n, 1)$.

Initially, let $\rho_0 = \{main \mapsto (\epsilon, \texttt{return log(1);}),\ log \mapsto ((x), s_1),\ pow \mapsto ((b, y), s_3)\}$. Furthermore, let $\gamma_1 = true$ and $\sigma_0 = \{\alpha_0 \mapsto \bot\}$. We begin in method `main()`, which evaluates to

$$\dfrac{\dfrac{\langle 1, \rho_0, \sigma_0, \gamma_1\rangle \to (1, \sigma_0, \gamma_1)\ \text{(Con)},\ \rho_0(log) = ((x), s_1),\quad (\text{Lemma}_1),\ \sigma_6(\alpha_0) = 0}{\langle \texttt{log(1)}, \rho_0, \sigma_0, \gamma_1\rangle \to (0, \sigma_6[\alpha_0/\bot], \alpha_2 == 0)}\ \text{(Invoke)}}{\langle \texttt{return log(1)}, \rho_0, \sigma_0, \gamma_1\rangle \rightsquigarrow (\rho_0, \sigma_6[\alpha_0/0], \alpha_2 == 0)}\ \text{(Ret)}$$

Performing an entire evaluation with this example is interesting, but lengthy. We therefore moved the detailed evaluation into the appendix (cf. Lemma$_1$) and use the opportunity to highlight some interesting evaluation steps here. In the final state, $\sigma_6 = \sigma_0[\alpha_0/0, \alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1]$.

The final result $\sigma_6(\alpha_0) = 0$ results from the constraint $\alpha_2 \leq 0$ obtained from evaluating Wh_f (Lemma$_9$ in the appendix provides context):

$$\dfrac{\dfrac{\langle \texttt{i}, \rho_4, \sigma_3, \gamma_1\rangle \to (0, \sigma_3, \gamma_1)\ \text{(Var)},\quad \langle \texttt{y}, \rho_4, \sigma_3, \gamma_1\rangle \to (\alpha_2, \sigma_3, \gamma_1)\ \text{(Var)}}{\langle \texttt{i} < \texttt{y}, \rho_4, \sigma_3, \gamma_1\rangle \to (0 < \alpha_2, \sigma_3, \gamma_1)}\ \text{(AOp2)},\quad \gamma_1 \not\models (0 < \alpha_2)}{\langle \texttt{while (i < y) s}_6, \rho_4, \sigma_3, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1 \land \neg(0 < \alpha_2))}\ (\text{Wh}_f)$$

where $\rho_4 = \rho_0[b/\alpha_3, y/\alpha_4, i/\alpha_7, r/\alpha_8]$ and $\sigma_0[\alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1]$. $\alpha_2 \leq 0$ is further refined to $\alpha_2 == 0$ by the labeling rule in Lemma$_2$ in the appendix.

In Lemma$_2$, the constraint store is used to deduce that $\alpha_2 == 0$ is consistent with the current constraint, $\alpha_2 \leq 0$, as well as with the constraint store γ_2. Therefore, labeling non-deterministically imposes the more restrictive constraint $\alpha_2 == 0$. Other branches may impose further constraints consistent with $\alpha_2 < 0$.

If we had non-deterministically chosen rule Wh_t in Lemma$_9$, we would have performed an iteration of the `while` loop, leading to more computations that would not result in solutions, as they would be discarded as incorrect by the `fail` statement of the `log` method.

The evaluation of rule Assign in Lemma_7 creates a new memory location α_7 in σ_2 for the new value of `i` and updates the environment accordingly. At this point, no references to the old location α_5 exist, so an implementation could use garbage collection to free that location. Hypothetically, if rule Wh_t had been chosen in Lemma_9, an iteration of the loop would have resulted in additional evaluations of rule Assign, e.g. to increment `i`, thus reserving additional locations. In the case of `i`, the new value would depend on the value in α_7. However, as the old value and the increment are constant, the new value would be computed by evaluating rule AOp1, so that, again, no reference to α_7 is needed.

16.5 Discussion

The key aspect of the semantic rules for the presented core language is the interaction between constraint-logic programming and imperative programming. Some aspects of it offer themselves for thorough discussion.

The (potentially) non-deterministic evaluation of our rules If_f, If_t, Wh_f, and Wh_t highly depends on the included constraint solver. Our definition allows to follow a branch if the negation of its condition is not entailed by the current constraint store γ. When implementing this, a constraint solver will be used to check whether $\gamma \not\models \neg v$ (analogously for $\gamma \not\models v$). If the constraint solver is not able to show that the constraints entail $\neg v$, this may have three reasons: 1) $\gamma \models v$, or 2) the current constraints neither entail v nor $\neg v$, or 3) the constraint propagation abilities of the employed constraint solver are insufficient to show that $\gamma \models \neg v$, but in fact $\gamma \models \neg v$. In case 1), the system behaves deterministically and only one rule for `if` (or `while`) will be applied. In case 2), one of the two rules for `if` (or `while`) can be chosen non-deterministically. Only case 3) is problematic. In this case, a branch can be chosen that corresponds to inconsistent constraints. In practice, solvers do not achieve perfect constraint propagation and also no global consistency of the constraints. Consequently, results corresponding to inconsistent constraints may only be discovered later, e. g. during labeling. In the meantime, non-backtrackable statements (e. g. ones that result in input / output) of search regions may have been executed in branches that prove infeasible later. Thus, we suggest to avoid input / output in search regions.

We would like to point out that the aforementioned problem is not specific to Muli, as this can occur in Prolog (using CLP(FD) [Tri12]) as well. Consider the Prolog program provided in Listing 16.2. When you execute the first goal, the output will (among the unreduced constraint system) contain a line that says `successful`, even though it is apparent to the human reader that there is no solution, so that the `write` statement should

```
1 use_module(library(clpfd)).
2 ?- [X,Y,Z] ins 0..1, all_different([X,Y,Z]),
3      write('successful').
4 ?- [X,Y,Z] ins 0..1, all_different([X,Y,Z]),
5      label([X,Y,Z]), write('successful').
```

Listing 16.2: Demonstration of the limits of constraint propagation using an example in Prolog+CLP(FD).

not have been reached. In contrast, if `label` is invoked before `write` (second goal), Prolog realises that there is no solution and therefore gives the correct result `false`.

We see two options to handle this situation in Muli programs. The first option is to explicitly label variables sufficiently at every branch such that the constraint solver is able to either infer $\gamma \vDash v$ or $\gamma \vDash \neg v$. However, as explained in context of the Label rule, this also introduces a lot of non-deterministic branching by creating one branch per label. Therefore, the effectivity of constraint propagation is reduced and the overall effort for search is increased. For the same reason we decided that Muli should not implicitly perform labeling at every branch either, as performance would deteriorate.

The second option is to perform labeling only after a solution has been found during encapsulated search. In fact, such a solution is merely a potential solution, under the condition that the corresponding constraints are also satisfiable. As a result, encapsulated search produces a stream of pairs, each of which comprises one potential solution and its corresponding set of constraints. Thus, at this point the enclosing application can iterate over this stream and perform (sufficient) labeling, until it is clear whether the constraints are actually satisfiable. This rules out infeasible solutions afterwards. The implementation of Muli provides an explicit `label` operation, which the application developer can use for this purpose. We decided not to do this implicitly in order to give the developer more flexibility. It is easy to wrap this functionality into a search operation which labels every found solution implicitly.

Both mentioned options are available to the developer. We recommend the second one, possibly in the wrapped version with implicit labeling. For search regions that involve only backtrackable statements, the result does not depend on the chosen option, but the second option is presumably more efficient as fewer branches have to be evaluated. For other search regions, only the first option can avoid unwanted side effects of illegally accessed branches. However, search then becomes less efficient. Therefore, in case that non-backtrackable side effects have to be avoided, we recommend that the

```
1 int x = 5; int y = x;
2 x = 3;
```

Listing 16.3: Minimal example demonstrating that variables may be mutated directly, in contrast to results of their uses: After evaluation, y is 5.

developer removes input / output operations from search regions and moves them behind encapsulation instead.

Formalising the operational semantics of Muli has also helped uncover some operations whose semantics are sufficiently clear in deterministic Java, but become ambiguous when non-determinism and symbolic execution are added. Consequently, some alternatives could be discussed on a conceptual level using this semantics, before deriving a corresponding implementation. This particularly involves the interpretation of symbolic variables (rules Invoke and Var) and assignments, as outlined subsequently.

By rule Assign, an assignment x = e creates a new memory address for the variable x and changes the environment accordingly. As a result, memory usage of a Muli program is increased with every assignment, instead of with every declaration of a variable as in imperative OO languages. Nevertheless, this behaviour is required in order to avoid unwanted side effects on previous constraints involving x. The alternative, mutating $\sigma(\rho(x))$ directly, would result in assignments to x that could render constraints involving x unsatisfiable ex post, i. e. after branching has occurred that depended on such a constraint.

As another consequence, rule Assign ensures that the interpretation of symbolic variables is equivalent to that of regular values. Consider the simple excerpt from a Java program given in Listing 16.3 as an example: After evaluating the last line, y is still expected to be 5, even though x now holds a different value. After all, although primitive variables can be directly mutated in Java, their previous interpretations cannot. Similarly, for symbolic values, rule Assign ensures that references before and after an assignment are treated distinctly, even though memory efficiency is adversely affected. Nevertheless, unreferenced former meanings of a variable may be destroyed by the garbage collector, thus reclaiming (some) memory.

Implicitly, our rules Assign (or Invoke) and Var enable sharing of symbolic values. Assigning a free variable x to another free variable y means that the address $\rho(x)$ of x is stored in the memory location corresponding to y by modifying state as $\sigma[\rho(y)/\rho(x)]$. Consequently, subsequent constraints and expressions that involve either variable will actually reference the same variable. The sharing behaviour is exhibited in the example in Lemma_5 in the appendix, where a free variable is passed to the pow method as its second

parameter. `pow` adds constraints to that variable that only come into effect when labeling is performed in its invoking context in `log` (Lemma$_2$ in the appendix).

Regarding backtracking, the implementation is only implicitly affected by the presented operational semantics. Here, the semantics defines the desired state of the overall VM that must be achieved before evaluation in terms of $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$. Considering the multitude of options for achieving the desired VM state that lend themselves for the implementation, we briefly outline the options without prescribing either. Firstly, "don't care" non-determinism considers only one evaluation alternative and therefore does not require backtracking at all. Secondly, it would be possible to fork at statements that introduce non-determinism, thus evaluating all alternatives in parallel. This does not require backtracking either, however, consider that this generates a lot of overhead in terms of memory and computation, as the VM must be forked in its entirety to accommodate for any side effects, and as all forks must be joined in order to return to deterministic computation after a search region is fully processed. Thirdly, the alternatives can be evaluated sequentially. To achieve this, the VM must record changes to the data structures on a trail equivalent to that of Prolog in order to reconstruct a previous state during backtracking. Our implementation resorts to the latter option using a trail adapted from the Warren Abstract Machine. Nevertheless, the remaining options would also be interesting to pursue.

16.6 Related Work

To the best of our knowledge, this paper is the first to present a formal semantics of an imperative language enhanced by features of constraint-logic programming. For sake of clarity we focused on a core language. A full formal semantics of Java alone may require an entire book as in the work by Stärk et al. [SSB01]. K-Java [BR15] is another approach to define a formal semantics of Java. However, in the cited paper the authors focus on selected aspects of the language. The official semantics of Java is extensively described in natural language (cf. [Lin+15; Gos+15]).

Some existing core languages of Java such as Featherweight Java [IPW01] are tailored to the investigation of the typing system and not meant to be executable. Hainry [HP15] investigates an object-oriented core language focussing on computational complexity. As a result of their respective foci they were not suitable to be extended for Muli.

The encapsulated search of Muli has been inspired and adapted from the corresponding feature of the functional-logic language Curry. An operational semantics of Curry can

be found in [Alb+02]. It is simpler than our semantics, since Curry is purely declarative and does not have to bother with side effects.

Approaches for integrating object-oriented features into a (constrained) logic language are e. g. Oz [Van+03], Visual Prolog [Sco10], Prolog++ [Mos94], and Concurrent Prolog [ST83]. However, these approaches maintain a declarative flavour and mainly provide syntactic sugar for object-orientation. They are unfamiliar for mainstream object-oriented programmers.

There are also approaches which add constrained-logic features to an imperative / object-oriented language. Typically, the integration is less seamless than in Muli and the language parts stemming from different paradigms can clearly be distinguished [CV08; CV07]. CAPJa combines Java and Prolog and provides a simplified interface mapping Java objects to Prolog terms, but requires distinct code in each language nevertheless [Ost15]. LogicJava [MK11] is more restrictive than Muli and only allows class fields to be logic variables. Moreover, entire methods have to be declared as searching or non-searching.

16.7 Conclusions and Future Work

Our work formalises an operational reduction semantics for an imperative core of the novel integrated constraint-logic object-oriented language Muli. Muli extends Java by logic variables, non-determinism, encapsulated search, and constraint solving. Muli is particularly suited for enterprise applications that involve both searching and non-searching business logic. Encapsulated search ensures that non-determinism is only introduced deliberately where needed, instead of spreading out over the whole program. Thus, the code outside of encapsulated search regions behaves just as ordinary Java code.

The presented operational semantics provides a basis for implementations of compiler, symbolic JVM, and tools for processing Muli programs. In particular, the formalisation has helped clarify possible ambiguities w. r. t. the semantics of certain statements under non-determinism, such as that of assignments to variables and uses of them. Furthermore, the semantics will facilitate reasoning about programs developed in Muli as demonstrated in the example evaluation. We made the symbolic JVM that executes Muli programs available as free software on GitHub.[40]

As future work, we would like to extend our core language and its semantics by more features of Java, such as classes and inheritance. We expect these additions to be quite orthogonal to the presently supported concepts. However, when (non-deterministically)

[40] `https://github.com/wwu-pi/muli-env`.

instantiating a free variable with an object type, we have to take the whole corresponding inheritance hierarchy into account.

Appendix: Full Example Evaluation

In addition to ρ_0, σ_0, and γ_1 defined in section Section 16.4, the following auxiliary definitions will be needed as intermediate results: $\rho_1 = \rho_0[x/\alpha_1, y/\alpha_2]$, $\rho_2 = \rho_0[b/\alpha_3, y/\alpha_4]$, $\rho_3 = \rho_2[i/\alpha_5, r/\alpha_6]$, $\rho_4 = \rho_3[i/\alpha_7, r/\alpha_8]$, $\sigma_1 = \sigma_0[\alpha_1/1, \alpha_2/\alpha_2]$, $\sigma_2 = \sigma_1[\alpha_3/2, \alpha_4/\alpha_2]$, $\sigma_3 = \sigma_2[\alpha_7/0, \alpha_8/1]$, $\sigma_4 = \sigma_3[\alpha_0/1]$, $\sigma_5 = \sigma_4[\alpha_0/\bot]$, $\sigma_6 = \sigma_5[\alpha_0/0]$, $\gamma_2 = \gamma_1 \wedge \alpha_2 \leq 0$, and $\gamma_3 = \gamma_2 \wedge \alpha_2 == 0$. To simplify the understanding of the full computation provided in Section 16.4, we have decomposed it into a couple of lemmas. We present the computation in a top-down fashion. If you prefer a bottom-up fashion, just read the lemmas in reverse order. The names of the applied rules are specified in each step.

$$\frac{\langle \texttt{int y free;}\rangle \rightsquigarrow (\rho_0[x/\alpha_1, y/\alpha_2], \sigma_0[\alpha_1/1, \alpha_2/\alpha_2], \gamma_1) \text{ (Free)}, \quad (\text{Lemma}_2)}{\langle \texttt{int y free; s}_2, \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)} \text{ (Seq)} \qquad (\text{Lemma}_1)$$

$$\frac{(\text{Lemma}_3),\ \gamma_2 \not\models \neg\texttt{true},\ \dfrac{\gamma_2 \not\models \sigma_5(\alpha_2) \neq 0,\ (\text{Lemma}_4)}{\langle \texttt{return y;}, \rho_1, \sigma_5, \gamma_2\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)} \text{ (Label)}}{\langle s_2, \rho_1, \sigma_1, \gamma_1\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)} \text{ (If}_t\text{)} \qquad (\text{Lemma}_2)$$

$$\frac{(\text{Lemma}_5),\ \langle x, \rho_1, \sigma_5, \gamma_2\rangle \to (1, \sigma_5, \gamma_2) \text{ (Var)},\ 1 == 1 = \texttt{true}}{\langle \texttt{pow(2, y) == x}, \rho_1, \sigma_1, \gamma_1\rangle \to (\texttt{true}, \sigma_5, \gamma_2)} \text{ (AOp1)} \qquad (\text{Lemma}_3)$$

$$\frac{\langle y, \rho_1, \sigma_5[\alpha_2/0], \gamma_2 \wedge \alpha_2 == 0\rangle \to (0, \sigma_5, \gamma_3) \text{ (Var)}}{\langle \texttt{return y;}, \rho_1, \sigma_5, \gamma_2\rangle \rightsquigarrow (\rho_1, \sigma_5[\alpha_0/0], \gamma_3)} \text{ (Ret)} \qquad (\text{Lemma}_4)$$

$$\frac{\begin{array}{c}\langle 2, \rho_1, \sigma_1, \gamma_1\rangle \to (2, \sigma_1, \gamma_1) \text{ (Con)}, \\ \langle y, \rho_1, \sigma_1, \gamma_1\rangle \to (\alpha_2, \sigma_1, \gamma_1) \text{ (Var)}, \\ \rho_1(pow) = ((b, y), s_3),\ (\text{Lemma}_6),\ \sigma_4(\alpha_0) = 1\end{array}}{\langle \texttt{pow(2, y)}, \rho_1, \sigma_1, \gamma_1\rangle \to (1, \sigma_4[\alpha_0/\bot], \gamma_2)} \text{ (Invoke)} \qquad (\text{Lemma}_5)$$

$$\frac{\begin{array}{c}\langle \texttt{int i;}, \rho_2, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_2[i/\alpha_5], \sigma_2, \gamma_1) \text{ (Decl)}, \\ \dfrac{\langle \texttt{int r;}, \rho_2[i/\alpha_5], \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_2[i/\alpha_5, r/\alpha_6], \sigma_2, \gamma_1) \text{ (Decl)}, \quad (\text{Lemma}_7)}{\langle \texttt{int r; i = 0; r = 1; s}_4, \rho_2[i/\alpha_5], \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)} \text{ (Seq)}\end{array}}{\langle \texttt{int i; int r; i = 0; r = 1; s}_4, \rho_2, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)} \text{ (Seq)} \qquad (\text{Lemma}_6)$$

$$\dfrac{\dfrac{\dfrac{\langle 0, \rho_3, \sigma_2, \gamma_1 \rangle \to (0, \sigma_2, \gamma_1)\ \text{(Con)}}{\langle \mathtt{i = 0;}, \rho_3, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1)}\ \text{(Assign)}, \quad \dfrac{\dfrac{\langle 1, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \to (0, \sigma_2[\alpha_7/0], \gamma_1)\ \text{(Con)}}{\langle \mathtt{r = 1;}, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1)}\ \text{(Assign)}, \quad \text{(Lemma}_8\text{)}}{\langle \mathtt{r = 1; s_4}, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}}{\langle \mathtt{i = 0;\ r = 1;\ s_4}, \rho_3, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}}{}\quad \text{(Lemma}_7\text{)}$$

$$\dfrac{\text{(Lemma}_9\text{)}, \quad \dfrac{\langle r, \rho_4, \sigma_3, \gamma_2 \rangle \to (1, \sigma_3, \gamma_2)\ \text{(Var)}}{\langle \mathtt{return\ r;}, \rho_4, \sigma_3, \gamma_2 \rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2)}\ \text{(Ret)}}{\langle \mathtt{s_5;\ return\ r;}, \rho_4, \sigma_3, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2)}\ \text{(Seq)} \qquad \text{(Lemma}_8\text{)}$$

$$\dfrac{\dfrac{\langle \mathtt{i}, \rho_4, \sigma_3, \gamma_1 \rangle \to (0, \sigma_3, \gamma_1)\ \text{(Var)}, \quad \langle \mathtt{y}, \rho_4, \sigma_3, \gamma_1 \rangle \to (\alpha_2, \sigma_3, \gamma_1)\ \text{(Var)}}{\langle \mathtt{i < y}, \rho_4, \sigma_3, \gamma_1 \rangle \to (0 < \alpha_2, \sigma_3, \gamma_1)}\ \text{(AOp2)}, \quad \gamma \not\models (0 < \alpha_2)}{\langle \mathtt{while\ (i < y)\ s_6}, \rho_4, \sigma_3, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1 \land \neg(0 < \alpha_2))}\ (\text{Wh}_f) \qquad \text{(Lemma}_9\text{)}$$

References

[Alb+02] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver and Germán Vidal. 'An operational semantics for declarative multi-paradigm languages'. In: *Electronic Notes in Theoretical Computer Science* 70.6 (2002), pp. 65–86. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)80600-5.

[BR15] Denis Bogdanas and Grigore Rosu. 'K-Java: A Complete Semantics of Java'. In: *POPL '15* (2015), pp. 1–12. ISSN: 15232867. DOI: 10.1145/2676726.2676982.

[CV07] Maurizio Cimadamore and Mirko Viroli. 'A Prolog-oriented Extension of Java Programming based on Generics and Annotations'. In: *Proceedings PPPJ*. Ed. by Vasco Amaral and et al. Vol. 272. ACM ICPS. ACM, 2007, pp. 197–202. ISBN: 978-1-59593-672-1. DOI: 10.1145/1294325.1294352.

[CV08] Maurizio Cimadamore and Mirko Viroli. 'Integrating Java and Prolog through generic methods and type inference'. In: *Proceedings of the 2008 SAC*. Ed. by Roger L. Wainwright and Hisham Haddad. ACM, 2008, pp. 198–205. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1363740. URL: http://doi.acm.org/10.1145/1363686.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Berlin Heidelberg: Springer, 2003. ISBN: 978-3-642-08712-7.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 09/06/2017).

[Hoo06] John N. Hooker. 'Operations Research Methods in Constraint Programming'. In: *Handbook of CP*. Ed. by Francesca Rossi, Peter van Beek and Toby Walsh. Elsevier, 2006. Chap. 15. DOI: 10.1016/S1574-6526(06)80019-2.

[HP15] Emmanuel Hainry and Romain Péchoux. 'Objects in Polynomial Time'. In: *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015*. 2015, pp. 387–404. ISBN: 9783319265285. DOI: 10.1007/978-3-319-26529-2_21.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce and Philip Wadler. 'Featherweight Java: A Minimal Core Calculus for Java and GJ'. In: *ACM Trans. Program. Lang. Syst.z* 23.3 (2001), pp. 396–450. DOI: 10.1145/503502.503505.

[KO08] Goh Kondoh and Tamiya Onodera. 'Finding bugs in Java Native Interface programs'. In: *ISSTA '08* (2008), p. 109. DOI: 10.1145/1390630.1390645.

[Kuc03] Krzysztof Kuchcinski. 'Constraints-driven scheduling and resource assignment'. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 10844309. DOI: 10.1145/785411.785416.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 09/06/2017).

[Lou93] Kenneth C Louden. *Programming Languages: Principles and Practice*. Ed. by Patricia Adams. Belmont, CA, USA: Wadsworth Publ. Co., 1993. ISBN: 0534932770.

[MK11] Tim A Majchrzak and Herbert Kuchen. 'Logic Java: Combining Object-Oriented and Logic Programming'. In: *WFLP 2011*. 2011, pp. 122–137. ISBN: 978-3-642-22530-7.

[Mos94] Chris Moss. *Prolog++ - the power of object-oriented and logic programming*. International series in logic programming. Addison-Wesley, 1994. ISBN: 978-0-201-56507-2.

[Ost15] Ludwig Ostermayer. 'Seamless Cooperation of Java and Prolog for Rule-Based Software Development'. In: *Proceedings of the RuleML 2015, Berlin.* 2015.

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog.* Outskirts Press, 2010. ISBN: 9781432749361.

[SSB01] Robert Stärk, Joachim Schmid and Egon Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation.* Springer-Verlag, 2001. ISBN: 978-3-642-63997-5. DOI: 10.1007/978-3-642-59495-3.

[ST83] Ehud Shapiro and Akikazu Takeuchi. 'Object oriented programming in Concurrent Prolog'. In: *New Generation Computing* 1.1 (1983), pp. 25–48. ISSN: 02883635. DOI: 10.1007/BF03037020.

[Tri12] Markus Triska. 'The Finite Domain Constraint Solver of SWI-Prolog'. In: *FLOPS.* Vol. 7294. LNCS. 2012, pp. 307–316.

[Van+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte and Martin Henz. 'Logic programming in the context of multiparadigm programming: the Oz experience'. In: *Theory and Practice of Logic Programming* 3.6 (2003), pp. 717–763. ISSN: 14710684. DOI: 10.1017/S1471068403001741.

[War83] David H. D. Warren. *An Abstract Prolog Instruction Set.* Tech. rep. Menlo Park: SRI International, 1983.

Curriculum Vitae

The curriculum vitae that appeared in the printed version of this thesis is omitted from the digital version for data protection reasons.

List of Publications

Since the year 2014, the following scientific publications that I authored and co-authored have been published or are currently undergoing formal review procedures. Some publications tackled issues that did not match the focus of this thesis, so they were neither reproduced in Part II nor incorporated into the research overview in Part I.

1. Jan C. Dageförde and Herbert Kuchen. 'Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming'. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems.* Ed. by Pedro Lopez-Garcia, Roberto Giacobazzi and John Gallagher. LNCS. Springer, 2020. Under review

2. Jan C. Dageförde and Herbert Kuchen. 'Free Objects in Constraint-logic Object-oriented Programming'. In: *Proceedings of the ACM on Programming Languages (OOPSLA).* 2020. Under review

3. Jan C. Dageförde and Finn Teegen. 'Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming'. In: *Declarative Programming and Knowledge Management.* Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen and Dietmar Seipel. Vol. 12057. Lecture Notes in Artificial Intelligence. 2020, pp. 199–214. DOI: 10.1007/978-3-030-46714-2_13

4. Jan C. Dageförde. 'Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming'. In: *Functional and Constraint Logic Programming.* Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8

5. Jan C. Dageförde, Sandra Dylus, Jan Christiansen, Finn Teegen and Jan Rasmus Tikovsky. 'Strukturierte Traversierung des Ausführungsbaums von Muli-Programmen'. In: *36th Annual Meeting of the GI Working Group "Programming Languages and Computing Concepts".* Ed. by J. Knoop, M. Steffen and Trancón y Widemann, B. 2019, pp. 1–6. URL: https://www.duo.uio.no/handle/10852/72477

6. Lars Beyer, Jan C. Dageförde, Herbert Kuchen and Claus A. Usener. 'Automated Data-Flow Analysis and Validation in Process Automation Projects'. In: *Advancing Technology Industrialization Through Intelligent Software Methodologies, Tools and Techniques.* 2019, pp. 333–346. DOI: 10.3233/FAIA190061

7. Jan C. Dageförde and Herbert Kuchen. 'A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli'. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001

8. Jan C. Dageförde and Herbert Kuchen. 'Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space'. In: *Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing*. Limassol, Cyprus, 2019, pp. 1552–1561. DOI: 10.1145/3297280.3298912

9. Jan C. Dageförde and Herbert Kuchen. 'Muli: Constraint-Programmierung in Java auf symbolischer JVM'. in: *35th Annual Meeting of the GI Working Group "Programming Languages and Computing Concepts"*. Ed. by J. Knoop, M. Steffen and B. Trancón y Widemann. 2018, pp. 23–29. URL: http://urn.nb.no/URN:NBN:no-65294

10. Jan C. Dageförde and Herbert Kuchen. 'An Operational Semantics for Constraint-Logic Imperative Programming'. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5

11. Jan C. Dageförde and Herbert Kuchen. 'A Constraint-logic Object-oriented Language'. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*. ACM, 2018, pp. 1185–1194. DOI: 10.1145/3167132.3167260

12. Tim A. Majchrzak, Jan C. Dageförde, Jan Ernsting, Christoph Rieger and Tobias Reischmann. 'How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps'. In: *Apps Management and E-Commerce Transactions in Real-Time*. Ed. by Sajad Rezaei. Hershey PA: IGI Global, 2017. Chap. 5, pp. 104–140. DOI: 10.4018/978-1-5225-2449-6.ch005

13. Florian Runschke, Jan C. Dageförde, Hendrik Scholta and Sebastian Bräuer. 'Management von Informationsobjekten in hybriden Wertschöpfungsnetzwerken'. In: *Planung koordinierter Wertschöpfungspartnerschaften*. Ed. by Jörg Becker, Torben Bernhold, Ralf Knackstedt and Martin Matzner. Berlin, Heidelberg: Springer Gabler, 2017, pp. 179–202. ISBN: 978-3-662-55361-9

14. Jan C. Dageförde, Tobias Reischmann, Tim A. Majchrzak and Jan Ernsting. 'Generating app product lines in a model-driven cross-platform development approach'. In: *Proceedings of the Annual Hawaii International Conference on System Sciences.* 2016. ISBN: 9780769556703. DOI: 10.1109/HICSS.2016.718

15. Jörg Becker, Kevin Ortbach, Sebastian Köffer, Jan C. Dageförde and Björn Niehaves. 'Old dogs and new tricks – Exploring the benefits and drawbacks of IT consumerization in the context of aging workforces'. In: *Tagungsband Multikonferenz Wirtschaftsinformatik, MKWI 2014.* 2014. ISBN: 9783000453113