



University of Novi Sad
Faculty of Sciences
Department of Mathematics and Informatics



UNIVERSITY OF NOVI SAD

DOCTORAL DISSERTATION

Metadata-Supported Object-Oriented Extension of Dynamic Geometry Software

DOKTORSKA DISERTACIJA

**Objektno-orijentisano proširenje softvera za
dinamičku geometriju podržano metapodacima**

Author:
Davorika RADAKOVIĆ, M.Sc.

Supervisor:
Dr. Miloš RADOVANOVIĆ

Novi Sad, June 2019

“No mathematical subject is more relevant than geometry. It lies at the heart of physics, chemistry, biology, geology and geography, art and architecture. It also lies at the heart of mathematics, though much of the 20th century the centrality of geometry was obscured by fashionable abstraction. This is changing now, thanks to computation and computer graphics which make it possible to reclaim this core without loss of rigor. (Marhorie Senechal, personal communication)”

Julie Sarama, Douglas H. Clements (2009).
*Early Childhood Mathematics Education Research:
Learning Trajectories for Young Children.*
Studies in Mathematical Thinking and
Learning Series. Taylor & Francis.

Contents

Contents	viii
Abstract	ix
Apstrakt	xi
Preface	xiii
Acknowledgements	xv
List of Figures	xviii
List of Tables	xix
List of Listings	xxi
List of Abbreviations	xxiii
I Preliminaries	1
1 Introduction	3
1.1 Metadata	4
1.2 Functional Domain-Specific Languages	5
1.3 DGS Extensibility	7
1.3.1 Introducing OO Features Into DGS	8
1.4 Main Goals	9
1.5 Contributions	10
1.6 Summary	10
2 Related Work	11
2.1 Dynamic Geometry Software	11
2.2 Lazy Evaluation	15
2.3 Metadata	17
2.4 Components	19
2.5 Discussion	21
2.6 Summary	22

II	Implementation	23
3	Motivation	25
3.1	Attributes in C#	25
3.2	Why Not Use Attributes?	27
3.3	Property Functions vs. Objects with Properties	28
3.4	Dot Notation for Property Access	30
3.4.1	Introducing Semantic Extensions into FLG	30
3.4.2	Language Composition Considerations in FLG	32
3.4.3	Evaluation Optimization	35
3.5	Summary	36
4	System Architecture Overview	37
4.1	SLGeometry Framework	37
4.2	FLG Grammar Definition	39
4.3	Expressions	42
4.3.1	Functions and Visual Functions	43
4.3.2	Constants	43
4.3.3	Object Constants	44
4.4	Summary	44
5	Implementation Details	45
5.1	Metadata	46
5.1.1	Metadata for Arguments	47
5.1.2	Metadata for Signatures	47
5.1.3	Metadata for Properties	48
5.1.4	Metadata for Functions and Controls	50
5.1.4.1	NameInfo	50
5.1.4.2	FnInfo	51
5.1.4.3	ConstInfo and ConstObjectInfo	52
5.1.5	Metadata for User Interactive Controls	53
5.1.6	Implementation Examples	54
5.1.6.1	A Simple Function	55
5.1.6.2	Visual Functions and Objects	55
5.2	Type Conversions	57
5.3	Operations	57
5.3.1	Result Caching	58
5.3.2	Operation Examples	59
5.3.3	Late vs. Early Operation Binding	59
5.3.4	Generalized Binary Operation Algorithm	60
5.4	Lazy Evaluation with Property Activation	61
5.4.1	Implementation Requirements	61
5.4.2	Structural Correspondence	62
5.4.3	Backing Fields and Accessors	63
5.4.4	Property Evaluators	64
5.4.5	Instance Initialization and Updating	64
5.5	Partial Compilation of Expression Trees	65

5.5.1	Partial Compilation Implementation	66
5.6	XML-Based Serialization of Geometrical Constructions	69
5.7	Summary	70
III Validation		71
6	Subject-Specific Components in DGS	73
6.1	Motivation	74
6.2	Components in DGS	75
6.3	The Country and City Components	76
6.3.1	The Experiments	76
6.3.1.1	Task 1 – Measuring a River’s Length	77
6.3.1.2	Task 2 – Tracing the Great Explorers’ Voyages	78
6.3.1.3	Task 3 – Tracing One’s Own Travels	78
6.3.2	Results and Comments	78
6.4	Sequential Behavior Controllers in SLGeometry	79
6.4.1	Example 1 – Arithmetic Sum	80
6.4.2	Example 2 – A Simple Game	81
6.4.3	Experiment	83
6.4.3.1	Assignments	83
6.4.4	Results and Comments	83
6.5	Summary	85
7	Testing Lazy Evaluation	87
7.1	Why the Triangle?	88
7.2	Constructions	88
7.3	Experimental Setup	89
7.4	Results, Analysis and Discussion	91
7.4.1	CPU Time Tests for 5 Triangles	92
7.4.2	CPU Time Tests for 100 Evaluations	94
7.4.3	Object Count and Memory Tests	96
7.4.3.1	Measuring for 9 Properties of Triangle	96
7.4.3.2	Measuring for 27 Properties of Triangle	98
7.5	Summary	98
8	Conclusion and Future Work	101
8.1	Conclusion	101
8.2	Future Work	103
Bibliography		105
Prošireni izvod (Extended Abstract in Serbian)		123
Biography		139
Biografija (in Serbian)		141
Key Words Documentation		143

Abstract

Metadata-Supported Object-Oriented Extension of Dynamic Geometry Software

by Davorka RADAKOVIĆ, M.Sc.

Nowadays, Dynamic Geometry Software (DGS) is widely accepted as a tool for creating and presenting visually rich interactive teaching and learning materials, called dynamic drawings. Dynamic drawings are specified by writing expressions in functional domain-specific languages. Due to wide acceptance of DGS, there has arisen a need for their extensibility, by adding new semantics and visual objects (*visuals*). We have developed a programming framework for the Dynamic Geometry Software, SLGeometry, with a genericized functional language and corresponding expression evaluator that act as a framework into which specific semantics is embedded in the form of code annotated with metadata. The framework transforms an ordinary expression tree evaluator into an object-oriented one, and provide guidelines and examples for creation of interactive objects with dynamic properties, which participate in evaluation optimization at run-time. Whereas other DGS are based on purely functional expression evaluators, our solution has advantages of being more general, easy to implement, and providing a natural way of specifying object properties in the user interface, minimizing typing and syntax errors.

SLGeometry is implemented in C# on the .NET Framework. Although attributes are a preferred mechanism to provide association of declarative information with C# code, they have certain restrictions which limit their application to representing complex structured metadata. By developing a metadata infrastructure which is independent of attributes, we were able to overcome these limitations. Our solution, presented in this dissertation, provides extensibility to simple and complex data types, unary and binary operations, type conversions, functions and visuals, thus enabling developers to seamlessly add new features to SLGeometry by implementing them as C# classes annotated with metadata. It also provides insight into the way a domain specific functional language of dynamic geometry software can be genericized and customized for specific needs by extending or restricting the set of types, operations, type conversions, functions and visuals.

Furthermore, we have conducted experiments with several groups of students of mathematics and high school pupils, in order to test how our approach compares to the existing practice. The experimental subjects tested mathematical games using interactive visual controls (UI controls) and sequential behavior controllers.

Finally, we present a new evaluation algorithm, which was compared to the usual approach employed in DGS and found to perform well, introducing advantages while maintaining the same level of performance.

Apstrakt

Objektno-orijentisano proširenje softvera za dinamičku geometriju podržano metapodacima

mr Davorka RADAKOVIĆ

U današnje vreme softver za dinamičku geometriju (DGS) je široko prihvaćen kao alat za kreiranje i prezentovanje vizuelno bogatih interaktivnih nastavnih materijala i materijala za samostalno učenje, nazvanih dinamičkim crtežima. Kako je raslo prihvatanje softvera za dinamičku geometriju, tako je i rasla potreba da se oni proširuju, dodajući im novu semantiku i vizualne objekte. Razvili smo programsko okruženje za softver za dinamičku geometriju, SLGeometry, sa generičkim funkcionalnim jezikom i odgovarajućim evaluatorom izraza koji čini okruženje u kom su ugrađene specifične semantike u obliku koda označenog metapodacima. Ovo okruženje pretvara uobičajen evaluator stabla izraza u objektno orijentiran, te daje uputstva i primere za stvaranje interaktivnih objekata sa dinamičkim osobinama, koji sudeluju u optimizaciji izvršenja tokom izvođenja. Dok se drugi DGS-ovi temelje na čisto funkcionalnim evaluatorima izraza, naše rješenje ima prednosti jer je uopštenije, lako za implementaciju i pruža prirodan način navođenja osobina objekta u korisničkom interfejsu, minimizirajući kucanje i sintaksne greške.

SLGeometry je implementirana u jeziku C# .NET Framework-a. Iako su atributi preferiran mehanizam, koji povezuje C# kôd sa deklarativnim informacijama, oni imaju određena ograničenja koja limitiraju njihovu primenu za predstavljanje složenih strukturiranih metapodataka. Razvijanjem infrastrukture metapodataka koja je nezavisna od atributa, uspeli smo prevladati ta ograničenja. Naše rešenje, predstavljeno u ovoj disertaciji, pruža proširivost: jednostavnim i složenim vrstama podataka, unarnim i binarnim operacijama, konverzijama tipova, funkcijama i vizuelnim objektima, omogućavajući time programerima da neprimetno dodaju nove osobine u SLGeometry implementirajući ih kao C# klase označene metapodacima. Takođe, okruženje pruža uvid na koji se način jezik specifičan za funkcionalni domen softvera za dinamičku geometriju može napraviti generičkim i prilagođen specifičnim potrebama: proširivanjem ili ograničavanjem skupa tipova, operacija, konverzija tipova, funkcija i vizuelnih elemenata.

Pored toga, sproveli smo nekoliko eksperimenata sa više grupa studenata matematike i srednjoškolaca, sa ciljem da testiramo kako se naš pristup može uporediti sa postojećom praksom. Tokom eksperimenata testirane su matematičke igre koristeći interaktivne vizualne kontrole (UI kontrole) i kontrolere sekvencijalnog ponašanja.

Na kraju je prikazan i novi algoritam za izračunavanje, koji se pokazao uspešnim u poređenju sa uobičajenim pristupom koji se koristi u DGS-ima, a opet uvodeći prednosti dok je zadržao istu razinu performansi.

Preface

Ever since the invention of the Jacquard loom¹, people have been trying to develop automata and programs to help them in everyday life. Today, computers are in everyday use, in all segments of our lives. One of the most important of these segments is education. Hence, one of the primary purposes of computers is their use in education. However, computers are not all that we need for teaching, proper software is also needed. This is the reason why software for dynamic geometry (DGS) is becoming increasingly more developed over the years. DGS allows the user to manipulate ('drag') the geometric object into different shapes or positions, in that manner being able to see what is happening with construction. However, DGS is not used only for teaching geometry, but also other subjects, such as physics, geography, etc.

In time, two shortcomings of the current DGS have surfaced and come to our attention: DGS are not well suited for universal application, as they mostly contain geometric objects and those objects carry only a few properties, in order to be light and efficiently evaluated. Having this in mind, our aim is to develop DGS as a foundation for experiments in improving the development of teaching materials and games.

This dissertation presents a novel approach to source code annotations which contain as metadata complex structural information, allowing the use of custom value types, in a flexible and a simple way to use, and also, accessible via reflection. We created a generalized extensible DGS SLGeometry in C# on the .NET Framework, in order to observe the design requirements of such a software and propose a viable implementation of the extensibility framework based on metadata.

The main components of developed framework are:

- Expression parser;
- Expression evaluator (Engine);
- Graphical surface (GeoCanvas).

The described approach contributes to the SLGeometry framework with crucial features:

- Extensibility with new types, functions and visual objects;
- The unification of objects with their properties and accessing them using dot notation;
- Enriched structural metadata specification for DGS with optimization and lazy evaluation mechanisms.

The work in this dissertation is organized in three parts: preliminaries, implementation, and validation. The parts are divided into chapters as follows.

Part I - Preliminaries provides the basic concepts and overview of the current state-of-the-art. Chapter 1 gives a brief introduction presenting the formulation of the problems that we intend to solve, the main goals in the development of the framework, as well as an explicit list of contributions of this work. In Chapter 2 we briefly overview the state-of-the-art concerning some well-known DGS, lazy evaluation, metadata and object-oriented extensions, and use of components.

¹<https://www.computerhope.com/jargon/j/jacquard-loom.htm>

Part II - Implementation presents several motivating examples which highlight the problems in implementation of attributes and use of DGS with purely functional languages and gives implementation details of the proposed solution, along with the system architecture overview. Chapter 3. gives an overview of attributes in .NET and shows its shortcomings; discusses advantages of objects with properties and dot notation; introduces semantic extensions and language composition consideration in a functional domain-specific language (FLG). In Chapter 4, the SLGeometry framework and its architecture are described, as well as FLG language descriptions and the infrastructure of expression classes. The more detailed implementation represents our main contribution concerning the metadata-supported object-oriented extensions of DGS and mechanisms for lazy evaluation of calculated properties. Also, type conversions, operations with result caching, partial compilation of expression trees and XML representation of drawings are given in Chapter 5.

Part III - Validation deals with the validation of given concepts and mechanisms which we introduced in last part. A pattern for building mathematical games in SLGeometry, along with the features of our components are presented in Chapter 6. Furthermore, we conducted several experiments with pupils and students, and tested our approaches in the class environment. In addition, the experimental verification which confirms that our proposed metadata infrastructure with lazy evaluation is comparable with the functional evaluation scheme is presented in Chapter 7. Experiments were conducted with three different expression tree evaluation strategies: eager, functional, and lazy evaluation. Finally, Chapter 8 concludes the dissertation and discusses current and future work.

Relevant papers published during the development of this thesis are following: Radaković and Herceg (2010), Radaković, Herceg, and Löberbauer (2010), Herceg and Radaković (2011), Herceg, Herceg-Mandić, and Radaković (2012), Herceg, Radaković, and Herceg (2012), Radaković and Herceg (2013), Steingartner et al. (2016), Radaković and Herceg (2017), Herceg et al. (2019), and Radaković and Herceg (2018).

Acknowledgements

I am grateful to all of those with whom I have had the pleasure to work with during this long, hard, challenging and amazing journey, with many ups and downs. This research would not have been possible without the support of my family, supervisor and professors, colleagues and friends. They all have earned my gratitude for their contribution and inspiration during my doctoral studies.

I would like to give special thanks to my dissertation committee. Each of you have given of your time, patience, energy, and expertise and I am richer for it.

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Miloš Radovanović, who encouraged me to pursue this degree and who has mentored me over the last years. I am indebted for his time, insightful comments, and careful attention to detail, which influenced this dissertation and made it much better.

Further, I owe a profound gratitude to Dr. Đorđe Herceg and his family, for many hours we had spent together working on this research and writing papers, and many insightful discussions and suggestions. Also, I have regarded him as my co-mentor. Similar, my debt of gratitude goes to Dr. Zoran Budimac and Dr. Milan Vidaković for the friendly support, their input, and a careful reading of this dissertation.

I owe my thanks to the great people, Dr. Valerie Novitzká and Dr. William Steingartner, who hosted me during several CEEPUS network grant researches at KPI, Technical University of Košice, Slovakia, and where I have done one part of tests.

My special thanks are also addressed to colleagues Dr. Gordana Rakić with sms-support, Dr. Doni Pracner and Dr. Diana Stoeva with Latex support. My thanks also go to my colleagues from the Chair of Computer Science and lab mates for listening, offering me advice, and supporting me through this entire process. Completing this work would have been all the more difficult were it not for the major source of support provided by the other members of the Department of Mathematics and Informatics.

To my friends scattered around the world, thank you for your thoughts, well-wishes, long talks and walks, e-mails, and being there whenever I needed a friend.

Thanks are also due to the ResearchGate and Mendeley Researcher Network where I have found many scientific discoveries.

Finally, nobody has been more important to me, in the pursuit of this research, than the members of my family. Thank you for encouraging me in all of my aspirations and inspiring me to follow my dreams. I am especially grateful to my parents, my aunt Marija Lavrnja, my brother and his family, grandma Nada and Slobodan, who supported me emotionally and endlessly. I always knew that you believed in me and wanted the best for me.

I dedicate this dissertation to my dearly missed mother, Stanka, who taught me that my job in life was to learn, to be happy and enjoy. She guided me as a person, supported, encouraged, and most importantly unconditionally loved. I am deeply indebted to her for all she gave me, how she raised me and how there were no limits to the things she did for me. Mum, I miss you very much. . .

List of Figures

3.1	Parameter value suggestions in the input box in GeoGebra	33
3.2	The Suggestions Bar in Mathematica	33
4.1	SLGeometry system architecture overview	38
4.2	Mapping of C# implementations to members of τ	40
4.3	Syntax diagrams for SymbFunc, Prop and Factor	41
4.4	Productions Expression, Term and Pow	41
4.5	Productions Or-, And- and Rel-Exp	42
4.6	Productions Expressions and LongIdent	42
4.7	Expression classes in SLGeometry	43
5.1	Constant and function metadata	46
5.2	Metadata for binary and unary operations	58
5.3	Chained activation for the CTriangle.MedianA property	63
5.4	Dependency tree for the CTriangle.Circumcircle property	63
5.5	Result types of various expressions	66
5.6	A subset of the compilation infrastructure	67
5.7	Compiled CIL code	68
5.8	Execution flow illustration of CompileNew method in Plus function	69
6.1	Straightforward path to the solution	74
6.2	Deviating from the original problem to construct auxiliary visuals	74
6.3	Mainland Italy, drawn by the Country component	76
6.4	Measuring the length of a river	77
6.5	Tracing the voyages of the great explorers Magellan and Cook	77
6.6	Tracing a voyage using the Country and City components	77
6.7	Propagation of triggers between components - Arithmetic Sum example	81
6.8	The “guess the midpoint” game in SLGeometry	81
7.1	Single triangle with all 27 properties (A) and an example for $n = 40$ in SLGeometry (B)	92
7.2	Comparative graphs of mean values of CPU time for ratio Lazy/Eager (A) and Lazy/Functional (B) on all configuration ($n = 5$)	93
7.3	Box Plot of results made on Configuration 1: (A) 5 triangles (B) 10 triangles	95
7.4	Comparative graphs of mean values of CPU time of ratio Lazy/Eager (A) and Lazy/Functional (B) on all configuration ($n = 5, 10, 20, 30, 40$)	95
7.5	Memory (A) and object (B) comparison for Configuration 1 (9 properties)	97
7.6	Average memory and object comparison Functional/Lazy for all 27 properties of the triangle	98

List of Tables

1.1	A dynamic drawing, specified by four expressions, explicit and implicit expression input via text and tools	6
3.1	A subset of calculated properties of a triangle	29
3.2	Dot notation transformed into the functional equivalent	30
3.3	Functional vs dot notation in determining the foot of the altitude A	34
5.1	Function result type metadata in the Fn class	60
5.2	Binary operations applied to atomic values and lists	60
6.1	Percentage of finished tasks	79
6.2	Sequential behavior controllers and their properties	80
6.3	Daisy chaining the components via triggers	81
6.4	Definition of the ‘guess the midpoint’ game	82
6.5	Students’ test scores	84
6.6	Poll results	84
7.1	Constructions used in tests, specified in OO syntax. For dependent triangles, $i = 1, \dots, n; k = i 5; j = i 5$	90
7.2	Constructions used in tests, specified in Functional syntax. For dependent triangles, $i = 1, \dots, n; k = i 5; j = i 5$	91
7.3	Computer configurations used for testing	91
7.4	Multiple Comparisons p values (2-tailed); Results Conf3M50, Independent (grouping) variable: Groups Conf3M50 Kruskal-Wallis test: $H(2, N = 30) = 25.82944$ $p < 0.05$ Computer configurations used for testing	94
7.5	Mean \pm standard deviation values of CPU time on all configurations ($n = 5$)	94
7.6	Mean \pm standard deviation values of CPU time on all configurations ($n = 5, 10, 20, 30, 40$)	96
7.7	Average difference between heap object count and memory consumption before and after the test examples are loaded for Lazy, Functional and Eager testing (Config. 1, 9 properties)	96

Listings

3.1	Accessing attributes with reflection	26
3.2	Attribute annotations for the FCircle function	27
3.3	The FunctionAttribute class in C#	27
3.4	The CSegment class annotated with property metadata	29
3.5	Table algorithm	31
3.6	Metadata specification of arguments for the Table function	31
5.1	ArgNamedInfo metadata	47
5.2	SignatureBase metadata	48
5.3	PropInfo metadata	50
5.4	NameInfo metadata	51
5.5	FnInfo metadata	51
5.6	ConstObjectInfo metadata	52
5.7	UIControlInfo metadata	53
5.8	VisualInfo metadata	53
5.9	Struct representing a line	54
5.10	Metadata for the Sqrt function	55
5.11	Metadata for the Plus function	55
5.12	Metadata for the CCircle object constant type	55
5.13	Metadata registration for the FCircle and FCircle3 functions	56
5.14	Metadata for the VCircle helper class	56
5.15	Conversion metadata for the Number data type	57
5.16	Explicit conversion methods for the Number data type	57
5.17	Operations for the Number data type	59
5.18	The default late bound binary operation	60
5.19	The generalized late bound binary operation	61
5.20	Metadata for the CTriangle object data type	62
5.21	Backing fields and accessors for CTriangle	64
5.22	Property evaluators for the CTriangle object data type	64
5.23	Initialization and updating of the CTriangle object instance	64
5.24	Implementation of CanCompile method for different superclasses of GExpression	67
5.25	Implementation of CompileNew method for the Plus function	68
5.26	Implementation of CompileNew method for the Sqrt function	68
5.27	XML representation of the drawing with one point	69
6.1	Country component - Italy	76
6.2	Country and City components	78

List of Abbreviations

CAI	Computer-Assisted Instruction program
CAS	Computer Algebra Systems
CIL	Common Intermediate Language
CLR	Common Language Runtime
COM	Component Object Model
CSCL	Computer Supported Collaborative Learning
CTS	Common Type System
DIMLE	Dynamic and Interactive Mathematics Learning Environments
DGS	Dynamic Geometry Software
DLL	Dynamic-Link Library
DSL	DomainSpecific Language
FLG	Functional Language for Geometry
GC	Garbage Collector
GCLC	Geometry Constructions - LaTeX Converter
HL	Host Language
ICT	Information and Communications Technologies
KRC	Kent Recursive Calculator
PSE	Problem-Solving Environment
STEM	Science, Technology, Engineering and Mathematics

*Dedicated to my brave and everlasting beloved Mum,
an amazing, remarkable, obliging, high-minded lady...*

*Cancer takes too many lives!
It strikes you down when you do not expect...*

Part I

Preliminaries

Chapter 1

Introduction

Dynamic geometry software (DGS) is widely accepted by teachers around the world as a tool for creating, demonstrating and disseminating interactive teaching materials in the form of dynamic drawings at all levels of education, from elementary school to university. It has made a significant impact on the way geometry is taught in schools (Abramovich, 2013). The main reason for such enthusiastic reception, in the era of modern learning (Tankelevičienė, 2004; Targamadžė and Petrauskienė, 2010), is that DGS are intuitive and easy to use. Several good DGS have stood the test of time: *GeoGebra* (2019), *The Interactive Geometry Software Cinderella* (2019), *Cabri* (2019), and *The Geometer's Sketchpad* (2019).

Creation of geometric drawings, that was once a tedious process, has become easy and available to anyone with even a modest personal computer. In time, teachers started using DGS to create teaching and learning materials for subjects other than geometry. Many examples in the subjects of geography (Herceg, Herceg-Mandić, and Radaković, 2012), numerical analysis (Herceg and Herceg, 2009; Mulansky and Ahnert, 2011), combinatorics, architecture, mechanical engineering etc. can be found, see for example *GeoGebraTube* (*GeoGebra Materials*, 2019).

A typical DGS consists of an evaluation engine and a front end. The evaluation engine maintains expressions that make up a drawing and evaluates them. The front end receives user input and displays visual objects (*visuals*). Interactivity in DGS stems from a simple principle: as one part of dynamic drawing changes, all parts that depend on it are recalculated automatically.

Dynamic drawings are defined by expressions that are written in functional domain-specific languages. Consequently, there was a need for their expandability and application in other fields, in addition to the primarily conceived geometry.

Although the use of attributes is a preferred mechanism that allows the association of declarative information with C# code, they have certain restrictions which limit their use in the representation of complex structured metadata. This research primarily wants to overcome this problem and seeks to develop a metadata infrastructure that is independent of attributes and thus overcomes their shortcomings, focusing on the C# language and the .NET platform.

The new concepts in this research will provide extensibility to simple and complex data types, unary and binary operations, type conversions, functions and visual objects, thus enabling developers to seamlessly add new features to SLGeometry by implementing them as C# classes annotated with metadata. One of advantages of this concept is the introduction of alternatives for .NET attributes, further development of metadata specification suitable for the description of complex hierarchical

structures and their inter-dependencies, type and operational independent calculation optimization implemented, and metadata developed as plug-in elements.

In this chapter, we present the formulation of the problems that we intend to solve, giving the reader a clearer picture of the problems. We present main advantages over the current state-of-the-art. Finally, we note that results presented in the thesis are based on our previous publications with additional clarifications, and enriched with many useful examples.

1.1 Metadata

Metadata and techniques such as metaprogramming, attribute-oriented programming and component-object programming are widely used in last decades (Chlipala, 2016; Štuikys, Damaševičius, and Ziberkas, 2012; Lilis and Savidis, 2015; Hazard and Bock, 2013). Attribute-oriented programming is a program-level marking technique that allows developers to declaratively enhance programs through the use of metadata. More precisely, developers can mark program elements (e.g. classes, interfaces, methods, fields) with attributes (annotations) to indicate that they maintain application-specific or domain-specific semantics (Noguera and Pawlak, 2007). Attributes are declarative marks, associated with program elements, to indicate that they maintain application-specific or domain-specific semantics (Schwarz, 2004; MSDN, 2019). The existence of attributes can be checked at runtime and actions taken depending on their values. Besides providing data which have an effect on the runtime program behavior, they are used as an indication to developers about the aim and the behavior of the tagged code. Component software design has experienced enormous benefits from the re-usability point of view (Buriková, Steingartner, and Eldojali, 2016; Bose, 2010).

Metadata is data about data in computer science and can be interpreted in different ways according to needs. When observed in object-oriented programming, it is the information about the program structure itself (Guerra, Fernandes, and Fagundes Silveira, 2010). Metadata can be used in various scopes: it started with metadata in public digital libraries (Tarver et al., 2015), pedagogical metadata for learning objects (LO) (Dagiene, Jevsikova, and Kubilinskiene, 2013; Alvino et al., 2009), metadata for ontology description and publication (Dutta, Nandini, and Shahi, 2015), metadata in social-networks (Aggarwa, 2011; McAuley and Leskovec, 2012), metadata in DSL (Kendall, 2016; Mernik, 2013), up to database systems (Ristić et al., 2014). Metadata standards are introduced for simple and generic resource descriptions (*The Metadata Community — Supporting Innovation in Metadata Design, Implementation & Best Practices*, 2019). Most scientific databases consist of datasets with an identical structure which is associated with rich metadata, i.e. descriptive information about the content (e.g. genoms metadata describe data about DNA sequencing), and for their search metadata-aware query languages are used (Pinoli et al., 2019; Guzzi, 2019).

Obviously, a robust metadata specification is required to successfully describe all current and future members of types, type conversions, operations, functions and visuals, and enable their integration into a working DGS.

Due to design restrictions on attributes in the .NET Framework (*Using Attributes in C#*, 2019; Alvi, 2002), only a subset of built-in Common Language Runtime (.NET run-time environment, abbreviated CLR) value types are allowed as attribute properties. Since expressions in SLGeometry are not CLR value types, default values for function arguments cannot be stored in attributes. Furthermore, complex structural information, such as constant and function metadata (described in detail in Section 3.2), is also impossible to store in attributes. We have devised another approach to specifying complex metadata, which is flexible, accessible via reflection and simple to use.

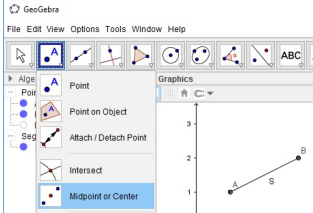
1.2 Functional Domain-Specific Languages

Dynamic Geometry Software (Narboux, 2007; Botana and Valcarce, 2001; Freixas, Joan-Arinyo, and Soto-Riera, 2010; Winroth, 1999; Kortenkamp, 1999; *GeoGebra*, 2019; *Cabri*, 2019; *The Interactive Geometry Software Cinderella*, 2019) is software that enables creation and real-time manipulation of visually rich interactive teaching and learning materials, called *dynamic drawings*. Dynamic drawings are specified by writing expressions in a functional domain-specific language (abbreviated: FLG - Functional Language for Geometry) and assigning them to named variables. In that regard, the set of variables can be considered equivalent to the drawing it represents. The set of types (T), type conversions (C), operations (O), functions (F) and visuals (V) in the FLG is denoted with $\tau = \{T, C, O, F, V\}$. A DGS consists of an expression evaluator (*Engine*) and a front end which displays dynamic drawings on the screen (*GeoCanvas*). Visual objects (*visuals*) are declaratively bound to certain types, and the DGS takes care of drawing them on the *GeoCanvas*. If a visual exists that corresponds to the constant type obtained by evaluating a variable, that visual appears on the screen. Dynamic drawings consist of visual and non-visual objects. Some objects are dependent on other objects, i.e. their expressions contain variable references. For example, a line segment is defined by its two endpoints. Whenever an endpoint is moved, the segment also moves. The Engine can also make use of a Computer Algebra System (CAS) (Ginsburg et al., 1997; *Wolfram Mathematica Comparative Analyses - Computer Algebra Systems*, 2019; *MathWorks - Computer Algebra System*, 2019; Wester, 1999) to manipulate and transform expressions.

A DGS is implemented in a host language (HL) (Chodarev and Kollar, 2016) such as Java or C#. FLG types, functions and operations, as well as visuals, are written as HL classes which conform to some pre-established contracts. For example, all function classes must implement the `Eval()` method. It is possible to use the DGS directly from the HL, by programmatically creating expressions and calling the Engine API. In that sense, the FLG can be considered a language extension of the HL (Mernik, 2013; Erdweg, Giarrusso, and Rendel, 2012), since its classes are compiled with the unmodified HL compiler. Usually, however, the FLG is used as a standalone language inside a dedicated DGS, which provides at least two ways of entering expressions: by textual input through a parser, or by using drawing tools in the graphical user interface (GUI). From this point on, we shall consider C# as the HL of choice.

Dynamic drawings contain visuals such as geometric shapes and other objects, and can be conveniently described by a set of named expressions written in a domain

TABLE 1.1: A dynamic drawing, specified by four expressions, explicit and implicit expression input via text and tools

Expression tree	Explicit expression input	Implicit expression input in GUI
$Point(Number(1), Number(1))$	$A = (1, 1)$	
$Point(Number(3), Number(2))$	$B = (3, 2)$	
$Segment(ValueOf(A), ValueOf(B))$	$S = Segment(A, B)$	
$Midpoint(ValueOf(S))$	$M = Midpoint(S)$	

specific functional language, which are stored in variables as shown in Table 1.1. Dependencies between variables, established by variable references, form directed acyclic graphs which are traversed during recalculation to maintain the consistency of the drawing. Each visual is generated by a function which produces a constant of a specific type that causes the visual to appear on screen. The functions can be roughly divided into the following categories:

Visual functions which return geometric shapes. For example, $Point(2, 3)$ returns a point constant and $Segment(p1, p2)$, where $p1$ and $p2$ are points, returns a segment constant. Other shapes are also created in this way; Geometric operations, which are applied to geometric shapes and perform constructions. For example, $Bisector(s)$ returns a line that is a bisector of the segment s . $Normal(s, p)$ and $Normal(l, p)$ calculate and return a line that is a normal through the point p to the segment s or the line l , respectively;

Property functions which extract or calculate properties of geometric shapes. They have the form $F(v)$, where v is a constant. The results are usually geometric objects and numerical values. For example, $Midpoint(s)$ returns a point constant that is a midpoint of the segment s , while $Area(p)$ returns the area of the polygon p ;

Other functions for example, mathematical, logical, string manipulation, vector and matrix functions.

Our aim is to use SLGeometry as a foundation for experiments in improving the development of teaching materials and games. Since SLGeometry is under our control, we are able to modify it, introduce new features, and develop some new principles of operation. Each UI control is represented by a single variable in SLGeometry, and has a number of properties, which control its appearance and behavior. Some UI controls react to user input, either by mouse or keyboard, and the changes are propagated to appropriate properties. Further, a special kind of custom components which we developed are behavior controller components. They act in a sequential manner, i.e. they have inputs, memory and outputs. With controllers, it is possible to achieve the most common behavior needed in mathematical games, without the need for programming.

1.3 DGS Extensibility

In time, two shortcomings of the current DGS have surfaced and come to our attention.

First, DGS are not well suited for universal application, as they mostly contain geometric objects and functions that operate on them. Tomiczková and Lávička (2013) support this viewpoint, arguing that the majority of available computer-aided teaching materials in geometry are oriented towards fundamental problems in elementary and secondary school mathematics. Authors employed the GeoGebra DGS (*GeoGebra*, 2019), supplemented with a professional-grade 3D modeling software *Rhinoceros*, to present and solve an array of practical engineering problems. In our early paper (Herceg, Herceg-Mandić, and Radaković, 2012) we successfully applied GeoGebra to solving geographical problems, but this effort resulted in very complex dynamic drawings. This is a common occurrence, because many simple geometric shapes need to be combined to represent complex visual objects. The number of those objects and their respective variables can quickly become overwhelming for the user.

Second, geometric objects in DGS carry within them only the minimal necessary amount of data. Additional properties of objects are calculated by applying separate functions to those objects. This way, expressions are light and efficiently evaluated. For example, a constant which represents a linear segment between two points carries only the data for the two points. The midpoint of the segment is calculated only if needed, by using the Midpoint function. One significant disadvantage of this approach is function namespace congestion, because Midpoint shares the same namespace with numerous other functions, although it is limited to a very specific argument type and purpose. This issue becomes pronounced as more geometric objects are added to DGS, because each new object type can have many properties, which all require separate functions to be imported into τ . The other problem affects HL developers of DGS, who have to know the current state and anticipate the future functions in τ and invest additional work to avoid name clashes or provide overloads.

There are two choices available to developers who wish to extend a DGS with new features: either try to include as many different visual objects and corresponding functions into a DGS as possible, or provide an extensibility mechanism, such as plug-ins, and defer the development of new features to plug-in authors. It is evident in GeoGebra and other DGS that many new functions, visual shapes and other functionalities are added in each new version, for example see Tomiczková and Lávička (2013) or Hall and Chamblee (2013).

Having this in mind, our aim was to create a DGS which would mitigate the aforementioned problems. We created a generalized extensible DGS SLGeometry in C# on the .NET Framework, in order to observe the design requirements of such a software and propose a viable implementation of the extensibility framework based on metadata.

1.3.1 Introducing OO Features Into DGS

Many teachers have gained significant experience with DGS, and the scope and complexity of their teaching materials has increased. A large number of examples in geometry, analysis, physics, mechanics, geography and other subjects have been developed. Creating complex drawings from basic geometric shapes, however, can be inconvenient. In time, developers of DGS have started introducing additional geometric shapes and graphical representations of complex concepts, such as numerical integration. This has caused some inherent drawbacks of the functional languages employed in DGS to become obvious. The ever-increasing number of new visuals requires at least as many new functions. Some visuals have numerous properties which are of interest to teachers, (see, for example, properties of triangles in Kimberling (2019)). Accessing these properties leads to complex expression compositions which are difficult to read, contain many parentheses and, due to the functional language semantics, are formed in the order opposite from natural one. For the user, it is difficult to distinguish numerous functions, some of which only apply to a very narrow subset of types or even just one type. DGS are written in classical object-oriented languages, and their functional languages are usually interpreted at run-time. Each function is implemented as a separate entity, i.e. C# or Java class. For the developer, maintaining an increasing set of functions, scattered in many unrelated files can be a tedious task.

The main idea behind this dissertation is to address the above problems by bringing structure to both the syntax of the expressions and to the underlying implementation of the functional languages used in DGS. This can be accomplished by introducing object-oriented features, i.e. support for objects with calculated properties and dot syntax for property access. Our aim is to unify the objects with their properties, both from the users' and the developers' point of view. The obvious advantages of our approach are simpler expression notation (Radaković and Herceg, 2010) and natural support for helper mechanisms during input, such as code completion. In a pedagogical experiment (Radaković and Herceg, 2013) we demonstrated that creating dynamic drawings using the object-oriented approach was better adopted by the students than the functional approach. The benefit for developers is that all properties of an object can be implemented inside a single class, i.e. one source file, rather than being scattered through a number of unrelated functions. The global context, i.e. namespace which contains all available functions, is also freed of a large number of property functions.

A naturally occurring question is how the added complexity in the evaluation engine affects performance and memory footprint, since DGS are used on many low-spec devices such as tablets and phones. In a purely functional DGS, an object contains only the necessary properties that describe it, such as coordinates for a point or endpoints for a segment. In contrast, placing a large number of calculated properties inside an object can quickly lead to a growth in heap space allocation due to deep nesting of objects. A sensible approach is that the properties be instantiated and calculated only if they are referenced. Therefore, it is essential to devise a lazy evaluation scheme which would provide efficient evaluation of potentially infinite structures. Object oriented extensions to functional languages and an approach to function overloading, discussed in Savidis (2006), are, to an extent, applicable to

our work. We also followed ideas for lazy evaluation and path analysis, especially concerning reducing the evaluation costs, from Bloss, Hudak, and Young (1988).

The concrete implementation of the proposed solution is based on the SLGeometry DGS (Radaković, Herceg, and Löberbauer, 2010; Herceg and Radaković, 2011; Herceg, Herceg-Mandić, and Radaković, 2012; Herceg, Radaković, and Herceg, 2012). Computational costs and memory consumption were compared against the purely functional expression evaluation engine. Load tests were performed with dynamic drawings of increasing complexity. The results demonstrate that the modifications introduce only a negligible increase in performance cost for simpler constructions, while the advantage shifts towards the object-oriented engine as the complexity of the drawings increases.

1.4 Main Goals

The overall subject of the research described in this dissertation is object-oriented extension of Dynamic Geometry Software supported with metadata and based on an expression evaluator which supports lazy evaluation of object properties and partial compilation.

This research focuses on creating a framework with:

1. The unification of objects with their properties;
2. Realization of access to features without the use of special functions;
3. Defining the mechanism for extensibility with new types, functions and visual objects;
4. Adopting a strategy of verification, evaluation and optimization;
5. Metadata specification for DGS.

To this end, we have identified the following goals:

1. Our aim is to unify the objects with their properties, both from the users' and the developers' point of view. The obvious advantages of our approach are simpler expression notation (Radaković and Herceg, 2010) and natural support for helper mechanisms during input, such as code completion;
2. Answer on naturally occurring question of how the added complexity in the evaluation engine affects performance and memory footprint, since DGS are used on many low-spec devices such as tablets and phones;
3. Objects must be unified with their properties, so that they can be implemented together in HL, and property access must be realized without the need for separate functions;
4. An extensibility mechanism, such as plug-ins, must be devised for the DGS, which would enable adding new types, operations, functions and visuals to τ ;
5. There must be no differences in treatment of built-in and imported members of τ ;
6. Existing algorithms must be applicable to new types, i.e. highly decoupled and generic;
7. Verification, evaluation and optimization strategies must also be generic;
8. All members of τ must contain all necessary metadata to successfully integrate with the DGS.

1.5 Contributions

The main contribution of this dissertation is the proposed system for efficient management of a set of interactive objects with dynamic properties, which is implemented as an object-oriented extension of a classical expression tree evaluator. Further, we proposed an alternative for .NET attributes, that are structurally designed, and accessible via reflection. Computational efficiency is preserved by the introduced caching scheme, dynamic property activation and lazy property evaluation.

Some of the contributions have already been mentioned throughout previous discussion. Here we summarize all of the main contributions of this dissertation:

1. An alternative for .NET attributes, suitable for representing complex hierarchical metadata is described;
2. An overview of the SLGeometry DGS and its extensibility framework is given;
3. Pervasive use of metadata annotations enables decoupling of general algorithms from concrete semantics;
4. Further development of the metadata specification, suitable for expressing complex hierarchical structures and dependencies is implemented and discussed;
5. Type- and operation-neutral evaluation optimization schemes are implemented, and metadata specification is provided for development of new pluggable implementations;
6. An example of a generic functional language is presented, in which semantics is decoupled from the language implementation and realized in the form of pluggable types, operations and functions. The example is implemented on the SLGeometry DGS, as C# classes annotated with our metadata;
7. Introduction of object data types with calculated properties;
8. A new algorithm for lazy property evaluation;
9. For users, dot notation reduces complexity of written expressions;
10. For developers, placement of logically related code inside single units;
11. No performance penalty in practical applications.

1.6 Summary

In this dissertation, we address the shortcomings of attributes, i.e. restrictions which limit their application to representing complex structured metadata. We proposed a structural metadata infrastructure which is independent of attributes, flexible, accessible via reflection and simple to use.

Further, we provide extensibility to simple and complex data types, unary and binary operations, type conversions, functions and visuals. Developers are empowered to seamlessly add new features to SLGeometry by implementing them as C# classes annotated with metadata.

The presented architecture and the property activation infrastructure, using lazy evaluation, result in substantially lower computational complexity compared to traditional functional solutions.

Chapter 2

Related Work

This dissertation is concerned with metadata-supported object-oriented extension of a Dynamic Geometry Software with lazy evaluation. Thus, related work covers several areas: the first part covers some well-known DGS, the second part covers related work in lazy evaluation, the third part covers metadata and object-oriented extensions and the fourth covers the use of components.

2.1 Dynamic Geometry Software

Since the late 1960s some of mathematics educators supported the role of mathematical applications, models and modeling in the teaching and learning of mathematics. This was based on two compatible ideas: a) utilization of mathematics for applications, models and modeling and b) learning of mathematical concepts through applications, models and modeling (Niss, 2012). In that time the availability of computers has changed the nature of the solution process in mathematics and mathematics curriculum. Furthermore, many conferences held panels giving the recommendations to explore and summarize current thinking about the role of the computer for the curricula in the four fields of science: mathematics, physics, statistics and chemistry. This supported the fact that computer assisted instruction offers unexplored opportunities for improving mathematics instruction and that the real problem is the generation of appropriate learning situation coupled to a computer with well defined educational objectives in view (Underkoffler, 1969).

The use of computers and different software for education constantly interests scientists who discuss and disseminate school curricula, especially in science and mathematics.

Kaufmann and Schmalstieg (2002) observe that using computer-aided design software in high school and university education contributes to better and faster comprehension of geometry problems than using traditional methods. Drijvers et al. (2016) emphasize that new demands are imposed on educational systems in order to prepare students for future professions where technology offers opportunities for teaching and learning; i.e. exploiting these opportunities requires rethinking educational paradigms and strategies.

In the early 1990s, Blum and Niss (1991) reviewed applied problem solving, modeling and applications in mathematics education, extended with use of computers, and its relations to other subjects. In contemporary mathematical modeling, the process of translating between the real world and mathematics in both directions is one of the topics in mathematics education (Blum and Borromeo Ferri, 2009).

STEM (science, technology, engineering, and mathematics) education tries to develop tools and processes for teaching which integrate concepts that are usually taught as separate subjects in different classes and emphasizes the application of knowledge to real-life situations, enabling teachers to teach mathematics and science much better and more effectively (Burkhardt, 2018; Gonzalez and Kuenzi, 2012). Providing early exposure to STEM content can ensure that students will continue their interest in STEM subjects through middle and high school up to university level (DeJarnette, 2012). Nowadays, the new goal is to move from STEM to STE-A-M, i.e. to include arts, design and creation, as one of the successful models of coexistence between art and science education (Lavicza et al., 2018).

The use of computer-aided design software in education, starting from elementary school all the way to the university, contributes to better and faster comprehension of geometry problems than using traditional methods like the so called “chalk and talk”. Sometimes the teachers are willing to learn new tools for teaching geometry, but they are not also ready to implement these tools in their teaching in schools. This appears in particularly in developing countries, mostly in the rural parts where resources for integration of technology in school education are very limited (Mainali and Key, 2012; Bhagat and Chang, 2015; Khalil et al., 2018; Han et al., 2013).

Similar problem appears in Serbia as well, where adequate teaching materials and applications in the Serbian language that teachers can use and display to the students are lacking. Furthermore, teachers are obliged to have adequate skills and knowledge to use computers, and there are curricula instructions compatibility improvement with modern trends also. However, there is an increasing number of teachers who successfully implement information and communication technologies (ICT) in their classroom environment (Radović, Marić, and Passey, 2019; Jezdimirović, 2014; Ljajko and Ibro, 2013; Diković, 2009).

Notwithstanding that the computers are getting more and more into classrooms, the importance of technology in mathematics is accepted at a rhetorical level, since the computers are everywhere, but school mathematics has not changed to reflect this, i.e. the school curriculum still needs to be improved. The use of well known DGS makes the students play a much more active role than in the traditional learning (Burkhardt, 2013).

According to Sarama and Clements (2009) geometry is the the most relevant mathematical subject which lies at the heart of physics, chemistry, biology, geology and geography, art and architecture. Using the Logo environment they showed that computer manipulations can help students build on their physical experiences, tying them tightly to symbolic representations. Also, computer manipulations can encourage students to make their knowledge explicit, which helps them build integrated-concrete knowledge (Sarama and Clements, 2012).

Research and practice in education need to enhance each other through the development of a new set of tools for understanding and supporting powerful mathematics classroom instruction as well as instruction across a wide range of disciplines (Schoenfeld, 2014).

Games, as educational tools, are also used in health care: Maskeliūnas et al. (2018) propose their serious games related to dementia targeting directly at people

working with dementia (the carers themselves) believing that its use will eventually improve the quality of life of both patients and caregivers.

Mathematical visualization is the process of forming images (either mentally, with pencil and paper, or with the aid of technology) and afterwards applying such images efficiently for mathematical discovery and understanding. Visual thinking played a dominant role in the thinking of some of great scientists like Faraday, Galton, Tesla etc. (Shepard, 1978). Computers' impact on visual reasoning in mathematics education is undeniable and learner's use of visualization through DGS which facilitates visualization processes has been discussed in the last few decades (Jezdimirović, 2014).

In the late 1950s, in the first decade of the computer age, Wang (1960) developed a program on an IBM704 computer that proved 220 theorems in propositional logic offered by Whitehead and Russell (1910) in the Mathematical Principle. Gelernter and his co-workers at the IBM Research Center in New York (Gelernter, Hansen, and Loveland, 1960; Gelernter, 1963) developed a system for Euclidean geometry. This geometry machine was able to discover proofs for a significant number of interesting theorems within the domain of its ad hoc formal system (comprising theorems on parallel lines, congruence, and equality and inequality of segments and angles) using the diagram only to indicate which sub-goals are probably valid.

In 1978 Wen-Tsün introduced an algebraic method for mechanical proofs of geometry theorems, known as Wu's method (Wu, 1978; Wu, 1999), where he explains how to translate geometrical problems into polynomial equations. He also founded the School of Mathematics Mechanization and re-studied ancient Chinese mathematics.

Following theorem provers use algebraic style or synthetic style methods: the area method, an efficient coordinates-free method for a fragment of Euclidean geometry, developed by Chou, Gao, and Zhang (1994); algebraic method which is based on some elimination procedures proposed by Wang (1993); synthetic proof methods (Gelernter, 1995); the Gröbner Cover algorithm Montes and Wibmer (2014).

It is important to mention that the proof theory has been deeply influenced by the Haskell Curry and William Howard correspondence between propositions and types, where each formal proof can be seen as a functional program (i.e. the proofs-as-programs paradigm) that meets a specification given by the proved formula (i.e. the formulæ-as-types paradigm). It also led to the development of proof assistants such as Coq, MinLog, Agda (Howard, 1980; *The Coq Proof Assistant*, 2019; Coquand and Huet, 1988; Agda, 2019).

In 1963, Sutherland introduced Sketchpad (Sutherland, 1963), where the user sketches directly on a computer display, as a novel complete graphical user interface. It can be said that it was the pioneer of human-computer interaction (HCI) and also the ancestor of modern CAD programs.

The use of mathematical software, starting with Computer Algebra Systems (CAS) on HP calculators, and continuing with, e.g. Mathematica and Maple, has made significant influence in teaching of mathematics. Some researchers use CAS and DGS for the mathematical modeling of a real-life problems as a problem-solving activity

that suits the purposes of mathematical learning, which contributes to the understanding of known mathematical concepts and establishing interdisciplinary relationships (Zbiek and Conner, 2006; Aktümen, Horzum, and Ceylan, 2013).

Along with the development of automatic provers occurred the need for developing accompanying graphic interfaces which allow interactive manipulation by mouse dragging and manipulation of mathematical expressions in symbolic form. Commercial products such as Cabri (Cabri, 2019; Laborde and Capponi, 1994), Cinderella (*The Interactive Geometry Software Cinderella*, 2019) and Geometer's Sketchpad (*The Geometer's Sketchpad*, 2019; Jackiw and Finzer, 1993), cover high and middle school mathematics and physics education through algebra, analysis, geometry, trigonometry, mechanics and optics, as well as many other subjects. Books, tutorials and forums contain many teaching materials with presented and solved problems.

In the last few years, besides well-known commercial DGS emerged new free and open source dynamic geometry systems, some of them having functionality of computer algebra systems.

The SageMath (Software for Algebra and Geometry Experimentation) project (Stein, 2019b; Stein, 2012), created by William Stein professor at the University of Washington, is a viable free open source mathematics software system built out of nearly 100 existing open-source packages like NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R. It is an alternative to Magma, MapleTM, Mathematica and MATLAB, which are closed, i.e. it is not possible to modify or extend them, exactly opposite to SageMath. Online computing extension is the CoCalc (Collaborative Calculation in the Cloud) (Stein, 2019a) – online workspace for mathematical calculation, statistics, data science, document authoring with collaborative environment and tools for teaching.

GeoGebra (*GeoGebra*, 2019; *GeoGebra Materials*, 2019) is a free DGS, created by Markus Hohenwarter in 2001, which contribute to a more comprehensive set of features integrating geometry, algebra, calculus, statistics, graphing, spreadsheets and 3D augmented reality, and it is maintained by an international team of developers. A large self-sustained community of users helps students and teachers with numerous examples of teaching materials and offers help through online forums. It had become the leading provider of DGS worldwide as software for teaching and learning: available in many languages, organizing International GeoGebra institutes (IGI) which focus on training and maintain on-line support system for teachers, develop and share workshop resources and classroom materials, conduct and implement research projects in teaching and learning mathematics and other STEM subjects.

Users can create and manipulate geometric constructions in GeoGebra, they discover conjectures and interactively build formal proofs with the support of Coq, a system which allows users to construct fully traditional proofs in the same style as the ones in school (Pham and Bertot, 2012). In the last years GeoGebra was extended with several automated deduction tools as presented by Botana et al. (2015) along with the educational scenarios that could be supported by automated reasoning features. Both, teachers and students, can benefit from these scenarios.

In the last couple of years, a tremendous number of works has been accumulated on technology use in mathematics education and visual learning of mathematics. There are thousands of pre-made GeoGebra illustrations that teachers can use in

their classrooms to motivate and teach with, to help pupils better learn and understand mathematics concepts (Furner and Marinas, 2016). GeoGebra Applets can be used to visualize the concepts taught in the Calculus course, such as limits, derivatives, integrals, in order to facilitate the teaching and learning, since it is even more difficult to understand when only static images are used, no matter how good they are (Caligaris, Schivo, and Romiti, 2015).

GeoGebra can be used with Critical Thinking skills as reflective tools to develop student understanding of probability (Aizikovitsh-Udi and Radakovic, 2012). There are many researches that investigated students' achievements and analyzed the contributions and benefits of GeoGebra in teaching mathematics (Hohenwarter et al., 2009; Reis, 2010; Tatar, 2013; Takači, Stankov, and Milanovic, 2015). By using parametrization it is possible to visualize geometric objects, discover and prove formula correctness, etc. (Pech, 2012).

An interactive environment, joining open source software (Sage, GeoGebra and InterGeo), a dynamic geometry system and a computer algebra system is presented in Botana and Abánades (2014). It provides robust algebraic methods to handle automatic deduction tasks for a dynamic geometry construction, and it is illustrated with several examples of loci and envelopes.

GCLC (Janičić, 2019; Janičić, 2010; Janicic, Narboux, and Quaresma, 2012) is a mathematical tool for visualizing geometry, teaching geometry and producing mathematical illustrations based on formal descriptions. It was initially built as a tool for producing LaTeX figures from descriptions of geometric constructions. It also contains automated theorem provers. It has been integrated as one of the theorem provers in GeoGebra (Botana et al., 2015).

ArgoTriCS (Automated Reasoning GrOup Triangle Construction Solver) (Marinković, 2016; Schreck et al., 2016) is a system that automatically solves triangle construction problems using some background geometrical knowledge.

The common feature of the aforementioned DGS is that dynamic drawings are specified using the functional approach.

2.2 Lazy Evaluation

In the late 1970s and early 1980s idea of lazy (non-strict) languages appeared as a response to Scheme (Sussman and Steele Jr, 1975; Sussman and Steele Jr, 1998), Milner's meta-language ML (Milner, 1978) and some other strict (call-by-value) languages. Lazy evaluation, also called call-by-need, is used mostly in functional languages as an evaluation strategy. The lazy evaluation approach is intuitive: expressions and sub-expressions are evaluated at run-time only as they are needed, and if so, they should be evaluated only once. This enables infinitely recursive data-structures to be expressed directly in the language, and allows for improved performance in some cases by leaving unnecessary calculations unevaluated (Sinot, 2008; Peelar, 2016).

According to Hudak et al. (2007) lazy evaluation has been invented independently three times:

- The technical report by Friedman and Wise (Friedman and Wise, 1976) introduces the Cons constructor function and redefines five elementary functions

presented by McCarthy to postpone evaluation of its arguments, getting it from a LISP perspective;

- Henderson and Morris (Henderson and Morris, 1976) presented algorithms that delay the evaluation of parameters and list structures in LISP programs;
- Kent Recursive Calculator (KRC) developed by David Turner (Turner, 1982; *Kent Recursive Calculator*, 2019) is a “miniaturised” version of SASL (St Andrews Static Language), used for teaching functional programming at the University of Kent and Oxford University in the early 1980s.

Subsequently, by the mid-1980s there were a lot of researchers who independently designed their own pure lazy languages, intently concerned with both design and implementation techniques:

- Miranda is the direct successor to KRC, with a non-strict (i.e. “lazy”) semantics developed by David Turner (Turner, 1985);
- Lazy ML is a strongly typed, statically scoped functional language with lazy evaluation and an abstract graph manipulation machine, the G-machine by Johnsson (1984) and Augustsson (1984);
- Orwell, a lazy language developed by Wadler, as a free alternative to Miranda (Wadler, 1988);
- Id, a non-strict dataflow language developed at MIT by Arvind and Nikhil (Arvind, Nikhil, and Pingali, 1989; Nikhil, Pingali, and Arvind, 1986), whose target was the MIT Tagged-Token Dataflow Machine.

As reported by Hudak et al. (2007) at the time Haskell was born, by far the most mature and widely used non-strict functional language was Miranda by which Haskell’s design was, therefore, strongly influenced. Laziness was undoubtedly the single theme that united the various groups that contributed to its design. So, technically, Haskell as a language with a non-strict semantics and lazy evaluation is simply one implementation technique for a non-strict language.

By the early 1990s, considering a growing interest in lazy functional programming languages, as Miranda had a well supported implementation, and a nice interactive user interface, it was installed and taught at 250 universities (Bird and Wadler, 1988; Joosten, Van Den Berg, and Van Der Hoeven, 1993). Van Den Berg (1995) in his doctoral thesis addresses some issues on the quality of software with respect to the programmers where the experimental findings supported the main hypothesis that programmers perform better on structured Miranda function definitions than on non-structured definitions.

A class of program schemes for use in programming methodology for lazy functional languages is shown by Gilst and Broek (1995) using a divide and conquer strategy, dividing problems into sub-problems, which are given by a call graph for functions. Frost and Karamatos (1993) have shown that the integration of lazy functional programming and attribute grammar paradigms is straightforward.

In functional programming (Backus, 1978; Slodičák and Macko, 2011) lazy evaluation enables the use of powerful structures such as infinitely long lists, but typically with penalties in execution time compared to some imperative languages. The paper by Bloss, Hudak, and Young (1988) presents a detailed overview of several optimization techniques for thunk evaluation, including path analysis, which are applicable

to non-strict functional languages. Ariola et al. (1995) showed equivalence between the call-by-name and call-by-need evaluator and proved its correctness with respect to the original lambda calculus, i.e. that the call-by-need theory is a strict sub-theory of the call-by-name lambda-calculus.

Hughes (1990) emphasizes modularity of a programming language as support to programmers, by means of decomposing a problem and gluing solutions together – which is enabled in functional programming languages by higher-order functions and lazy evaluation. Hughes stressed the need for lazy evaluation as the most powerful tool for modularization in functional programming.

Researchers in the Intel corporation (Seger et al., 2005) designed the Forte verification environment for data-path-dominated hardware. It uses FL, a full-featured lazy functional programming language to script proof efforts. FL provides a specification language for hardware, where the implementation language of the Forte theorem prover as well as the term language for its higher-order logic are included.

Chang (2013) demonstrates the theoretical and practical feasibility of a semantics-based refactoring that helps strict programmers manage manual lazy programming. Developed tool uses laziness analysis to automatically insert delay and force expressions as needed. Barzilay and Clements (2005) stress that usually students have trouble understanding the difference between lazy and strict programming. They used PLT Scheme’s syntax to add extension of laziness to show students in a programming languages course the difference between strict and lazy languages in isolation, i.e. students could compare strict and lazy evaluations of the same program text.

Another use of lazy evaluation is implemented in LazyJ (Warth, 2007) – the extension of the Java programming language where variables with type lazy T are evaluated with a recipe for their computation. Kiselyov, Peyton-Jones, and Sabry (2012) proposed simple generators for the pretty-printing problem to complement or supplant lazy evaluation in stream-preprocessing programs.

Object-oriented extensions to functional languages have been explored in Delta (Savidis, 2006), Wolfram Mathematica (Maeder, 1993) and many other languages. Classes and objects are first class citizens in functional languages such as F# (Podwysocki, 2019).

2.3 Metadata

Mechanisms which enable the attachment of arbitrary metadata to portions of a program have become more and more common in programming languages. Many authors use attributes for metadata specification. An attribute carries the information in two ways: its type and the values of its properties.

The specification of meta-information, by using attributes in .NET or annotations in Java, attached to parts of a program is widely used among programmers. Some researchers noticed certain shortcomings of existing metadata specification mechanisms, such as validation of the correctness of metadata or re-usability. Even though there exist code completion abilities of contemporary Integrated Development Environments (IDE) such as Eclipse or Visual Studio, they can not determine if annotations are used in the right place or if there will be some errors at deployment-time or even at run-time. Some of solutions are presented next.

The annotations mechanism in Java attaches custom, structured meta-data to declarations of classes, fields and methods. It has some limitations because annotations can be attached to declarations and their instantiation can only be resolved statically. Cazzola and Vacchi (2014) proposed an extension to Java (called @Java) where they introduced block and expression annotations which allow every annotation to hold dynamically evaluated values.

Noguera and Duchien (2008) observe that the validation of constraints in the use of annotations requires more engineering support. Thus, they propose a tool that allows to specify the annotations by adding to their declarations meta-annotations that define wanted constraints, and then they are validated using a given domain model at compile time. Eichberg, Schäfer, and Mezini (2005) check the correctness of the properties of annotated elements, defined with constraints, via a user-extensible framework.

Cepa and Mezini (2004) check the correctness of using custom attributes in .NET by providing meta-attributes that define dependency relationships between custom attributes, i.e. stating that a certain attribute requires or excludes another attribute.

Song and Tilevich (2015) stress that the mainstream metadata formats, such as annotations in Java, attributes in .NET or XML configuration files, are not sufficiently expressive to be systematically reused. Annotations ease a program understanding but they provide no structural or summary information. They present a new format (PBSE - Pattern Based Structural Expressions) for specifying metadata reuse in program code, its advantages over attributes are discussed and a translator from attributes to PBSE is developed.

Plux.NET (Löberbauer et al., 2010; Jahn et al., 2013) is a framework that allows building of extensible applications consisting of an ultra-thin core using dynamic plug-and-play composition for .NET. It uses the analogy of slot-extension, where the slot declares the kind of information a host expects, and extensions provide this information.

Schult and Polze (2002) have developed tools which allow automatic generation of proxy classes and a replica management mechanism that deals with crash faults of objects. They discuss the usage of aspect-oriented programming techniques in the context of the .NET framework using C# custom attributes for non-functional component properties. Using a simple calculator as a case study they showed how to express non-functional component properties without any programming language extension using attributes.

Berzal et al. (2005) stress the need for mechanisms which allow the representation and management of fuzziness in the object-oriented data model. They propose a framework which provides management of imprecision in description (such as age, name, height, room floor etc.) of objects, i.e. an easy-to-use transparent mechanism that can be used to develop applications which deal with fuzzy information. The framework defines a metadata attribute, FuzzyImportance, which is used to indicate the weight of a property during comparison. In this way the implementation of fuzzy user-defined classes is easy and can be used without interfering with the usual object-oriented software development tasks.

Greaves and Singh (2008) use attributes for hardware specification, such as I/O port declarations, register widths, clocks, etc. They are used by the Kiwi compiler

as input, along with .NET assembly language, producing Verilog output. The implementation is described through examples. Benton, Cardelli, and Fournet (2004) proposed a join-based syntactic extension of the C# language with modern concurrency abstraction for asynchronous programming called Polyphonic C#. They used `OneWay` attribute which indicates to the .NET infrastructure that appropriate methods should be genuinely non-blocking. Using the `Dependency` attribute, Casero, Cesarini, and Monga (2003) have developed a tool which extracts attribute-defined dependencies from components and helps programmers in sub-classing. The tool shows how far a modification to a method would propagate and prevents undesired behaviors. A programmer can easily see dependencies in a flat fashion (the set of all members directly or indirectly influenced) or can see all possible paths of execution toward a given member.

Solutions other than attributes have also been used successfully. In (*Windows Presentation Foundation*, 2019) (WPF), component metadata are specified using custom classes, which are instantiated and assigned to a static field of each annotated class. This approach enables the use of arbitrary data types, as well metadata discovery and querying at runtime, which is a foundation for the extensibility of the WPF.

Nosál', Sulír, and Juhár (2016) examine source code annotations from the viewpoint of formal languages, their abstract syntax, concrete syntax, and semantics. They scrutinize a set of all annotations and their parameters processed by the same reference implementation, thus showing how pure embedding with annotations can be used for language unification, language referencing by extension, and language extension. The paper by Sulír, Nosál', and Porubän (2016) regards an interesting idea of recording developer's intentions behind a piece of code, and explores the correspondence of the mental model of a problem with the actual code, as well as the benefits for program comprehension and correctness.

Erdweg, Giarrusso, and Rendel (2012) present a classification of language composition. This classification and Robot language is also used for case studies by Mernik (2013) and Chodarev et al. (2014). Mernik uses attribute grammars as a formalism for language supported by LISA compiler generator tool, while Chodarev et al. use abstract syntax supported by the YAjCo parser generator.

A language composition editor Eco (Diekmann and Tratt, 2014) is an excellent example of an incremental parser with language boxes.

2.4 Components

According to Heineman and Council (2001), a software component is a software element that conforms to a component model and can be independently deployed and composed without modification, according to a composition standard. Component-Based Design (CBD) addresses issues related to providing, developing, and integrating such components in order to improve reuse (Bourque and Fairley, 2014). Component-based software development is associated with a shift from statement-oriented coding to system building by plugging together components. Reused and off-the-shelf software components should meet the same security requirements as new software. In general, reuse of software has obvious advantages, such as reduced

development and maintenance costs and a positive impact on software quality (Pree, 1997).

The component technologies that meet these definitions include Java and Enterprise Java Beans (introduced by Sun Microsystems), the COM (Component Object Model), DCOM (Distributed Component Object Model), and .NET components from Microsoft Corporation, and CORBA (Common Object Request Broker Architecture). The systems which are developed as component-based products (Bass, Clements, and Kazman, 2012) are able to take faster advantage of new products and new technology, significantly reduce time-to-market, increase employee productivity. They are also more reliable, more changeable and more extensible systems.

Chen et al. (2007) have proposed a model supporting component-based programming using processes to model application programs, and glue programs to build new components from existing ones. ComponentJ (Seco, Silva, and Piriquito, 2008) is a Java-like programming language with the basic idea to be employed as a glue language for existing components that are afterwards used in standard Java code. ArchJava (Aldrich, Chambers, and Notkin, 2002) is used for expressing software architectural structure, thereby providing the confirmation for implementation of the specified architecture at every stage of the software life cycle.

The Dream framework, i.e. a domain specific type system for messages and components that manage messages is presented by Bidinger et al. (2005). Bruneton et al. (2004) define a hierarchical component model with sharing, Fractal, that supports an extensible set of component control capabilities.

As already mentioned in the previous section, Rouvoy and Merle (2006) have presented an abstract component model as an annotation framework which assembles the basic concepts of the abstract component model. Fabresse, Dony, and Huchard (2008) propose an extended version of the Simple Component Language (SCL) that fulfills five requirements: decoupling, adaptability, unplanned connections, encapsulation and uniformity, that need to be satisfied to support Component-Oriented Programming based on an analysis of the state-of-the-art and the limitations of existing work.

Granström (2012) introduces the notion of world map and shows that worlds and world maps form a category with arbitrary products. The construction of the category is carried out entirely in type theory and directly implementable in dependently typed programming languages. After replacing the notion of world map with the standard notion of component, he obtains a rigorous paradigm for component-based development.

The notion of Abstract Behavior Type (ABT) as a higher-level alternative to Abstract Data Type (ADT) is introduced by Arbab (2005). They propose it as a proper foundation model for both components and their composition, which encapsulate data structures behind operations that manipulate them, and hide the details of those operations as well.

2.5 Discussion

Nowadays, software is complex, and development in parts is enabled with the introduction of software components. Many modern programming languages, development tools and development methodologies support the development of software in parts, where different teams cooperate and have different development cycles for different parts of the final software product.

In the sense of Lakos (1996), components embody a subset of the logical design that makes sense to exist as an independent, cohesive unit. Classes, functions, enumerations and so on are the logical entities that make up these components. We wanted to introduce this principle into DGS by allowing component development, independent of the development of DGS itself. These components are included in DGS at run-time and have the same treatment as first-class citizens in DGS. The natural extension of this approach is to provide visual components that, beside their program logic aspects (properties and behavior), also have a graphical representation. Our components are either interactive visual controls (UI controls) or sequential behavior controllers. Their use is further demonstrated in Chapter 6.

Visualization in modern teaching is of immense importance. New generations like to see and feel what they are learning and want to try it themselves. Dynamic Geometry Software enables these learning concepts and encourages the pupils/students autonomy. We can notice that in the given overview of DGS state-of-the-art, the widely accepted approach for specifying dynamic components is the functional approach, i.e. object properties are defined with separate functions from those objects, whilst we introduce OO features and dot notation into DGS.

The use of attributes as mentioned by Löberbauer et al., 2010, Jahn et al., 2013, Schult and Polze, 2002, Berzal et al., 2005, Greaves and Singh, 2008, Casero, Cesarini, and Monga, 2003, Benton, Cardelli, and Fournet, 2004 was initially used in SLGeometry, but was found insufficient, as discussed in Chapter 3, and later matured to our metadata model (Chapter 5) where we followed the idea of structured component metadata (*Windows Presentation Foundation*, 2019).

We leaned on the idea presented in Sulír, Nosál', and Porubán (2016), Nosál', Sulír, and Juhár (2016), providing a way to express correspondence between mathematical definitions of geometric objects' properties and the code to correctly implement them in SLGeometry. To implement our metadata model as by Nosál', Sulír, and Juhár (2016), we also apply some kind of binding rules, i.e. each annotated part with metadata targets some program element, while metadata references are used as with the reflection mechanism and can be used as a plug-ins (Chapter 5). We also expect that the bindings between metadata and their target elements are meaningful (see Sections 5.4.1 and 5.4.2). Tansey and Tilevich (2008) present their solution for annotation refactoring and upgrading scenarios to solve the Vendor and Version lock-in problems used in annotation-based frameworks. This solution differs from ours since it makes automatic transition between annotation-based versions of frameworks, while our framework enables automatic generation of plug-ins based on metadata from manually annotated code.

Our extensible framework leverages the common Component Object Programming practices. After identifying the core structures, mechanisms and behaviors in

a typical DGS, we devised the abstract model of an expression evaluator, and a set of code annotations (metadata) to provide concrete functionality (Rouvoy and Merle, 2006). Principles of template metaprogramming are employed in SLGeometry to develop the generic operation meta-algorithm, which supports pluggable operation implementations depending on data types, see Mulansky and Ahnert (2011) and references cited therein.

Our findings correspond to those expressed by Guerra and Fernandes (2013), that is, frameworks bring benefits to software design, such as higher reuse level, coupling reduction and increase of productivity of developers (Siqueira, Silveira, and Guerra, 2016).

The introduction of object-oriented features in our system (Section 3.4.2) represents a step in the direction explored by Mernik (2013), Erdweg, Giarrusso, and Rendel (2012), Chodarev et al. (2014), while in laying the groundwork for partial expression compilation we followed the guidelines from Palmer and Smith (2011).

One of the language workbenches that support language composition on the semantic level is SugarJ (Erdweg et al., 2011), which enables syntax and semantics extensions of a language via plug-ins. This is similar to our work, where the base language FLG contains only the basic semantics and generic actions. The semantics is added via plug-ins. The difference between SugarJ and FLG is that the syntax of FLG cannot be changed via plug-ins, which is a design limitation. However, in the designated domain of application, semantic extensibility of FLG is sufficiently appropriate.

The functions in FLG can be considered as semantic boxes, as they carry within them arbitrary HL code, which represents action units embedded in FLG (Section 3.4.1). Different developers can implement different functions, and the final user will be able, by using function composition, to achieve action composition, without breaking the enclosing functional paradigm of FLG. Our approach to metadata in DGS serves as a foundation for the development of the editor environment for non-programming users with the similar aim as presented by Nosál', Porubän, and Sulír (2017).

Lazy evaluation is still one of the very popular strategies for evaluation, in the manner of the expression and sub-expression evaluation at run-time only as they are needed, and only once. Considering our objects with large numbers of properties, such as the triangle (Section 7.2), we needed to implement lazy evaluation in our framework. Once an object's property is denoted for evaluation, a property activation mechanism starts the evaluation of required properties, and after they are all evaluated the given property can be evaluated. This approach is described in Section 5.4.

2.6 Summary

In this chapter we briefly overview the state-of-the-art concerning some well-known DGS, lazy evaluation, metadata and object-oriented extensions, and use of components. Also, we discussed how our results follow these established principles and add value to them.

Part II

Implementation

Chapter 3

Motivation

The goal of this chapter is to present several motivating examples which highlight the problems in implementation and use of DGS with purely functional languages, and demonstrate how they can be overcome by using our metadata infrastructure in SLGeometry.

First, we give an overview how attributes in .NET are used. In Section 3.2 we give the main reason why we could not use attributes anymore for our system, as it became more complex. Section 3.3 discusses advantages of objects with properties as an alternative to a large number of property functions, which contributed to introducing the dot notation into FLG. The last two sections introduce semantic extensions and give an overview of language composition consideration in FGL.

3.1 Attributes in C#

We can say that an attribute is like an adjective, which gives more information about some program entity. It is a declarative tag that is used to convey information to runtime about the behaviors of various program entities like types, classes, methods, modules, properties, structures, enumerators, assemblies etc. in a program (*Using Attributes in C#*, 2019; *Writing Custom Attributes*, 2019; *C# - Attributes*, 2019; Alvi, 2002; Agarwal, 2013).

The .NET framework stipulates two types of attribute implementations:

Predefined Attributes All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly;

Custom Attributes Any additional information that is required can be specified using custom attributes.

In the .NET Framework predefined attributes are: *AttributeUsage* – describes how a custom attribute class can be used; *Conditional* – marks a conditional method whose execution depends on a specified preprocessing identifier; and *Obsolete* – marks a program entity that should not be used.

Attributes have the following properties:

- Attributes add metadata to a program, e.g. information about the types defined in a program;
- One or more attributes can be applied to entire assemblies, modules, or smaller program elements such as classes and properties;
- Some attributes can be specified more than once for a given entity;

- Attributes can accept arguments in the same way as methods and properties;
- The metadata is examined by using reflection.

To create an attribute which has some additional information, we use custom attributes by defining an attribute class, a class that derives directly or indirectly from *System.Attribute*, as in Listing 3.3, which makes identifying attribute definitions in metadata fast and easy.

An attribute is specified by placing the name of the attribute enclosed in square brackets (e.g. `[NamedArgsSignature("A","r")]`, line in Listing 3.2) above the declaration of the entity to which it applies. By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code (e.g. attribute *NamedArgsSignature* can be used instead of *NamedArgsSignatureAttribute*).

Attributes can have parameters, which can be:

- Positional** Any positional parameters must be specified in a certain order and cannot be omitted, and they are specified first;
- Unnamed** Any unnamed parameters used in the attribute are passed to the constructor in the order in which they are given;
- Named** Any named parameters are used as initialization statements after the instance has been constructed, they are optional and can be specified in any order.

The C# specification recommends to use named parameters, because there is less possibility to become invalidated, while with positional parameters the position of the parameters is defined in the constructor. The named parameters can be assigned to initial values by using their names. The following form describes the attribute specification [*attribute*(*positional – param – list*, *named1 = value1*, *named2 = value2*, ...)]. The target of an attribute is the entity which the attribute applies to. By default, an attribute applies to the element that it precedes.

After an attribute is associated with the program entity, the attribute can be queried at run-time using reflection, which provides objects that describe assemblies, modules and types (*Reflection (C#) (2019)*, *Retrieving Information Stored in Attributes (2019)*, Listing 3.1).

LISTING 3.1: Accessing attributes with reflection

```
System.Reflection.MemberInfo info = typeof(FCircle);
object[] attributes = info.GetCustomAttributes(true);
for (int i = 0; i < attributes.Length; i++){
    System.Console.WriteLine(attributes[i]);
}
```

To summarize, attributes are mostly used for:

- Describing component object model properties for classes, methods, and interfaces;
- Calling unmanaged code;
- Describing assembly in terms of title, version, description, etc.;

- Describing which members of a class to serialize for persistence;
- Describing how to map between class members and XML nodes for XML serialization;
- Describing the security requirements for methods;
- Specifying characteristics used to enforce security;
- Controlling optimizations by the just-in-time compiler;
- Obtaining information about the caller to a method.

3.2 Why Not Use Attributes?

As was seen in the previous section, the use of attributes is multipurpose. In this section we explain why we needed to propose a new metadata infrastructure when we encountered insurmountable obstacles.

As an illustration we use a circle. The circle as a geometric object in geometry can be defined in several ways. One way is defining a center of the circle and a radius. To define the circle in SLGeometry function *FCircle* needs the following data: a name of her call 'Circle' and a return type *CCircle*; arguments data: *A* – a center of a circle (a point), *r* – a radius (a number); a thickness of the circle; signatures: *A, r* – circle is defined with a point *A* and a radius and *r* – a center of circle is a point (0,0) with radius *r*.

The source code in Listing 3.2 shows how custom attributes were used to annotate the Circle function in SLGeometry. The metadata required to describe the function contains the following information:

- Function name and return type;
- Argument names and types, one occurrence for each argument;
- Function signatures, one occurrence for each signature.

LISTING 3.2: Attribute annotations for the FCircle function

```
[Function("Circle", typeof(CCircle))]
[NamedArgument("A", typeof(CPoint))]
[NamedArgument("r", typeof(Number))]
[VisualProperty("Width", typeof(Number))]
[NamedArgsSignature("A", "r")]
[NamedArgsSignature((Expr)new CPoint(0,0), "r")]
public class FCircle { }
```

The *FunctionAttribute* is presented in Listing 3.3 as an example. It contains the metadata about the function name and its result type. The *NamedArgumentAttribute* and *NamedArgsSignatureAttribute* are quite similar and thus omitted for brevity.

LISTING 3.3: The FunctionAttribute class in C#

```
[AttributeUsage(AttributeTargets.Class)]
public class FunctionAttribute : Attribute{
    public FunctionAttribute(string name, Type resultType){
        Name = name;
        ResultType = resultType;
    }
    public FunctionAttribute(string name): this(name, null) { }
    public string Name { get; private set; }
    public Type ResultType { get; private set; }
}
```

We could use custom attributes to describe all functions needed for any DGS, but a problem occurs when the function metadata is more complex and has default values that are not standard .NET types, i.e. Common Type System (CTS) constants. A special case of the Circle function is defined only with the radius, i.e. the center is assigned a default value, point (0,0) represented with the *CPoint* class instance (the last attribute in Listing 3.2). The *NamedArgsSignatureAttribute* specified in this way is accepted by the C# code editor, but it is not possible to use this constructor since a compilation error occurs: ‘Attribute constructor parameter ‘p’ has type ‘Expr’, which is not a valid attribute parameter type’. This behavior is by design, as the parameters to attribute constructors are limited to simple types/literals and arrays of those types (*Using Attributes in C#, 2019*; Alvi, 2002) which prevents their use for the annotation of C# classes that comprise τ in SLGeometry. Since default values of named arguments are impossible to specify with attributes, we needed another solution (see Section 5.1 for a detailed discussion).

Another argument against attributes is that they cannot represent structural relationships. In the example in Listing 3.2 the FCircle class is annotated with six attributes which only make sense if observed together. Should any of the six attributes be omitted, the information carried by the remaining ones would become incorrect or meaningless. Although attribute validation could be performed at run-time using reflection, it would only detect errors, not help avoid them in the first place.

Our solution to this problem was to develop custom C# classes to represent metadata. The same function, annotated with our metadata classes, is shown in Listing 5.12, Section 5.1. By convention, function metadata is stored in the public static Metadata field, which is assigned in the static class constructor. Two important points must be noted in this example. First, all metadata related to the function is contained within a single FnInfo object, and second, metadata for named arguments is assigned to static variables so it can be later quickly referenced from the C# code.

With our custom metadata classes, we were able to obtain the following advantages over attributes:

1. Values of non-CLR types are allowed in metadata;
2. Structural relationships are correctly represented;
3. Correctness checks are performed in constructors of metadata classes;
4. The IntelliSense and AutoComplete features of the Visual Studio IDE provide help to developers while writing metadata.

3.3 Property Functions vs. Objects with Properties

In most DGS, objects are lightweight and carry within them only the mandatory properties (*GeoGebra, 2019*; *Cabri, 2019*; *The Interactive Geometry Software Cinderella, 2019*). In SLGeometry, objects can have many additional properties. Let us consider the triangle as a geometric concept. It has three vertex points as mandatory properties. Besides, it has many additional *calculated properties* which can be calculated from the values of mandatory properties by some algorithms. A subset of those properties is listed in Table 3.1.

If each of the calculated properties was implemented as a separate function, the number of functions in the global namespace would increase significantly. One must

TABLE 3.1: A subset of calculated properties of a triangle

Property	Type
A, B, C	CPoint
Area, Perimeter	Number
SideA, SideB, SideC	CSegment
AngleA, AngleB, AngleC	CAngle
MedianA, MedianB, MedianC	CSegment
Centroid, Orthocenter	CPoint
AltitudeA, AltitudeB, AltitudeC	CSegment
Incircle, Circumcircle	CCircle

have in mind that any object type, such as segment, circle, ellipse, polygon etc. can introduce multiple new properties, each requiring a new function. From the point of view of a DGS user, this leads to an overwhelming number of functions to choose from when writing expressions. From the point of view of a DGS developer, care must be taken when writing new function implementations, to avoid name clashes and provide correct overloads when a function operates on different types. To mitigate these problems, we decided to include calculated properties into object types in SLGeometry, thus eliminating the need for a separate function for each property. Instead, the properties are declared in the object type metadata, inside the corresponding C# class. For each property, its name, type, dependencies and property evaluator delegate are specified. A property evaluator is a method within the class which calculates the value of the property and stores it into the backing field. The property infrastructure within the Engine takes care of invoking the delegate when needed, i.e. only if the property is referenced from within an expression. This way, only the required properties are calculated.

In Listing 3.4, a fragment of the CSegment object constant type implementation is presented, which shows how mandatory properties (A, B) and calculated properties (Length, Midpoint, Bisector) are declared in the metadata. The metadata for each property contains its name, type and the name of a backing private field, used to store the property value for easy access. The metadata for calculated properties contains additional information about property evaluators and dependencies. Dependencies are properties which must be calculated before the value of the current property can be calculated. For example, the Bisector property metadata specifies that the Midpoint property must be calculated beforehand.

LISTING 3.4: The CSegment class annotated with property metadata

```
public class CSegment: ConstObject{
    // Property metadata
    public static ConstObjectInfo Metadata;
    public static readonly PropInfo AProperty =
        new PropInfo("A", typeof(CPoint), "_a");
    public static readonly PropInfo BProperty =
        new PropInfo("B", typeof(CPoint), "_b");
    public static readonly PropInfo LengthProperty =
        new PropInfo("Length", typeof(Number), "_length");
    public static readonly PropInfo MidpointProperty =
        new PropInfo("Midpoint", typeof(CPoint), CalcMidpoint, "_midpoint");
    public static readonly PropInfo BisectorProperty =
        new PropInfo("Bisector", typeof(CLine), CalcBisector, "_bisector",
            MidpointProperty);
}
```

```

static CSegment() { // Metadata initialization
    Metadata = new ConstObjectInfo(typeof(CSegment), AProperty, BProperty,
        MidpointProperty, LengthProperty, BisectorProperty);
}
// Backing private fields
private CPoint _a, _b, _midpoint;
private Number _length;
private CLine _bisector;
...
}

```

3.4 Dot Notation for Property Access

By introducing calculated properties, the need for separate property calculation functions was eliminated from SLGeometry. All property values are extracted from objects by way of the Property function, with the following syntax: *Property(object, "propertyname")*. However, actually using this function during expression input would produce cumbersome expressions. Therefore, the dot notation was taken from object-oriented languages and introduced into FLG as an example of language extension (Mernik, 2013; Erdweg, Giarrusso, and Rendel, 2012). In our case, the inclusion of this syntactic feature does not restrict any features from the base language. Furthermore, it is transformed into the functional equivalent during parsing, thanks to appropriate semantic actions (Mössenböck, 2010), which can be applied recursively.

Let us consider the following input example where point $C = Point(2, 3)$ and a segment $D = Segment(C, (0, 0))$ are defined. Table 3.2 shows how dot notation gets translated into its functional equivalent during parsing.

TABLE 3.2: Dot notation transformed into the functional equivalent

Input	Result
$E = C.X$	$E = Property(C, "X")$
$F = D.A.X$	$F = Property(Property(D, "A"), "X")$

Thanks to the fact that object metadata contains descriptions of the object's properties, this feature is a good starting point for the development of code completion in the expression editor. Also, in the GUI, properties of objects can now be "discovered" by the user, since the system can simply show a list of properties from metadata for the selected object.

3.4.1 Introducing Semantic Extensions into FLG

Thanks to metadata, arbitrary C# code can be annotated and imported as a function into SLGeometry, which can significantly expand the domain of application of the DGS. The Table function demonstrates inclusion of a procedural concept (for loop, iterative array building) into FLG. The syntax is *Table(expression, iterator, min, max, step)* where *step* has a default value of 1 and can be omitted. The function returns a list of values obtained by evaluating expression (*expression*) when iterator (*iterator*) takes values from *min* to *max* with the given step. It is assumed that *iterator* occurs in *expression*. The function is implemented in C#

and executes the following procedural algorithm, presented in C#-like pseudo-code (Listing 3.5):

LISTING 3.5: Table algorithm

```
list = { }
for (int i = min; i<=max; i+=step){
    tempExpr = replace iterator with i in expression
    tempResult = evaluate tempExpr
    append tempResult to list
}
return list;
```

For example, the expression $Table(Point(i, i^2), i, 1, 4)$ returns the list of points $\{(1, 1), (2, 4), (3, 9), (4, 16)\}$. For simplicity, we shall use the notation $Point(x, y)$ and (x, y) interchangeably.

The expression argument requires non-standard evaluation order, because the iterator argument is an identifier, which should remain unevaluated and replaced with a concrete integer value in each cycle of the for loop. Thus it is marked with the *ArgNamedSymbolInfo* metadata type (Listing 3.6).

It should be noted that the Table function does not break the functional nature of the FLG, as it returns a single list as a result for each evaluation, with a concrete set of arguments. There are no side-effects, because all procedural code and local variables are confined to the internal implementation of Table. In this regard, we consider the semantics within the Table function as a “semantic box” within the FLG, a term somewhat analogous to the language box (Diekmann and Tratt, 2014), since both are imported as plug-ins.

The metadata specifies two signatures, one without the *step* argument and one with it. If the first signature is used, the argument is assigned the default value of 1, specified in the argument metadata. The example of the Table function demonstrates an extension of the dynamic semantics of the language, which is immersed in the existing syntax without artifacts, that is, we obtain procedural behavior without breaking the functional paradigm of the language.

LISTING 3.6: Metadata specification of arguments for the Table function

```
public class Table : Fn {
    public static readonly ArgNamedInfo ExpressionProperty =
        new ArgNamedInfo("Expression", typeof(GExpression));
    public static readonly ArgNamedInfo IteratorProperty =
        new ArgNamedSymbolInfo("Iterator", ExpressionProperty);
    public static readonly ArgNamedInfo MinProperty =
        new ArgNamedInfo("Min", typeof(Number));
    public static readonly ArgNamedInfo MaxProperty =
        new ArgNamedInfo("Max", typeof(Number));
    public static readonly ArgNamedInfo StepProperty =
        new ArgNamedInfo("Step", typeof(Number), false, new Number(1));
    static Table() {
        Metadata = new FnInfo("Table", typeof(CList),
            new ArgNamedInfo[] {ExpressionProperty, IteratorProperty,
                MinProperty, MaxProperty, StepProperty
            },
            new string[] {"Expression, Iterator, Min, Max",
                "Expression, Iterator, Min, Max, Step"
            });
    }...
}
```

3.4.2 Language Composition Considerations in FLG

Domain-specific languages (DSLs) provide abstraction over the concrete realization of domain concepts, i.e. enable programmers to think about the components and their relations without delving too deep into unnecessary detail (Erdweg, Giarrusso, and Rendel, 2012). Since each application domain provides a separate DSL, there is a need to compose them together if a project covers several domains. Erdweg, Giarrusso, and Rendel (2012) propose a classification of language composition: language extension, language restriction (it is also subsumed in language extension), language unification, self-extension and extension composition.

In language extensions (\triangleleft) a base language is extended or restricted with a new language without been modified (e.g. Java before support for generics and after). Language unification (\oplus) denotes two equally dominant and standalone languages that are connected without modification, using some glue code (e.g. HTML and JavaScript). Self-extension (\leftarrow) does not change the language description, i.e. it is implemented as pure language embedding (e.g. the embedding of XML into Java using JDOM). Extension composition combines all previously described types to produce composition of several language extensions.

Using a simple language for controlling robot movements Mernik (2013) provides a comprehensive discussion of language composition terminology in a case study of his object-oriented language composition framework in the LISA tool. The same Robot language is used by Chodarev et al. (2014) to demonstrate all types of composition using language description tool YAJCo, where language notion definition is separated from the semantics definition.

In our case we presented several examples of language composition:

1. FLG is a self-extension of C#, as expressions can be instantiated and SLGeometry Engine API invoked directly from the C# code. This is not a preferred way of using FLG, however;
2. Dot notation for property access is a syntax extension of FLG, Section 3.4,
3. Semantic language extension of FLG by importing functions (Chapter 4), operations (Section 5.3) and types (Section 5.2). The example of *Table* function demonstrates an extension of the dynamic semantics of the language, where we obtain procedural behavior without breaking the functional paradigm of the language (see Section 3.4.1),

In practical application of SLGeometry, a C# developer may develop the geographic map data type *CMap*, which contains information pertinent to a country on a map, a *Map(countryname)* function, as well as a corresponding visual, *VMap*, which actually draws the map on GeoCanvas. The developer must annotate the classes with metadata and compile them into a plug-in (DLL) file. Let us note that many properties of a country map, such as population, cities, rivers, mountains, roads etc. can be implemented as object constant properties inside a single C# class. This way, the developer captures the information, actions and visuals which represent a certain domain (i.e. a teaching subject).

A teacher (non-programming user) can import the metadata-annotated DLL into SLGeometry and immediately use one or more instances of the geographic map in

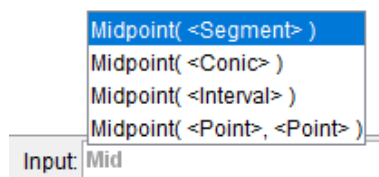


FIGURE 3.1: Parameter value suggestions in the input box in GeoGebra

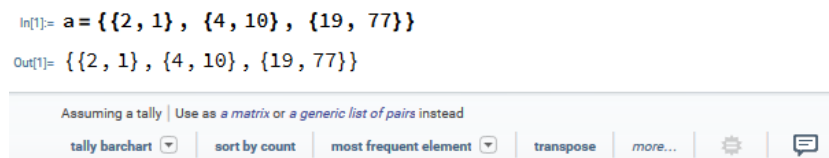


FIGURE 3.2: The Suggestions Bar in Mathematica

dynamic drawings. In general, save for a few of mandatory functions such as `ValueOf`, `Property` and generic operations, `SLGeometry` can be customized by importing new members into τ , or removing some of existing ones.

We refer to the dynamic geometry software, based on functional languages, as 'functional' or 'classical' DGS throughout this dissertation. The classical DGS have certain drawbacks which stem from the nature of their underlying functional languages. Textual input can be difficult. In Steketee (2010) it is argued that using textual input and a functional language to create a geometric drawing can be confusing and error-prone. This is mainly due to the following reasons: the long list of functions is difficult to remember, the command line fails to inform the user of possible choices, and the rigid syntax of commands results in errors. Some of these issues have been addressed in different ways. For example, GeoGebra shows a code completion popup when the user starts typing into the input box (Figure 3.1).

An interesting feature is used in the functional language of Mathematica (Figure 3.2). When a result is displayed, its structure is analyzed, and several functions which can be applied to it are offered in the Suggestions Bar. There is no guarantee that the offered functions are what the user needs, so she can change Mathematica's assumptions by clicking "Use as a matrix or a generic list of pairs instead", and get another list of suggested functions.

A DGS based on an object-oriented language has significant advantages in this regard. The dot syntax is cleaner and free of the clutter caused by parentheses. It is also a natural way of expressing structural relationships in object-oriented languages. The input editor can recognize the type of the object being entered, read its properties from metadata and show them as suggestions. For example, when a user types in an expression that evaluates to a triangle, its properties can be shown in a code-completion popup. This approach also applies to the GUI paradigm. For this approach to work, object constants must have their properties specified in the metadata.

Functional notation can be counterintuitive. Using function composition to perform constructions can be difficult for users, because functions are specified in the

reverse order from the construction steps. Moreover, the functional approach to constructions only reflects how the construction is performed, but does not reveal its meaning or name. Function composition also tends to produce expressions which have many parentheses and are difficult to manage. On the other hand, by using an object-oriented language with dot notation, simpler expressions are obtained.

Let us observe a textbook exercise in Table 3.3: In a triangle ABC , to obtain the foot of the altitude from vertex A , first create a perpendicular from A on the side $[BC]$ and then find the intersection D of the perpendicular with the line through B and C . The solution in a functional language strictly follows the construction steps, albeit in reverse order. The dot notation in an object-oriented language simply leads the user through the properties of a triangle, until the foot D of the altitude from vertex A is found. This approach is simpler to write, and was preferred by students in our tests. Note that the $T.AltitudeA$ is a segment with two endpoints, A and B , which are different from the free points A , B and C . The object-oriented DGS hides the construction steps from the user, but it can still implement exactly the same functions as the functional DGS, and offer the user a choice between using the purely functional approach and the object-oriented one.

TABLE 3.3: Functional vs dot notation in determining the foot of the altitude A

Functional notation	Dot notation
$T = Polygon(A, B, C)$	$T = Triangle(A, B, C)$
$D = Intersect(Line(B, C), PerpendicularLine(A, a))$	$D = T.AltitudeA.B$

For this approach to work, object constants must have a mechanism that calculates all their properties, based on the state of the object. Implementing new shapes and their properties saturates the global context. As the scope of a DGS expands, the developers are adding new data types and functions which operate on them. Numerous properties of many geometric objects, such as triangles (Kimberling, 2019), polygons, curves, conics etc. are of interest to teachers, but implementing them as separate functions quickly saturates the global context. Namely, as the number of distinctly named functions and their type polymorphic implementations increases, the user has to memorize or look up a large number of function names when dealing with objects that have many properties. For a developer, maintaining a large number of functions, some of which target only very specific argument types, can become a tedious task. This problem can be overcome by introducing support for objects with calculated properties into the evaluation engine, and dot notation into the syntax of the language.

3.4.3 Evaluation Optimization

In a functional language, if an expression references a property of an object, it does so by way of a property function. When two distinct references are created, the property function is written and evaluated twice. The redundant evaluation can be avoided by assigning the property function to a variable, as a new expression, and then referencing that variable. In the proposed object-oriented language, property references are tracked by the engine, and referenced properties activated only when needed. The activated properties are calculated only once, regardless of the number of references. Pure functional languages are efficient to evaluate and their expression evaluators can be simple. On the other hand, object support and dot notation provide benefits both to the users and developers, but with the penalty of more complex evaluation algorithms and the possibly increased cost in memory and performance. These points can be summarized as follows:

1. Pure functional approach

- Geometric objects contain the minimal number of mandatory properties;
- Efficient evaluation;
- To obtain additional properties of objects, new specific functions must be implemented, which co-exist in the same namespace and increase the number of names available to users, which can be confusing;
- In the case of two different objects having the same named property, two specific implementations of a function must be created, which only differ in argument types;
- Deep property referencing is possible via function composition, but properties are chained backwards and the syntax produces expressions containing lots of parentheses;
- Dependencies are established by using variable references.

2. Object-oriented approach

- Geometric objects can contain additional properties, besides the mandatory ones;
- Properties are referenced by using the dot syntax, and property names exist only in the domain of objects they belong to;
- Different object types can have same named properties, with separate implementations;
- Property activation must be employed to instantiate only the referenced properties;
- Lazy evaluation must be used;
- Deep property referencing is possible via the dot notation, in the natural order;
- Dependencies are established by using variable and property references.

3.5 Summary

Main motivation for development of the data structure suitable to express non-CLR types in metadata, represented in structural relationships, is to overcome the shortcomings of existing metadata specification mechanisms. This kind of representation of metadata and its structure enable all metadata to be contained within one metadata object, for which correctness is checked in the constructors of metadata classes, while data structures are suitable for expressing complex property dependencies.

In SLGeometry, objects are given with all properties, not only the mandatory one as can be seen in most DGS. In this way, the number of additional functions is reduced to a minimum. This also empowers the dot notation like in object-oriented languages.

Furthermore, it is easy to implement semantic extension into FLG, due to straightforward import of arbitrary C# code using our metadata structure. In addition, all this facilitates the syntax extension of FLG and semantic language extension of FLG by importing functions, operations and types.

Finally, we present on-demand property evaluation in the following way: each time when a set of expressions is changed, there is a need to re-evaluate the expressions; property references are tracked by the engine, and referenced properties are activated only when needed, so lazy property evaluation is implemented. The activated properties are calculated only once, regardless of the number of references. The corresponding algorithm is given in Section 5.4.

Chapter 4

System Architecture Overview

In this chapter we describe our SLGeometry framework for dynamic geometry, the central contribution of this dissertation. Section 4.1 describes the architecture, and includes a brief overview of SLGeometry components. Section 4.2 presents the language description for the functional domain-specific language (FLG). Dynamic drawings in SLGeometry are specified by writing expressions in FLG and assigning them to named variables. Structure $\tau = \{T, C, O, F, V\}$ consists of the set of types (T), type conversions (C), operations (O), functions (F) and visuals (V) in the FLG.

The last section describes expressions, which are made from atomic values, objects, lists and functions.

4.1 SLGeometry Framework

The SLGeometry framework is complex dynamic geometry software, written in C# on the .NET platform. The source code consists of more than 200 .cs files representing more than 16500 lines of code. Implementation details for the classes which create objects on the screen are given in Section 5.1.6. SLGeometry can be used for teaching different subjects besides geometry, i.e. mathematics, such as geography. Shapes on screen (GeoCanvas) are represented by expressions, which can depend on other expressions. The result is interactivity – when, for example, the user moves one point, everything connected to that point also moves. In this way, the user can create and manipulate geometric constructions. In Chapter 6 we show some of our tests with students and pupils. The framework is extensible, therefore software developers can design new functions following the design requirements based on metadata and easily extend the software with new objects.

SLGeometry follows the motivation and ideas given in Chapter 3 and consists of:

- Specifications for Types, Functions and Visual elements;
- CAS Engine;
- Graphical surface;
- Extensibility infrastructure;
- JIT compilation subsystem;
- Expression parser;
- Interactive components.

The main components are: the parser, the expression evaluator (Engine) and the graphical surface (GeoCanvas) (Figure 4.1). The Engine maintains a set of expressions, stored in named variables, which represent the elements of a dynamic drawing. GeoCanvas displays geometric shapes and UI controls, and responds to user interaction.

When the program is started, the Engine initializes the variable storage, name registration storage, operations storage and conversion storage, and scans assemblies and registers the built-in functions using reflection. The name registration storage contains the function factory and function metadata management, i.e. all functions, types, UI controls, constants and constant objects are registered there. Registration of visuals is initiated in GeoCanvas. The visual registration holds forward type mappings, e.g. $CPoint \rightarrow VPoint$ and reverse type mappings, e.g. $VPoint \rightarrow CPoint$. All these classes can be imported and registered via external DLLs.

The user can interact with the system in two ways: by writing text input or using the mouse and moving objects along the screen. Textual input is processed by the parser which uses the expression factory to convert expression trees from `SLGParserTypes` classes into `GExpression` derived objects. We introduced `SLGParserTypes` classes to store the result of parsing, where the name of the variable to be assigned to and its value are kept. Further, it also holds the type of the result, i.e. whether the variable gets the value, a property of an object needs to be assigned, the value is deleted, the property is deleted, or just expression is parsed without the variable name.

All evaluation is placed in the Engine, and the Engine listens to the GeoCanvas events and recalculates all needed properties.

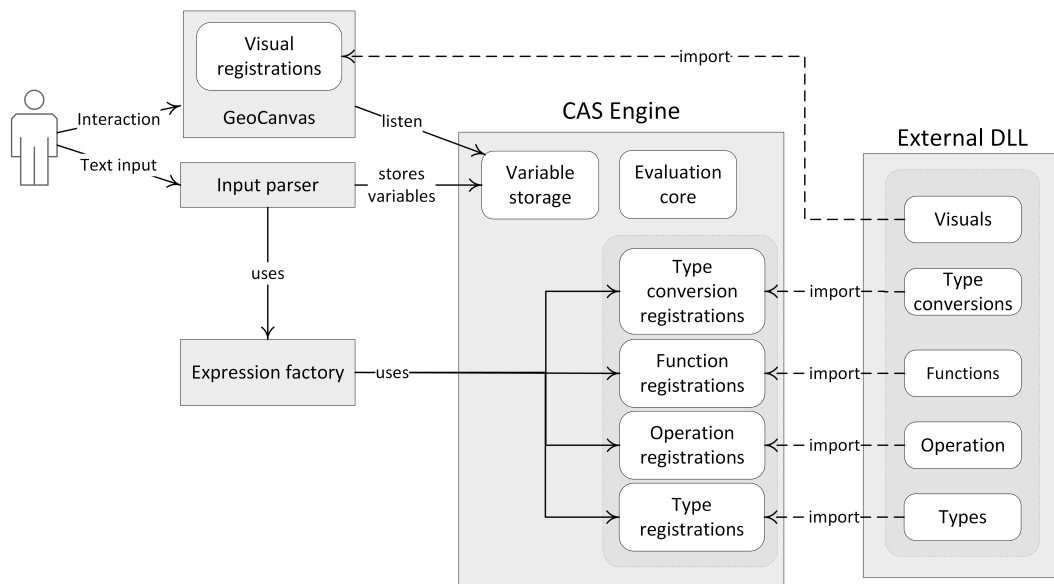


FIGURE 4.1: SLGeometry system architecture overview

Hence, dynamic drawings are created by constructing expressions and assigning them to variables in the Engine. The Engine evaluates the variables. If the evaluation result of a variable corresponds to a visual type registered with the GeoCanvas, the visual is created and shown on the GeoCanvas. In this way, a one-to-one mapping is established between a variable and a visual. Dependencies between variables are

established by using variable references within expressions. A change in one variable triggers recursive recalculation of all dependent variables. Since the variables are bound to visuals on the GeoCanvas, the user can change a variable by interacting with the visual. All dependent variables are then recalculated and their corresponding visuals on the GeoCanvas are updated.

FLG is type-, operation- and function- agnostic, i.e. it only contains generic algorithms, while all types, operations, and functions are imported from metadata-annotated C# classes (Figure 4.1, Figure 4.2). Visuals and type conversions are imported into the GeoCanvas, also as metadata-annotated C# classes. Using reflection, SLGeometry checks for existence of static fields containing metadata in all imported classes, and registers them into τ , i.e. FLG set of types (T), type conversions (C), operations (O), functions (F) and visuals (V) (Figure 4.1). This makes SLGeometry adaptable for various purposes, beyond the domain of computer geometry.

FLG recognizes the following constructs:

1. Constants of simple types, such as numbers, logical values and strings, that appear as literals in expressions;
2. Functions with or without arguments;
3. Variable references, i.e. variable names that appear in expressions. Variable names are replaced with the *ValueOf("name")* function by the parser;
4. Operators for the usual operations: addition, subtraction, multiplication, division and modulus. They are replaced with appropriate functions by the parser;
5. Variable assignment in the form $v = expression$ which, strictly speaking, is not a part of the language. It is interpreted by the Engine and cannot appear within expressions;
6. Object constants with mandatory and calculated properties, which appear as results of function evaluation and cannot be entered manually;
7. Dot notation for property access. It is replaced with the *Property(object, "name")* function by the parser;
8. Property assignment in the form $v.property = expression$ which, strictly speaking, is not a part of the language. It is interpreted by the engine as visual property assignment and cannot appear within expressions.

Compared to other mainstream DGS, 6, 7 and 8 are new features which we introduced in SLGeometry. Visual properties are of particular importance, since they can be bound to FLG expressions, which makes visual properties dynamic.

4.2 FLG Grammar Definition

In this section we present the language description for the FLG, i.e. its syntax and semantics specifications.

Attribute grammars were introduced by Knuth Knuth (1968); Knuth (1971) as a tool for describing and implementing the static semantics of programming languages. An attribute grammar is a context-free grammar, each production of which is augmented with a set of semantic rules. Each semantic rule states how the value of an attribute associated with a syntactic construct in the production is derived by

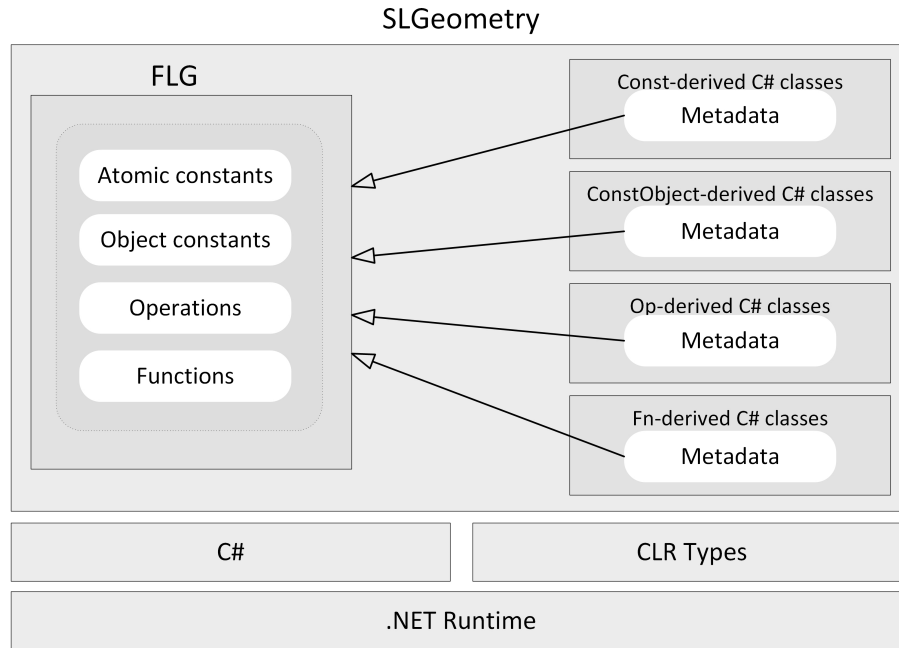


FIGURE 4.2: Mapping of C# implementations to members of τ

applying a semantic function to values of attributes associated with other syntactic constructs in the production.

The production rules are shown in Figures 4.3 - 4.6. The productions of the syntax of the SLGeometry language are described with the compiler description language Cocol/R (Mössenböck, 2010) which is written in Extended Backus-Naur Form (EBNF).

The vocabulary of the language (lexical specification) consists of identifiers, numbers, strings, operators and comments. Blanks, line feeds, carriage returns and tabs are ignored. All statements are terminated with new line. The syntax of a terminal symbol is presented by a regular EBNF expression. Identifiers start with letter or special characters (“\$” and “_”) and contain letters, digits and special characters. Numbers are sequences of digits with usual decimal interpretation and they represent *NumberConstants*. *StringConstants* are sequences of arbitrary characters excluding the quotation mark, or a single quotation mark, enclosed by a pair of quotation marks or single quotation marks. *LogicalConstants* are treated as a predefined enumeration type with elements ‘false’ and ‘true’. Attribute specifications (Knuth, 1968) carry semantic information from leaves up to the root of a derivation tree.

Factor can be *NumberConstants*, *StringConstants*, *LogicalConstants*, *SymbFunc*, a sequence of expressions (*Expression*) separated by comma, enclosed in a pair of parentheses or arguments *Arguments* enclosed in a pair of braces, e.g. $A = true, B = A, C = Point(A, 5 + B)$ (Figure 4.3). *Prop* denotes *Factor* followed by zero, one or more dots with property names. *SymbFunc* represents a function where the identifier refers to the name of the function whose arguments are enclosed in a pair of parentheses.

Pow expressions represents raising the *Prop* to a power (“^”) of another *Prop* (Figure 4.4). *Term* consists of one or more *Pow* which are separated with multiplying operators (“*”, “/”, “%”), e.g. $A/2$. *Expression* (*Expression*) is built from one or more

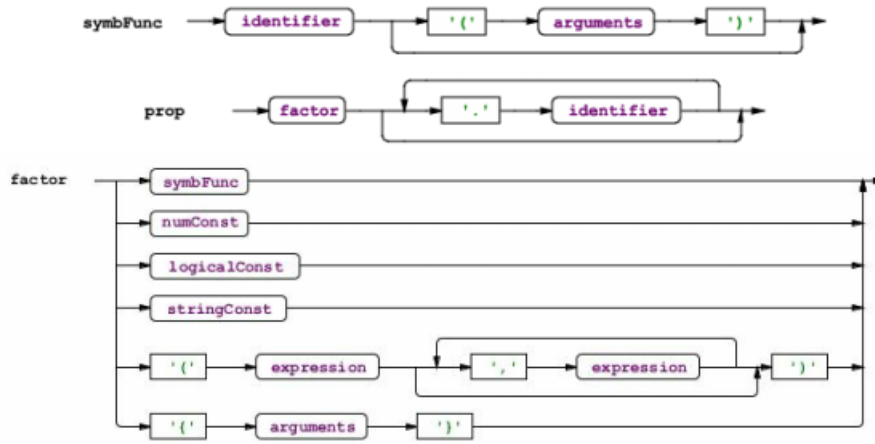


FIGURE 4.3: Syntax diagrams for SymbFunc, Prop and Factor

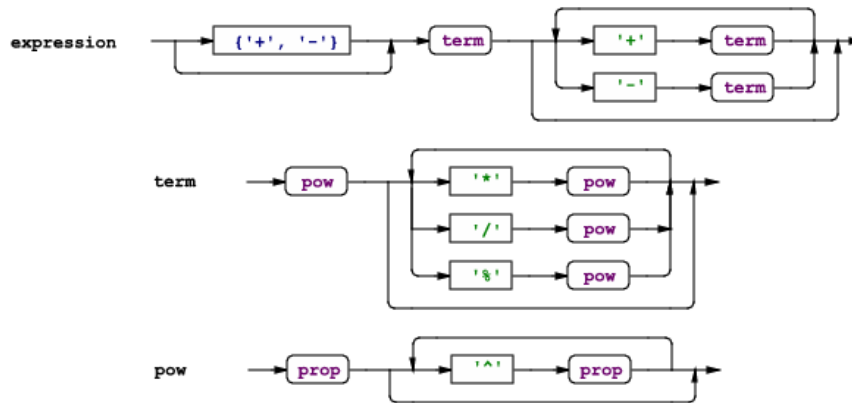


FIGURE 4.4: Productions Expression, Term and Pow

Term, where the first term has an optional sign, and they mutually represent addition ('+') or subtraction ('-'), e.g. $-6 + A$.

RelExp denotes two expressions (*Expression*) separated by relational operators ('<', '<=', '>', '>=', '!=', '==') (e.g. $A < C.X$). It can also consist of only one *Expression*. One or two *AndExp* separated by an or sign ('||'), create *OrExp*, whilst analogously *RelExp* separated by an and sign ('&&') constitute *AndExp* (Figure 4.5). They both form the basis for logical expressions (e.g. $D || E, D \&\& E$). Zero, one or more *OrExp* separated with comma sign (',') represent function arguments (*Arguments*), e.g. $Point(2, 3)$.

The long identifier (*LongIdent*, Figure 4.6) consists of an identifier or two identifiers separated by a dot ('.'). The first case is the variable name (e.g. A), while the other is a variable with its property (e.g. $C.X$). Although properties can be chained (explained in Section 5.4.2) (e.g. $C.X$ yields 2 and $D.A.Y$ yields 3), assignment is only allowed to the properties at the first level (e.g. $C.X = 5$ is allowed, while $D.A.X = 5$ is not). This behavior stems from the principle of operation of DGS (explained in Chapter 1).

The starting input of our language is defined with *Expressions* (Figure 4.6). *Expressions* is an assignment expression in the formal or informal way. Formal assignment is represented by an expression consisting of a left-hand side and a right-hand

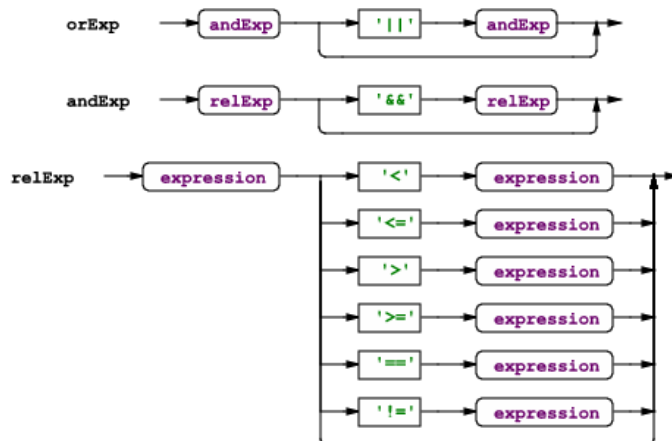


FIGURE 4.5: Productions Or-, And- and Rel-Exp



FIGURE 4.6: Productions Expressions and LongIdent

side which are separated by an equals sign. It can present a special assignment which clears a value of some variable in the form “*longindent = .*” (e.g. $A = .$). Otherwise, the variable is assigned the value of an OrExp (e.g. $B = 3, C = \text{Point}(2,3), D = \text{Segment}(C, (0,0))$). If the expression is not formal assignment, i.e. there is no variable to which the expression is assigned to, SLGeometry automatically transforms it to an informal assignment to the next available name (e.g. $B + 2$ is transformed to $E = B + 2$, because the E is next available letter).

4.3 Expressions

Expressions are built from atomic values, objects, lists and functions. *GExpression* is the base class for all expressions. Simple data types and errors are derived from the *Const* and *CLRConst<T>* classes, while object data types derive from the *ConstObject* class. Functions derive from the *Fn* class. Visual functions, represented by the *VisualFn* class, can have their arguments updated when corresponding visuals are manipulated on screen. All base classes for expressions are shown in Figure 4.7, as well as several classes that contain concrete implementations of simple data types (*Number*, *CString*, *Logical*), object data types (*CPoint*, *CSegment*, *CTriangle*) and functions (*FPoint*, *FSegment*, *FTriangle*).

Data types, conversions, operations, functions and visuals are implemented as C# classes which derive from the supplied base classes, and follow the implementation guidelines described in Chapter 5.

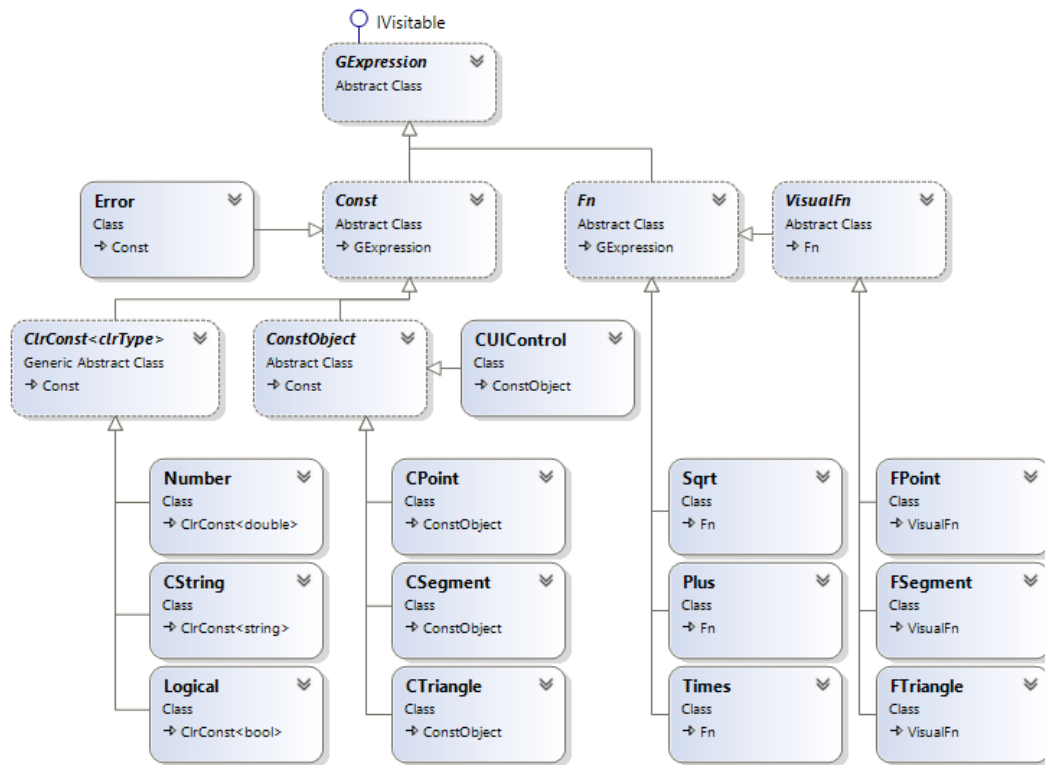


FIGURE 4.7: Expression classes in SLGeometry

4.3.1 Functions and Visual Functions

Functions can have zero or more arguments, and can be overloaded. The return type can be known in advance, as is the case of basic functions (*Abs*, *And*, *Cos*, *Sin*, *Sqrt* etc.), geometric functions (*Line*, *Segment*, *Triangle* etc.) and UI controls. Some functions, such as conditionals, can return results of different types. Results of functions can be constants (*Const*) such as numbers, boolean values and strings, or object constants (*ConstObject*), such as lines, circles and triangles. Functions are implemented by deriving from *Fn* and *VisualFn*. It is possible to have more than one C# class implementing the same function (i.e. with the same name), but their signatures must be distinguishable.

A visual in SLGeometry is implemented with three classes: a function, which generates an object constant; the object constant which represents the actual visual; and a helper class which draws the visual on the *GeoCanvas*. When the function is evaluated, *GeoCanvas* is notified that its result has been changed and the corresponding visual is drawn.

4.3.2 Constants

Constant values are atomic values or objects. Atomic values, such as numbers, boolean values and strings, can be entered directly as literals, or appear as results from functions. Simple data types are derived from the *Const* class. Types which have CLR equivalents, such as *Logical*, *Number* and *CString*, are derived from the *CLRConst<T>* class. Implementing a new atomic data type in SLGeometry is simple: a new C# class needs to be written, which derives from one of the constant base

classes, and the *Clone* method needs to be implemented, which creates a deep copy of the current instance. Standard binary and unary operations for the new data type can also be defined in the same class and provided with metadata, as described in Section 5.1.

4.3.3 Object Constants

Object data types are objects with named properties, which appear as results of function evaluation. Object constants are implemented as C# classes that derive from the *ConstObject* class. Property metadata is initialized in the static class constructor and assigned to the *Metadata* static field of the *ConstObjectInfo* type. Each property is described by an instance of the *PropInfo* class. Properties are either mandatory or calculated. Mandatory properties are necessary for the object to be valid, and are assigned during function evaluation. Calculated properties are obtained by some algorithm from the values of other properties within the object. Properties can also be of object types, thus a single constant value in FLG can actually be represented by a complex tree of C# objects.

Visuals are C# classes which expose public *visual properties* that control their appearance, such as point size, line width, color etc. Each visual contains C# code which draws it on the *GeoCanvas*. Since visual properties are ordinary .NET properties, type conversions between .NET (CLR) types and FLG types must be provided.

4.4 Summary

In this chapter the SLGeometry framework and its architecture are described. As the user communicates with the system through textual input, the language description for the FLG is given in detail, along with constructs which FLG recognizes. Furthermore, the infrastructure of expression classes is explained.

More detailed implementation descriptions are given in Chapter 5, while validations of these implementations, through various examples built in SLGeometry, are given in Part III.

Chapter 5

Implementation Details

In this chapter we present implementation details of metadata-supported object-oriented extension of DGS, based on the papers by Radaković and Herceg (2017) and Radaković and Herceg (2018). In Chapter 3 we have demonstrated the shortcomings of attributes as declarative tags that are used to carry information into run-time, which we used in previous versions of our framework (Radaković and Herceg, 2010; Radaković, Herceg, and Löberbauer, 2010; Herceg and Radaković, 2011; Herceg, Herceg-Mandić, and Radaković, 2012; Herceg, Radaković, and Herceg, 2012; Radaković and Herceg, 2013). Also, we have proposed an object model with calculated properties, instead of the concept which uses property functions that can be numerous. Furthermore, we introduced dot notation for property access, as an example of language extension.

After the first version of our metadata-based extensibility framework was completed and tested in practice, we tried to perceive the bigger picture and identify key components of a DGS which can benefit from becoming generic and extensible.

We decoupled unary and binary operations from the types they operate on and introduced generic operation templates. A generalized binary operation algorithm was developed, which is suitable for matrix and vector calculus, but also applicable to other data types. By leveraging the information available in operation metadata, we were able to implement result caching and early operation binding, which resulted in performance improvements. Also, knowing the result types of functions makes a good starting point for development of a partial expression tree just-in-time compiler.

The most significant result, however, is the support for lazy evaluation of object properties (Radaković and Herceg, 2018). More specifically, the metadata data structures suitable for expressing complex property dependencies in object constants were developed, similar to the idea discussed by Nosál', Sulír, and Juhár (2016).

In next section we give a detailed overview for metadata infrastructure which covers all data types, operations, type conversions and functions we need to annotate to have stable structure. Metadata structure covers metadata for: arguments, signatures, properties, functions, controls and UI controls. At the end of the section, metadata implementation examples are given. Section 5.2 specifies explicit conversion methods between CLR-based constant types and equivalent CLR types. Overview of operations in FLG presented in Section 5.3. One of the main contributions of the dissertation, lazy evaluation with property activation, is described in Section 5.4. Section 5.5 proposes partial compilation of expression trees which are

created by users at run-time. The last section shows how geometrical constructions are saved and loaded in our framework.

5.1 Metadata

All data types, operations, type conversions and functions need to be annotated with metadata and registered with the Engine, and visuals should be registered with the GeoCanvas. The metadata are bound via static fields, defined by convention. The FLG is an annotation-enabled language in the sense given by Nosál', Sulír, and Juhár (2016).

The metadata required by the extensibility infrastructure includes instances of custom classes, as well as dependency graphs in the form of N-ary trees. Visual objects may contain other related objects, thus creating complex structural relationships. For example, a triangle contains as properties special points, segments and lines, such as orthocenter, heights, angles, angle bisectors, circumcenter, circumradius, etc.

Metadata specification is straightforward and intuitive. At the top level, each expression type has an associated metadata class. Functions are described with *FnInfo*, simple constants and objects constants with *ConstInfo* and *ConstObjectInfo* respectively. Visual functions are described with *VisualInfo*, and UI controls with *UIControlInfo*. Function metadata contains signatures: *SignatureEmpty* for functions without arguments, *SignatureUnnamed* for functions with one or more arguments, such as lists, and *SignatureNamed* for functions with a fixed number of named arguments, such as points, lines and circles. Named arguments are represented by the *ArgNamedInfo* class, which contains data such as argument name, type and default value. Unary and binary operations, as well as data type conversions are represented in a similar way. Special attention should be paid to the *PropInfo* metadata class, which describes properties of object constant types. Besides the property name and type, it contains additional information which is used during evaluation and optimization. In order to successfully describe functions and objects, metadata classes shown in Figure 5.1 are used. A more complete overview is given in following sections.

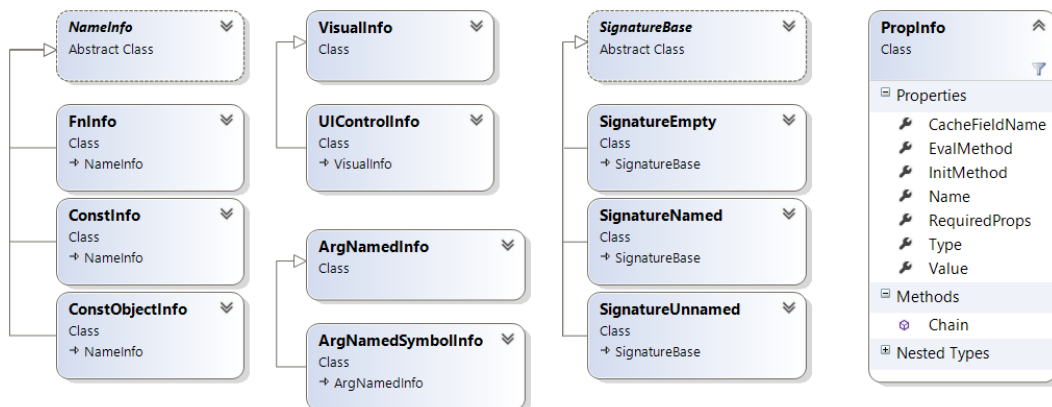


FIGURE 5.1: Constant and function metadata

5.1.1 Metadata for Arguments

Classes *ArgNamedInfo* (Listing 5.1) and *ArgNamedSymbolInfo* are used to describe named function arguments.

Metadata for each function argument carries the following information:

1. Argument name;
2. Argument type;
3. Default value (optional).

Thus, these classes have data for the argument name, argument type and default value. Also, it is controlled whether:

- The argument should be held unevaluated;
- The argument binds directly to a named property of the visual control;
- The argument needed to denote as a symbolic name.

If argument is denoted as a symbolic name, the names of the parameters in which the symbolic name appears is denoted, and *ArgNamedSymbolInfo* is used, e.g. in *Table* function *Iterator* argument is a symbol (see Listing 3.6).

LISTING 5.1: ArgNamedInfo metadata

```
public class ArgNamedInfo{
    public ArgNamedInfo(string argName, Type argType){
        Name = argName;
        ArgumentType = argType;
        IsVisualProperty = false;
        IsSymbol = false;
        DefaultValue = Empty.Value;
    }
    public ArgNamedInfo(string argName, Type propType, bool
        isVisual):this(argName, propType){
        IsVisualProperty = isVisual;
    }
    public ArgNamedInfo(string argName, Type propType, bool isVisual, Const
        defaultValue):this(argName, propType, isVisual){
        DefaultValue = defaultValue;
    }
    ...
    public string Name { get; private set; }
    public Type ContainingType { get; private set; }
    public Type ArgumentType { get; private set; }
    public Const DefaultValue { get; private set; }
    public bool IsHeld { get; private set; }
    public bool IsVisualProperty { get; private set; }
    public bool IsSymbol { get; protected set; }
    ...
}
```

5.1.2 Metadata for Signatures

Classes *SignatureBase* (Listing 5.2), *SignatureEmpty*, *SignatureNamed* and *SignatureUnnamed* present signatures for functions, which contain named arguments in a specific order. Upon instantiation, instances of this class cache *NamedArgInfo* records about named arguments from the Engine. Signatures should be matched against supplied arguments for number and for their type, i.e. it is checked whether the supplied arguments match the given signature. Further, it is checked whether

the given signature conflicts another signature. Additionally, an argument list is created, based on the signature and supplied argument values.

LISTING 5.2: SignatureBase metadata

```
public abstract class SignatureBase{
    public abstract bool Match(IList<GExpression> args);
    public abstract ArgList CreateArgList(FnInfo fn, IList<GExpression> args);
    public abstract bool Conflicts(SignatureBase sig);
}
```

Functions without arguments use signature *SignatureEmpty*, while functions with one or more unnamed arguments use signature *SignatureUnnamed*, e.g. signatures "Expression,Iterator,Min,Max" and "Expression,Iterator,Min,Max,Step" in *Table* function (Listing 3.6). The most functions belong to the group with fixed number of named arguments, such as point, lines, triangles, and they use signature *SignatureNamed* (e.g. functions for circle in Listing 5.13).

Since signatures are written in string format, arguments are split by comma (','). Loaded arguments are checked to match between raw function arguments and the signature in the following way:

1. The number of arguments is compared;
2. The arguments declared with *ArgNamedSymbolInfo* must be of the *TIdent* type and it is marked false, furthermore it should be replaced with a *ConstSymbol*;
3. The arguments are checked for type compatibility, one by one, except the ones marked false;
4. To the argument, which has not been specified in the signature, is assigned the default value.

5.1.3 Metadata for Properties

Metadata for named properties of objects constant types (*ConstObject*) is given in the class *PropInfo* (Listing 5.3). Along with property name, value, type and owner class, properties hold the data about required properties which must be evaluated before this property. Accordingly, a *ReqP* object is created which reference this property and all its required subproperties, e.g. *SideAProperty.Chain(CSegment.MidpointProperty)* returns a *ReqP* object containing: required property *SideAProperty* and subproperty *CSegment.MidpointProperty*, which needs to be calculated first. It also performs type-checking to ensure that the type containing all subproperties is the same as the type of this property. Furthermore, it checks whether the chain of required properties is valid, i.e. whether a property only requires subproperties that it owns.

We introduced this complex structure to accomplish optimization in evaluation. This kind of objects can be very bulky, e.g. triangle with his properties (see Section 7.2). Therefore, we need delegate which calculates the property, the local variable which caches the property value for quick access during evaluation, a mechanism to initialize the property value when it is activated, and to dispose the property value when it is deactivated. A given property is activated when its value needs to be calculated, and it is deactivated when there is no longer need for its value, in order not to burden evaluation of object properties when the object is moving along the *GeoCanvas*. This mechanism is presented in more details in Section 5.4.

The function metadata contains the following information:

1. Property name;
2. Property value;
3. Property type;
4. The type of owner class, i.e. the type of class which owns given property;
5. Holds information about a required property, properties which must be evaluated before this property, and all its required subproperties;
6. Delegate which calculates the property;
7. The name of the local variable which caches the property value for quick access during evaluation;
8. Delegate which is called when the property is activated to initialize the property value, before it can participate in evaluations;
9. Delegate which is called when the property is deactivated, to dispose of the property value.

LISTING 5.3: PropInfo metadata

```

public class PropInfo{
    public string Name { get; private set; }
    public Const Value { get; set; }
    public Type Type { get; private set; }
    public Type OwnerType { get; set; }
    public ReqP[] RequiredProps { get; private set; }
    public PropInfo(string name, Type type, Type ownerType){
        Name = name;
        Type = type;
        OwnerType = ownerType;
    }
    public PropInfo(string name, Type type, Type ownerType, params ReqP[]
        requiredProps):this(name, type, ownerType){
        if ((requiredProps != null) && (requiredProps.Length > 0)){
            if (requiredProps.Where(p => p == null).Count() == 0){
                RequiredProps = requiredProps;
            }else
                throw new ArgumentNullException("requiredProps",
                    string.Format("PropInfo constructor for {0}.{1}: One or more
                    null values encountered. Check the order of property
                    registrations in the metadata.", type.FullName, name));
            foreach (ReqP r in RequiredProps)
                if (!r.CheckChain())
                    throw new ArgumentException("requiredProps",
                        string.Format("PropInfo constructor for {0}.{1}: Types not
                        equal. Check the types required properties of property
                        registrations in the metadata.", type.FullName, name));
        }
    }
    ...
    public ReqP Chain(params PropInfo[] subProps){
        if (subProps.Length == 0)
            return new ReqP(this);
        else{
            ReqP me = new ReqP(this);
            me.SubProps = new ReqP[subProps.Length];
            for (int i = 0; i < subProps.Length; i++)
                me.SubProps[i] = new ReqP(subProps[i]);
            return me;
        }
    }
    public Action<GExpression> EvalMethod { get; set; }
    public string CacheFieldName { get; set; }
    public Action<GExpression> InitMethod { get; set; }
    public Action<GExpression> DisposeMethod { get; set; }
    public static implicit operator ReqP(PropInfo p){
        return new ReqP(p);
    }
    public class ReqP{
        public PropInfo Info;
        public ReqP[] SubProps;
        ...
        public bool CheckChain(){...}
    }
}

```

5.1.4 Metadata for Functions and Controls

5.1.4.1 NameInfo

Abstract class *NameInfo* (Listing 5.4) holds information about a Fn-derived type or a Control-derived type registration in the Engine. The name is used in the parser

as a function name, which creates instances of the type. The unique name, used to distinguish between different classes implementing multiple variants of a function. Registration types for specific types should inherit from this class. *Assembly* denotes the assembly this function was imported from, while it is null for built-in functions.

LISTING 5.4: NameInfo metadata

```
public abstract class NameInfo{
    public Type Type { get; protected internal set; }
    public string Assembly{ get; set; }

    public string Name { get; protected internal set; }
    public string UniqueName { get; protected internal set; }
}
```

5.1.4.2 FnInfo

FnInfo describes a function type for use in the parser and the evaluation Engine (Listing 5.5). A dictionary *ArgDescriptors* keeps all names of the named arguments belonging to the function together with the metadata for function argument. All arguments are checked if they exist in *ArgDescriptors*. The type of the result of this function is kept in *ResultType*.

Signatures determines a collection of signatures for the Fn-derived function. The matching signature is returned, or null, if no match is found. *IsFlat* determines whether this function can be flattened, i.e. $f(f(f(x, a), b), c) \rightarrow f(x, a, b, c)$.

The function metadata contains the following information:

1. Function name;
2. Result type;
3. An array of arguments;
4. An array of function signatures.

LISTING 5.5: FnInfo metadata

```
public class FnInfo : NameInfo{
    protected FnInfo(){
        Signatures = new List<SignatureBase>();
    }
    public FnInfo(string name, Type resultType, IList<ArgNamedInfo> arguments,
        string[] signatures, bool isFlat = false){
        this.Name = name;
        this.ResultType = resultType;
        this.IsFlat = isFlat;

        this.ArgDescriptors = new Dictionary<string,
            ArgNamedInfo>(arguments.Count);
        foreach (ArgNamedInfo ai in arguments)
            ArgDescriptors.Add(ai.Name, ai);

        this.Signatures = new List<SignatureBase>();
        foreach (string sig in signatures)
            if (CheckArguments(sig)){
                SignatureNamed sn = new SignatureNamed(this, sig);
                Signatures.Add(sn);
            }
    }
}
```

```

public FnInfo(string name, Type resultType, Type argType, int minArgs, int
    maxArgs, bool isFlat = false)
    : this(name, argType, minArgs, maxArgs, isFlat){
    this.ResultType = resultType;
}
public FnInfo(string name, Type argType, int minArgs, int maxArgs, bool
    isFlat = false){
    this.Name = name;
    this.IsFlat = isFlat;
    SignatureUnnamed sig = new SignatureUnnamed(argType, minArgs, maxArgs);
    this.Signatures = new List<SignatureBase>();
    this.Signatures.Add(sig);
}

private bool CheckArguments(string args){
    string[] argArray = args.Split(',');
    foreach (string a in argArray)
        if (!ArgDescriptors.ContainsKey(a.Trim()))
            return false;

    return true;
}
public Dictionary<string, ArgNamedInfo> ArgDescriptors = new
    Dictionary<string, ArgNamedInfo>();
public Type ResultType{ get; set; }
public bool IsFlat{ get; set; }
public List<SignatureBase> Signatures { get; internal set; }

public SignatureBase GetMatchingSignature(ICollection<GExpression> args){
    foreach (SignatureBase sig in Signatures){
        if (sig.Match(args))

            return sig;
    }
    return null;
}...
}

```

5.1.4.3 ConstInfo and ConstObjectInfo

ConstInfo and *ConstObjectInfo* (Listing 5.6) are metadata classes for constants and constants objects. Unlike *ConstInfo*, *ConstObjectInfo* contain a list of properties. Usually, constant objects are result of UI controls functions, e.g. *CCircle* in Listing 5.12.

The metadata for the constant object contains the following information:

1. Constant object type;
2. A list of properties, each described with an instance of the *PropInfo* metadata class.

LISTING 5.6: ConstObjectInfo metadata

```

public class ConstObjectInfo: NameInfo{
    public Dictionary<string, PropInfo> Properties { get; private set; }

    public ConstObjectInfo(Type constType, params PropInfo[] properties)
    {
        if (!constType.GetTypeInfo().IsSubclassOf(typeof(ConstObject)))
            throw new ArgumentException(string.Format("{0} does not
                inherit ConstObject"), constType.FullName);

        Assembly = constType.AssemblyQualifiedName;
        Type = constType;
    }
}

```

```

        Name = UniqueName = constType.Name;

        Properties = new Dictionary<string, PropInfo>();
        foreach (PropInfo pi in properties)
            Properties.Add(pi.Name, pi);
    }
}

```

5.1.5 Metadata for User Interactive Controls

Metadata class *VisualInfo* (Listing 5.8) contain: a constant which corresponds to a type *VisualBase*-derived class in the *GeoCanvas*, a type of visual, a belonging assembly, dictionaries with visual properties and dependency properties. At initialization following visual properties: visibility, label, label offset, and weather the object is selected.

LISTING 5.7: UIControlInfo metadata

```

public class VisualInfo{
    public Type ConstType { get; private set; }
    public Type VisualType { get; protected internal set; }
    public Assembly Assembly { get; set; }
    public Dictionary<string, ArgNamedInfo> VisualProperties { get; set; }
    public Dictionary<string, PropertyInfo> DependencyProperties { get; set; }
    public VisualInfo(Type constType){
        this.ConstType = constType;
        this.VisualProperties = new Dictionary<string, ArgNamedInfo>();
        VisualProperties.Add("$Visible", new ArgNamedInfo("$Visible", null,
            typeof(Logical), true, new Logical(true)));
        VisualProperties.Add("$Label", new ArgNamedInfo("$Label", null,
            typeof(CString), true, new CString()));
        VisualProperties.Add("$LabelOffset", new ArgNamedInfo("$LabelOffset",
            null, typeof(CPoint), true, new CPoint()));
        VisualProperties.Add("$Selected", new ArgNamedInfo("$Selected", null,
            typeof(Logical), true, new Logical(false)));
    }
    public VisualInfo(Type constType, ArgNamedInfo[] visualProperties):
        this(constType){
        foreach (ArgNamedInfo ai in visualProperties)
            VisualProperties.Add(ai.Name, ai);
    }
}

```

Metadata class *UIControlInfo* (Listing 5.7) contains metadata for UI controls such as control type, and its dependency properties. All dependency properties, which have CLR property wrappers, are discovered and registered, while dependency properties without CLR property wrappers are ignored. Also, *ArgNamedInfo* metadata are created for all discovered properties and added to the *ArgDescriptors* list.

LISTING 5.8: VisualInfo metadata

```

public class UIControlInfo: VisualInfo{
    public Type ControlType { get; set; }
    public UIControlInfo(Type controlType, Conversion.Conv conv):
        base(typeof(VUIControl)){
        ControlType = controlType;
        RegisterDependencyProperties(conv);
    }
    private void RegisterDependencyProperties(Conversion.Conv conv){
        DependencyProperties = new Dictionary<string, PropertyInfo>();
    }
}

```

```

VisualProperties.Add("$Location", new ArgNamedInfo("$Location",
    this.VisualType, typeof(CPoint), true, new CPoint()));
VisualProperties.Add("$LocationMode", new ArgNamedInfo("$LocationMode",
    this.VisualType, typeof(Number), true, 0.N()));
VisualProperties.Add("$Rotation", new ArgNamedInfo("$Rotation",
    this.VisualType, typeof(Number), true, 0.N()));
VisualProperties.Add("$RotationCenter", new
    ArgNamedInfo("$RotationCenter", this.VisualType, typeof(CPoint),
    true, new CPoint()));
DiscoverProps(ControlType, conv);
}
private void DiscoverProps(Type t, Conversion.Conv conv){
    IEnumerable<PropertyInfo> propInfos = t.GetProperties(BindingFlags.Public
        | BindingFlags.Instance);
    foreach (PropertyInfo pi in propInfos)
        if (conv.IsCLRTypeSupported(pi.PropertyType))
            DependencyProperties.Add(pi.Name, pi);
    var result0 = from p in propInfos
        where conv.IsCLRTypeSupported(p.PropertyType)
        select new ArgNamedInfo(p.Name, t,
            conv.GetTypeForCLRType(p.PropertyType), true);
    List<ArgNamedInfo> final = result0.ToList();
    final.ForEach(argInfo => VisualProperties.Add(argInfo.Name, argInfo));
}
}
}

```

5.1.6 Implementation Examples

To represent one geometric object on the screen we need to implement tree different C# classes which are structurally connected via metadata. These classes represent function, constant object and visual representation. Each geometric object has a unique *ConstObject* class that represents a result of a function. Its value is a light structure of a type struct representing lightweight objects (Listing 5.9). The reason why we decided to choose struct type for this value is because it might be more efficient. For example, as the drawing can have several hundreds of objects, the additional memory needs to be allocated for referencing each object; in this case, a struct would be less expensive.

One constant can be the result of several C# classes which implement the polymorphic function with the same name for the parser. For example, a line can be defined with three Number parameters a, b, c like $ax + by + c = 0$, or with two Point parameters A, B as a line through two points. These functions differ in the number and names of arguments, i.e. the signatures are not the same, therefore they are recognized properly by the parser. A definition of how the object is drawn on the screen is contained in its visual function. The visual function contains data about the shape, color, width and how to update screen location. This principle is shown on example in Section 5.1.6.2.

LISTING 5.9: Struct representing a line

```

public struct SLine{
    public double a { get; set; }
    public double b { get; set; }
    public double c { get; set; }
    public bool IsValid { get; private set; }
    public SLine(double a, double b, double c): this(){
        Assign(a, b, c);
    }
}

```



```

public void Assign(double a, double b, double c){
    this.a = a;
    this.b = b;
    this.c = c;
    IsValid = (a != 0) || (b != 0);
}
}

```

The following examples demonstrate how metadata is specified for a basic function (*Sqrt*, *Plus*), geometric functions (*FCircle*, *FCircle3*), constant objects (*CCircle*) and visual objects (*VCircle*).

5.1.6.1 A Simple Function

The square root function (*Sqrt*), given in Listing 5.10, is an example of a simple function with one unnamed argument. The metadata contains the name of the function, the type of the result, argument type and minimal and maximal number of arguments. The signature for this function is inferred from the metadata automatically by the Engine.

This function is called in input as ‘*Sqrt*’. It expect *Number* as argument, and results *Number*, also. Minimum and maximum number of arguments is equal, i.e. it is only one.

LISTING 5.10: Metadata for the Sqrt function

```

public class Sqrt : Fn {
    public static FnInfo Metadata;
    static Sqrt(){
        Metadata = new FnInfo("Sqrt", typeof(Number), typeof(Number), 1, 1);
    }...
}

```

Plus function has at least 2 arguments, where upper bound is limited with integer maximum value. Differently from the *Sqrt* function, the *Plus* function accepts arguments which are not only limited to *Number*. It can be also e.g. *Matrix*, but all arguments must have the same type.

LISTING 5.11: Metadata for the Plus function

```

public class Plus : Fn{
    public static FnInfo Metadata;
    static Plus(){
        Metadata = new FnInfo("Plus", typeof(GExpression),
            typeof(GExpression), 2, int.MaxValue);
    }...
}

```

5.1.6.2 Visual Functions and Objects

Let us consider a circle, as it demonstrates some important features shared by many geometric objects. The object constant class of type *CCircle* (Listing 5.12) contains relevant geometric properties of a circle, such as center and radius.

LISTING 5.12: Metadata for the CCircle object constant type

```

public class CCircle : ConstObject {
    public static ConstObjectInfo Metadata;
    public static readonly PropInfo SProperty =

```

```

        new PropInfo("S", typeof(CPoint));
public static readonly PropInfo RProperty =
        new PropInfo("r", typeof(Number));
public static readonly PropInfo RadiusSegmentProperty =
        new PropInfo("RadiusSegment", typeof(CSegment));
static CCircle() {
    Metadata = new ConstObjectInfo(typeof(CCircle), SProperty, RProperty,
        CenterProperty, RadiusProperty, RadiusSegmentProperty);
}...
}

```

There are two C# classes which implement the polymorphic function named "Circle" (Listing 5.13), one defined by center S and radius r (*FCircle*) and the other defined by three points A, B, C (*FCircle3*). The signatures inferred from the metadata differ in the number and names of arguments, therefore they are distinguished by the parser and the correct function can be instantiated. The *FCircle* and *FCircle3* functions both return a *CCircle* constant. Additional functions can be added in a similar way, as long as signatures are different.

LISTING 5.13: Metadata registration for the *FCircle* and *FCircle3* functions

```

public class FCircle : VisualFn{
    public static FnInfo Metadata;
    public static readonly ArgNamedInfo SArgument = new ArgNamedInfo("S",
        typeof(CPoint), false, new CPoint(new SPoint(0, 0)));
    public static readonly ArgNamedInfo RArgument = new ArgNamedInfo("r",
        typeof(Number));
    static FCircle(){
        Metadata = new FnInfo ("Circle", typeof(CCircle), new ArgNamedInfo[] {
            SArgument, RArgument }, new string[] { "S,r", "r" });
    }...
}
public class FCircle3 : VisualFn{
    public static FnInfo Metadata;
    public static readonly ArgNamedInfo AArgument = new ArgNamedInfo("A",
        typeof(CPoint)); // BArgument, CArgument omitted for brevity
    static FCircle3(){
        Metadata = new FnInfo("Circle", typeof(CCircle), new ArgNamedInfo[]
            {AArgument, BArgument, CArgument }, new string[] { "A,B,C" });
    }...
}

```

The *VCircle* helper class (Listing 5.14) contains code that actually draws a circle on the *GeoCanvas*. Its metadata, stored in a *VisualInfo* object, specifies the association with the *CCircle* type. The metadata also defines visual properties, such as line style and width, which affect the appearance of the circle.

LISTING 5.14: Metadata for the *VCircle* helper class

```

public class VCircle : VisualBase{
    public static VisualInfo Metadata; // StyleProperty omitted for brevity
    public static readonly ArgNamedInfo WidthProperty = new
        ArgNamedInfo("Width", typeof(Number), true, 1.N());
    static VCircle(){
        Metadata = new VisualInfo(typeof(CCircle), new ArgNamedInfo[] {
            StyleProperty, WidthProperty });
    }...
}

```

5.2 Type Conversions

For CLR-based constant types, explicit conversion methods must be implemented, to enable conversions to and from equivalent CLR types. The conversion metadata is assigned to the static `Conversions` field. It is represented by the `ConvInfo` class, which specifies the data type it applies to (FLG source type), the default conversion method to an equivalent CLR type, an array of conversion methods to other CLR types (if any) and an array of conversion methods from CLR types. An example for the `Number` data type is shown in Listing 5.15. For simplicity, lambdas are used for some conversions. Other conversions are defined as static methods (Listing 5.16).

LISTING 5.15: Conversion metadata for the `Number` data type

```
Conversions =
new ConvInfo(typeof(Number),           // applies to Number
new ToCLR(typeof(double), n => ((Number)n).Value), // default ToCLR
new ToCLR[] {
    new ToCLR(typeof(int), ToInt32),       // other ToCLR conversions
    new ToCLR(typeof(long), ToInt64),
    new ToCLR(typeof(string), ToString)
},
new FromCLR[] {                          // FromCLR conversions
    new FromCLR(typeof(double), typeof(Number), n => new Number((double)n)),
    new FromCLR(typeof(int), typeof(Number), n => new Number((int)n)),
    new FromCLR(typeof(long), typeof(Number), n => new Number((long)n))
}
);
```

LISTING 5.16: Explicit conversion methods for the `Number` data type

```
public static object ToInt32(Const c){
    double d = ((Number)c).Value;
    if (double.IsInfinity(d) || double.IsNaN(d) || (d > int.MaxValue) || (d
        < int.MinValue))
        return null;
    else
        return (int)d;
}
public static object ToInt64(Const c){
    Number n = (Number)c;
    if (n.Value > long.MaxValue)
        return long.MaxValue;
    else if (n.Value < long.MinValue)
        return long.MinValue;
    else
        return (long)n.Value;
}
public static object ToString(Const c){
    return ((Number)c).ToString();
}
```

5.3 Operations

Operations in FLG are generic and extensible. Standard built-in unary and binary operators act as templates, which only execute general algorithms. The actual code that performs calculations for each data type or a pair of types is implemented separately and registered in the Engine. Standard binary operations, addition, subtraction, multiplication, division, exponentiation and unary negation are specified by appropriate operators in textual form. In expression trees, operators are translated

into template functions, such as *Plus* and *Times*. When a template function is evaluated, it finds the appropriate implementation to execute, based on the operand types.

It is preferable that the code for new operations is placed in the class that implements the related constant type. In order to be recognized by the Engine, operations metadata (Figure 5.2) must be grouped in the *OperInfo* collection and stored within the static *Operations* field in the class. This step is done in the static class constructor. Unary and binary operation metadata are stored in the *UnaryOp* and *BinaryOp* classes respectively. The *UnOpDelegate* and *BinOpDelegate* reference static methods which perform the actual operation on the concrete operand(s). The *ResultInitDelegate*, if specified, is used to create a single instance of the result type, which is then cached and reused for performance.

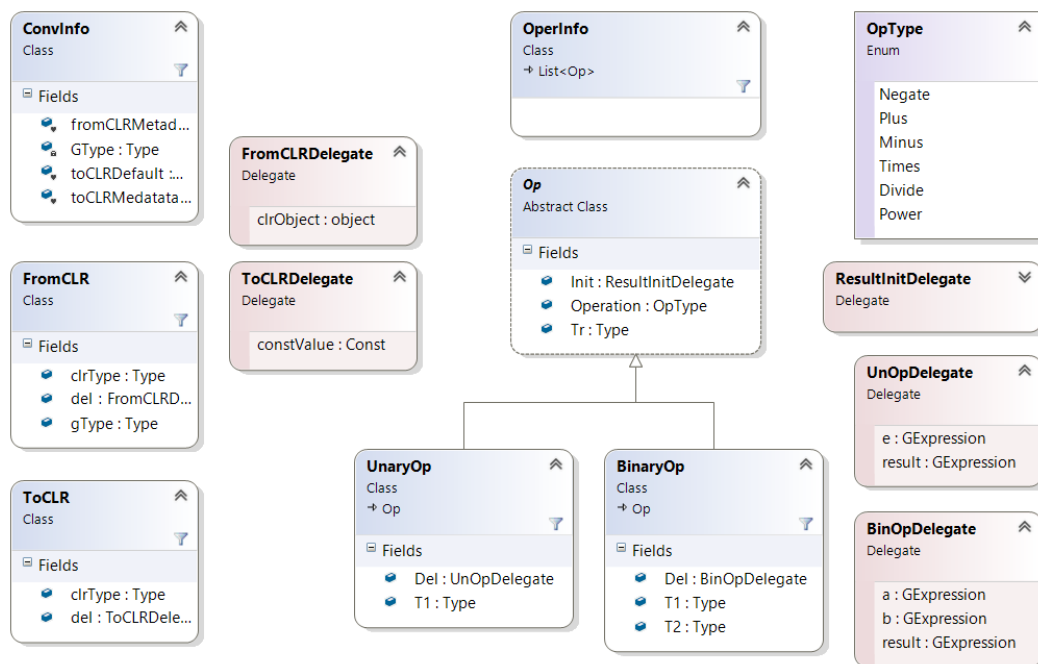


FIGURE 5.2: Metadata for binary and unary operations

5.3.1 Result Caching

The usual way operations are executed is given by the following algorithm:

1. Evaluate operand(s);
2. Check operand(s) for errors;
3. If an error is found, pass it up the expression tree as the result;
4. Create a new instance of the result data type;
5. Perform the actual operation and store the result in the result object;
6. Return the result object.

Since the expressions in a DGS are repeatedly evaluated many times per second, it makes sense to avoid unnecessary object instantiation in the heap. The result constant can be created beforehand by the *ResultInitDelegate* and cached. For that

reason, step 4 is moved out of the evaluation cycle and invoked once after the expression tree is constructed, and before evaluation starts. Whenever the operation is executed, the cached result object is passed to the *UnOpDelegate* and *BinOpDelegate* to be updated. In special cases, the implementers may choose to bypass caching and keep creating new instances of the result on each operation evaluation. In that case, the *ResultInitDelegate* should be assigned null in the metadata and the operation calculation methods should ignore the result parameter.

5.3.2 Operation Examples

An example of operation implementations for one unary operation (negation) and one binary operation (addition) in the Number data type is shown in Listing 5.17. The metadata is created in the static class constructor, and assigned to the static *Operations* field. The *NumberInit* method is used to create result objects.

Metadata for a unary and a binary operation contains the following information:

1. Operation type, used to bind with the actual operator;
2. Operand type (unary), i.e. operand types (binary);
3. Delegate which performs result initialization;
4. Delegate which performs the operation.

LISTING 5.17: Operations for the Number data type

```
public class Number : ClrConst<double>{
    public static OperInfo Operations;
    public static ConvInfo Conversions;
    static Number(){
        BinaryOp addition = new BinaryOp(OpType.Plus, typeof(Number),
            typeof(Number), typeof(Number), NumberInit, NumberPlus);
        UnaryOp negation = new UnaryOp(OpType.Negate,
            typeof(Number), NumberInit, NumberNegate);
        Operations = new OperInfo(addition, negation);
    }...// Conversions omitted for brevity
    private static GExpression NumberInit(){
        return new Number();
    }
    private static GExpression NumberPlus(GExpression c1, GExpression c2,
        GExpression result){
        ((Number)result).Value = ((Number)c1).Value + ((Number)c2).Value;
        return result;
    }
    private static GExpression NumberNegate(GExpression c1, GExpression result){
        ((Number)result).Value = -((Number)c1).Value;
        return result;
    }...
}
```

5.3.3 Late vs. Early Operation Binding

The *Fn* class contains three properties, which describe the return type of the function, based on function metadata in derived classes (Table 5.1). *GeoCanvas* uses these properties to determine whether the result of the function is a visual object or a list of visual objects, which should be drawn on screen. Optimization algorithms can make use of these properties to determine whether parts of the expression tree can be more efficiently evaluated, or to discover type incompatibilities. Execution

of unary and binary operations also depends on these properties. When operand types are known beforehand, early operation binding can be employed. In that case, the correct calculation method is found immediately after the expression tree is constructed, and the result constant is instantiated in advance. Late operation binding is employed when operand types are unknown until execution. The default late bound binary operation (Listing 5.18) is used as fallback, if no early bound operations can be found.

TABLE 5.1: Function result type metadata in the Fn class

Property	Value
DefiniteResultType	Guaranteed return type of the function. If the function returns different types, <i>typeof(GExpression)</i> .
FirstResultType	Same as DefiniteResultType, if the function has a single return type. If the function returns different types, the type of the first result (or any result). If return value is a list, <i>typeof(CList)</i> .
AtomicResultType	Same as FirstResultType, if the function returns atomic values. If the function returns a list, AtomicResultType of the first element.

LISTING 5.18: The default late bound binary operation

```
public static GExpression BinaryOpLateBinding(OpType opt, GExpression a1,
    GExpression a2){
    Type t1 = a1.GetType(); Type t2 = a2.GetType();
    BinaryOp bop = Engine.Oper.GetBinaryOp(opt, t1, t2);
    if (bop == null)
        return Error.IncompatibleTypes;
    else{
        GExpression result2 = bop.Init();
        return bop.Del(a1, a2, result2);
    }
}
```

5.3.4 Generalized Binary Operation Algorithm

Binary operations can be applied to operands which are either atomic, lists of equal lengths or mixed (Table 5.2). This behavior is suitable for vector and matrix operations. Let us note that, by applying the operation recursively in cases 2, 3 and 4, we get the generalized binary operation algorithm. The lazy bound variant of the generalized binary operation algorithm is presented in Listing 5.19.

TABLE 5.2: Binary operations applied to atomic values and lists

Case	Argument a	Argument b	Result of $a \circ b$
1	Atomic	Atomic	$a \circ b$
2	Atomic	$\{b_1, b_2, \dots, b_m\}$	$\{a \circ b_1, a \circ b_2, \dots, a \circ b_m\}$
3	$\{a_1, a_2, \dots, a_n\}$	Atomic	$\{a_1 \circ b, a_2 \circ b, \dots, a_n \circ b\}$
4	$\{a_1, a_2, \dots, a_n\}$	$\{b_1, b_2, \dots, b_m\}$	$\{a_1 \circ b_1, a_2 \circ b_2, \dots, a_n \circ b_m\}$ if $n = m$; Error otherwise

LISTING 5.19: The generalized late bound binary operation

```

public GExpression BinOpRec(OpType opType, GExpression arg1, GExpression arg2){
    if (arg1 is CList) {
        CList l1 = (CList)arg1;
        if (arg2 is CList) { // list + list
            CList l2 = (CList)arg2;
            if (l1.Elements.Length == l2.Elements.Length) {
                int len = l1.Elements.Length;
                Const[] tempRes = new Const[len];
                for (int k = 0; k < len; k++) {
                    tempRes[k] = (Const)BinOpRec(opType, l1[k], l2[k]);
                    if (tempRes[k] is Error)
                        return tempRes[k];
                }
                return new CList(tempRes);
            } else
                return Error.ListsNotEqualLengths;
        } else { // list + atomic
            int len = l1.Elements.Length;
            Const[] tempRes = new Const[len];
            for (int k = 0; k < len; k++) {
                tempRes[k] = (Const)BinOpRec(opType, l1[k], arg2);
                if (tempRes[k] is Error)
                    return tempRes[k];
            }
            return new CList(tempRes);
        }
    } else {
        if (arg2 is CList) { // atomic + list
            // omitted for brevity, analogous to the previous case
        } else { // atomic + atomic
            return (Const)Engine.BinaryOpLateBinding(opType, arg1, arg2, Oper);
        }
    }
}

```

5.4 Lazy Evaluation with Property Activation

Objects with a large number of calculated properties may require significant time to be updated. Furthermore, since their properties can also be objects, this creates the possibility of infinite N-ary trees in the heap. To mitigate this problem, we developed a property activation algorithm, which follows the call-by-need paradigm as presented by Sinot (2008), and evaluates only those properties which are referenced by other expressions, and if so, evaluate them only once. The property activation infrastructure in the Engine tracks property references in expressions, activates properties which are referenced, and deactivates them when references are removed. A property can also be activated if its value is required to calculate another activated property within the same object constant.

5.4.1 Implementation Requirements

There are several requirements for implementing constant object types, which participate in property activation:

1. Each property must be backed by a private field of the same type;
2. Metadata for a mandatory property must contain property name, type and the name of the backing private field;

3. Metadata for a calculated property must contain a delegate to a static evaluator method which calculates the property;
4. If a calculated property requires values of other calculated properties within the same object, those dependencies must be stated in the metadata;
5. If a calculated property requires the value of a subproperty of one of its properties (deep property dependency), the whole chain of properties must be stated in the metadata.

These requirements and guidelines are demonstrated on the example of the *CTriangle* data type in FLG, which is implemented as the *CTriangle* class in C# (Listing 5.20). The *CTriangle* object constant is returned from the *Triangle(A, B, C)* function, which first evaluates its arguments and then assigns the results to the mandatory properties A, B and C of the *CTriangle*. The *CTriangle* defines calculated properties such as *SideA*, *AltitudeAFoot*, *AltitudeA*, *Perimeter* and *MedianA*. The *SideA* calculated property depends only on mandatory properties A, B and C, therefore no additional dependencies are specified. Analogous properties for sides B and C also exist, as well as many other properties which are omitted from this example (Kimberling, 2019). Examples of dependent properties are the *AltitudeAFoot* property, which depends on *SideA*, and the *AltitudeA* property, which depends on both *SideA* and *AltitudeAFoot*.

LISTING 5.20: Metadata for the *CTriangle* object data type

```
public class CTriangle : ConstObject{
    public static ConstObjectInfo Metadata;
    // Mandatory properties (BProperty, CProperty omitted for brevity)
    public static readonly PropInfo AProperty = new PropInfo("A",
        typeof(CPoint), "_a");
    // Calculated properties
    public static readonly PropInfo SideAProperty =
        new PropInfo("SideA", typeof(CSegment), CalcSideA, "_aside");
    public static readonly PropInfo AltitudeAFootProperty =
        new PropInfo("AltitudeAFoot", typeof(CPoint), CalcAltitudeAFoot,
            "_altitudeafoot", SideAProperty);
    public static readonly PropInfo AltitudeAProperty =
        new PropInfo("AltitudeA", typeof(CSegment), CalcAltitudeA, "_altitudea",
            SideAProperty, AltitudeAFootProperty);
    // Deep property dependency
    public static readonly PropInfo MedianAProperty =
        new PropInfo("MedianA", typeof(CSegment), CalcMedianA, "_mediana",
            SideAProperty.Chain(CSegment.MidpointProperty));
    static CTriangle(){
        Metadata = new ConstObjectInfo(typeof(CTriangle), AProperty, BProperty,
            CProperty, SideAProperty, SideBProperty, SideCProperty,
            MedianAProperty, AltitudeAProperty, AltitudeAFootProperty);
    }...
}
```

5.4.2 Structural Correspondence

We can observe the structural correspondence between the concrete objects in the language and the metadata. In particular, deep property dependency is specified in the metadata for the *MedianA* property, which requires the *Midpoint* property of the *SideA* property (Figure 5.3). By invoking *SideAProperty.Chain(CSegment.MidpointProperty)* in the metadata specification,

a chained activation is specified, which states that the `CTriangle.MedianA` property requires the `CTriangle.SideA.Midpoint` property in order to be calculated. Since the required property is a subproperty of the `CTriangle.SideA` property, it is also marked for activation.

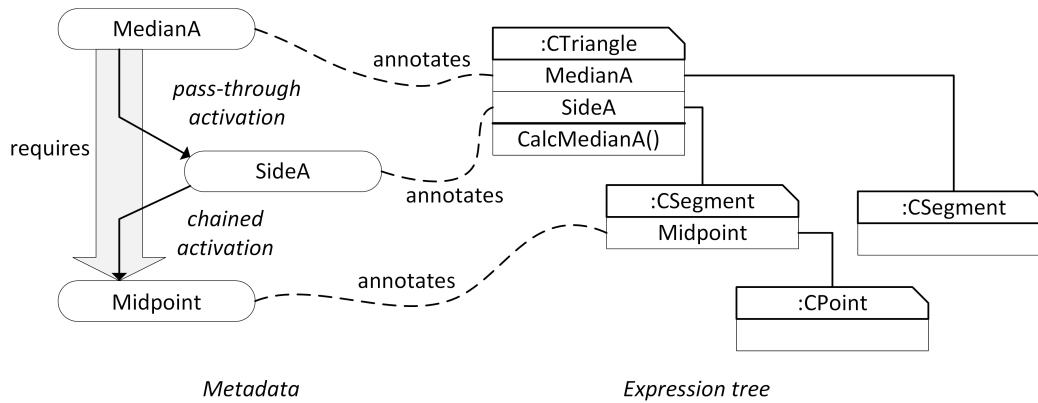


FIGURE 5.3: Chained activation for the `CTriangle.MedianA` property

It is worth noting that property dependencies can form more complex dependency graphs. For example, in Figure 5.4 the dependency graph for the `CTriangle.Circumcircle` property is shown. Explicit property requirements are marked with solid lines, while mandatory properties `CTriangle.A`, `CTriangle.B` and `CTriangle.C` are always updated and do not need to be activated.

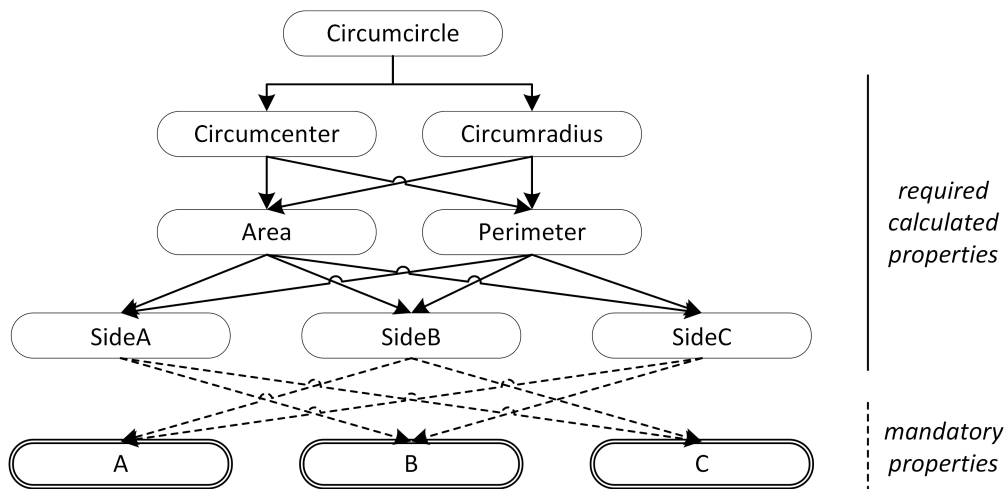


FIGURE 5.4: Dependency tree for the `CTriangle.Circumcircle` property

5.4.3 Backing Fields and Accessors

The `ConstObject` class provides access to properties via indexers: property names or property metadata can be used as indexes. From experience, we found it convenient to provide get accessors for all significant property values. Backing fields can also be used to access property values from within the class and they are automatically initialized by the property activation infrastructure. The developer needs only to declare them in the class and register them in the metadata (Listing 5.21).

LISTING 5.21: Backing fields and accessors for CTriangle

```

public class CTriangle : ConstObject{
    ...// Backing fields for properties
    private CPoint _a, _b, _c, _altitudefoota;
    private CSegment _aside, _bside, _cside, _mediana, _altitudea;
    // Get accessors for the commonly used properties
    public SPoint A { get { return _a.Value; } }
    public SPoint B { get { return _b.Value; } }
    public SPoint C { get { return _c.Value; } }
    ...
}

```

5.4.4 Property Evaluators

A property evaluator is implemented as a static method which accepts a reference to the instance of the object and subsequently calculates and updates the designated property. In the example in Listing 5.22, several property evaluators for the *CTriangle* constant object type are shown. Let us note that backing fields are used to quickly access property values, avoiding indexers which are slower. Also, the GMath helper class is used to perform actual geometric calculations.

LISTING 5.22: Property evaluators for the CTriangle object data type

```

public class CTriangle : ConstObject{
    ...// Property evaluators
    private static void CalcSideA(GExpression e){
        CTriangle ct = (CTriangle)e;
        ct._aside.Assign(ct.B, ct.C);
    }
    private static void CalcAltitudeAFoot(GExpression e){
        CTriangle ct = (CTriangle)e;
        ct._altitudeafoot.Assign(GMath.AltitudeFoot(ct.A, ct._aside));
    }
    private static void CalcMedianA(GExpression e){
        CTriangle ct = (CTriangle)e;
        CSegment aside = (CSegment)ct._aside;
        CPoint midpoint = (CPoint)aside[CSegment.MidpointProperty];
        ct._mediana.Assign(ct.A, midpoint.Value);
    }
}

```

5.4.5 Instance Initialization and Updating

The lifetime of object constants is divided between two phases: initialization and repeated updating. An object constant is usually initialized when a function is created, and then updated each time the function is evaluated. The initialization is conducted by invoking the `InitializeProperties()` method from the instance constructor (Listing 5.23). By convention, updating is done in the `Assign` method, which is implemented independently for each object type. The mandatory fields are updated first, followed by the activated calculated fields. By analyzing a dependency graph inferred from the property metadata, the system takes care of invoking property evaluators in a correct sequence.

LISTING 5.23: Initialization and updating of the CTriangle object instance

```

public class CTriangle : ConstObject{

```

```

...
public CTriangle() {
    InitializeProperties(); // Property initialization
}
// Assign is used to update the object state
public void Assign(Point pt1, Point pt2, Point pt3){
    _a.Assign(pt1);
    _b.Assign(pt2);
    _c.Assign(pt3);
    // Calculate activated properties
    for (int i = 0; i < propEvals.Count; i++)
        propEvals[i](this);
}...
}

```

5.5 Partial Compilation of Expression Trees

Expression trees are created by users at run-time. Expressions are either typed in text, or created implicitly by placing geometric objects on the graphic display. Since FLG is dynamically typed, a function must perform type checks on all of its arguments before each evaluation. This occurs on each node in an expression tree. The structure of the expression tree does not change between evaluations, and evaluations occur many times per second, due to the interactive nature of DGS. It is thus reasonable to assume that, in most cases, repeated type checks are unnecessary and can be avoided, or at least moved out of the evaluation cycle, resulting in a decrease in evaluation time.

The other point to be addressed is range checking. In specific cases, depending on the functions and allowed argument ranges, it could be avoided altogether. In general, however, range checking is necessary. Should an intermediate value fall out of the allowed range, the error should occur and propagate up the expression tree. Calculation errors of all kinds should stop evaluation and propagate to the root of the expression tree immediately.

Furthermore, functions can accept arguments of different types, and provide type specific implementations. This is a form of polymorphism in FLG. Since, in general, the types of arguments are not known beforehand, i.e. until evaluation, static binding is impossible. However, there are cases when data types of a function's arguments and result can be known beforehand.

Functions in FLG are implemented as C# classes which derive from the `Fn` base class. The `Fn.Eval` method must be overridden in the derived classes. It provides the implementation of the function's logic. The `Fn` class provides argument handling and the metadata infrastructure. The function metadata contains static information on the allowed function signatures, argument types and the expected result type, which is returned from the `Eval` method. Should a function's return type metadata be unspecified, special fields contain the runtime type information. Polymorphic functions contain several branches – evaluation paths – in the `Eval` method, which process particular combinations of argument tuples and types. Branch commands are executed upon each invocation of the `Eval` method in such functions.

Example 1. Let us observe a simple expression $b = \text{Sin}(a)$. Metadata for the `Sin` function specifies that its argument, as well as the result, are of type `Number`.

It is thus obvious that variable b must hold a value of the same type. If the type of variable a could unequivocally be determined beforehand, then type checking could be omitted altogether during the evaluation cycle.

Example 2. Let us observe the expression $c = \text{Sqrt}(\text{If}(b > 0, b, \text{"negative"}))$. Metadata for the *Sqrt* function specify that its argument, as well as the result, are of type *Number*. Therefore, the result of the evaluation depends on the return type of the *If* function, which cannot be determined beforehand.

Figure 5.5 presents several compilation cases.

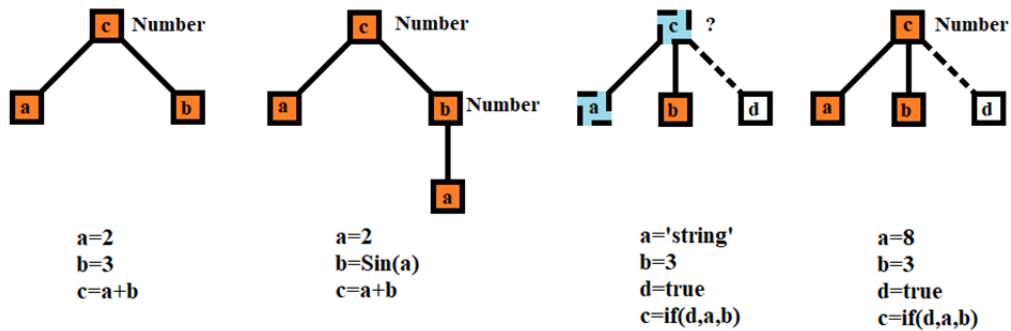


FIGURE 5.5: Result types of various expressions

5.5.1 Partial Compilation Implementation

Uncompiled expression trees are evaluated by recursively invoking the *Eval* method on each node. The compilation process transforms parts of the expression tree by replacing them with compiled code. There are three approaches we used.

- *ExpressionTree* with the simulated stack (ES);
- *ExpressionTree* with variables (EV);
- *Delegates* with lambdas (DL).

The first two make use of the *ExpressionTree* classes in the .NET Framework, which produce the Common Intermediate Language (CIL) code. The aim with the first two approaches was to generate the most efficient code possible. However, this approach requires substantial knowledge of the intricacies of the CIL bytecode. The third approach makes use of lambda expressions to implement each evaluation path separately.

In all three approaches, the expression node classes are extended with properties and methods which enable partial compilation (Figure 5.6).

Compiled CIL code, obtained using the first two approaches, has the structure shown in Figure 5.7. First, the input parameters are accepted. Next, the evaluation code is executed, followed by result checking. Execution then jumps either to error handling code or to the 'finish' label, where the result is saved and subsequently returned to the caller.

Upon creation, the expression tree is scanned for compilable subtrees. Each function node determines, based on the metadata, arguments and supplied compilation

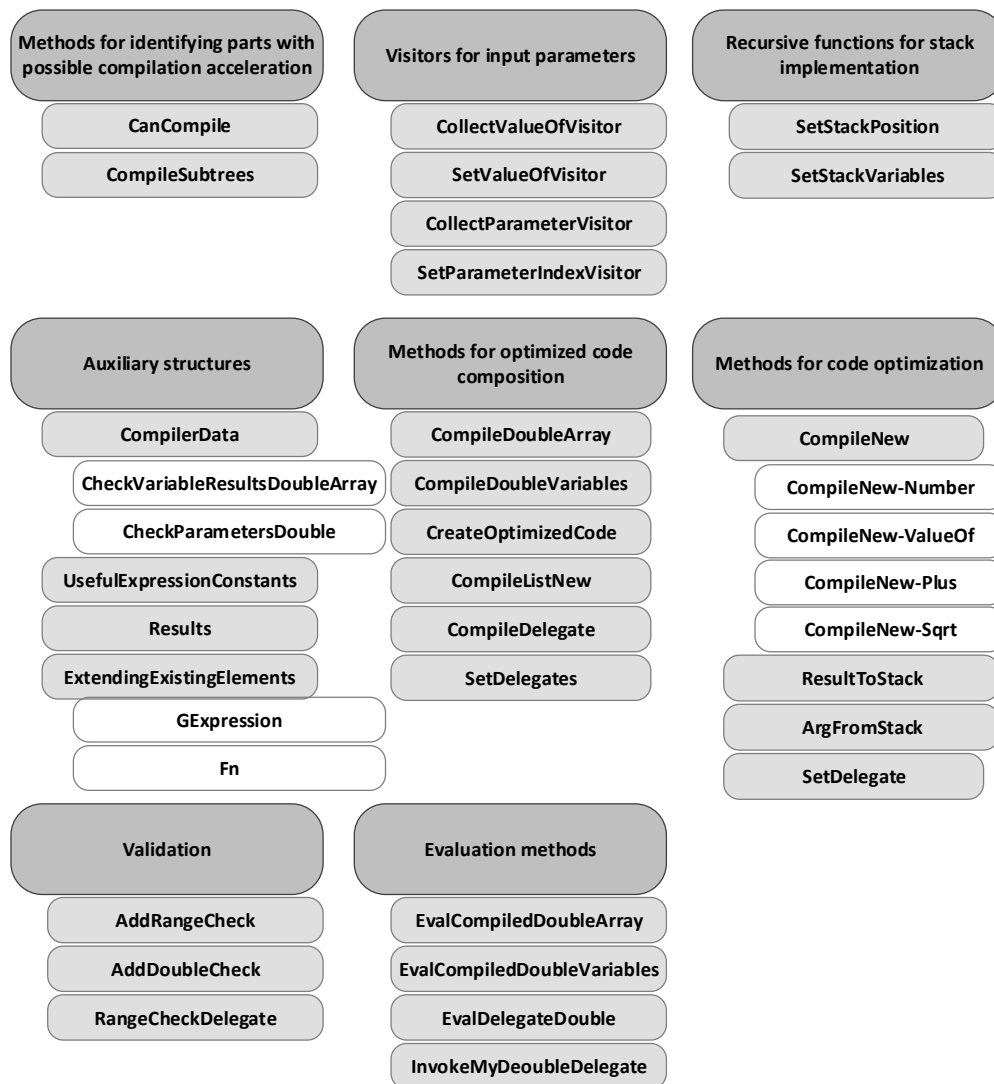


FIGURE 5.6: A subset of the compilation infrastructure

extensions, whether it can be compiled. A general rule is that a numeric function with numeric arguments must be compilable. This requirement is easily extended to any supported data type. Based on the chosen compilation approach, the Eval method of the topmost node is then replaced with the compiled code. Some general rules for compilation:

- Numeric constants are always compilable;
- ValueOf is compilable if its DefiniteResultType is numeric;
- Arithmetic operations are compilable if their arguments are numeric;
- Simple mathematical functions are compilable if their arguments are numeric.

Listing 5.24 shows the implementations of the CanCompile method for various node types.

LISTING 5.24: Implementation of CanCompile method for different superclasses of GExpression

```
//Number
public override bool CanCompile{get { return true; } }
```

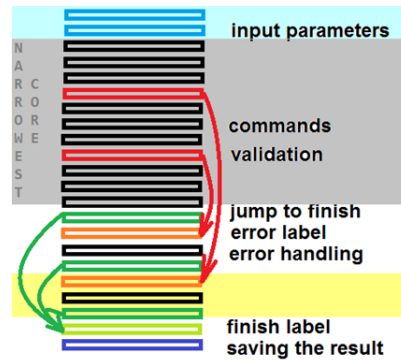


FIGURE 5.7: Compiled CIL code

```

//ValueOf
public override bool CanCompile{get{ return Var.Expr.DefiniteResultType ==
    typeof(Number);}}
//Plus
public override bool CanCompile{ get { return ((Args.Count == 2)
    && (Args[0].Expression.DefiniteResultType == typeof(Number))
    && (Args[1].Expression.DefiniteResultType == typeof(Number)));
    }
}
//Sqrt
public override bool CanCompile{get { return
    Args[0].Expression.DefiniteResultType == typeof(Number);}}

```

The third approach, with lambdas, provides a good balance between implementation complexity and computational performance. Listings 5.25 and 5.26 show the Compile methods for the Plus and Sqrt functions.

LISTING 5.25: Implementation of CompileNew method for the Plus function

```

public override void CompileNew(Compiler c, CompileType type){
    Args[0].Expression.CompileNew(c, type);
    if (CheckRequired(type)){
        c.AddRangeCheck(Args[0], type);
    }
    Args[1].Expression.CompileNew(c, type);
    if (CheckRequired(type)){
        c.AddRangeCheck(Args[1], type);
    }
    c.AddExpression(Expression.MakeBinary(
        System.Linq.Expressions.ExpressionType.AddAssign,
        c.ArgFromStack(Args[0], type),
        c.ArgFromStack(Args[1], type)));
}

```

LISTING 5.26: Implementation of CompileNew method for the Sqrt function

```

public override void CompileNew(Compiler c, CompileType type){
    arg.Expression.CompileNew(c, type);
    if (CheckRequired(type)){
        c.AddRangeCheck(arg, type);
    }
    c.AddExpression(c.ResultToStack(this,
        Expression.Call(sqrtMethod, c.ArgFromStack(arg, type), type));
}

```

In functions which are possible to compile, the number of arguments is always known. Let us consider the *Plus* function, depicted in Figure 5.8. Execution flow is composed recursively, in several steps:

- Calculation of the first argument and pushing the value on the stack;
- Validation of calculated value (optional);
- Calculation of the second argument and pushing the value on the stack;
- Validation of calculated value (optional);
- Calculation of the function value, using the arguments from stack, and pushing the calculated value on the stack.

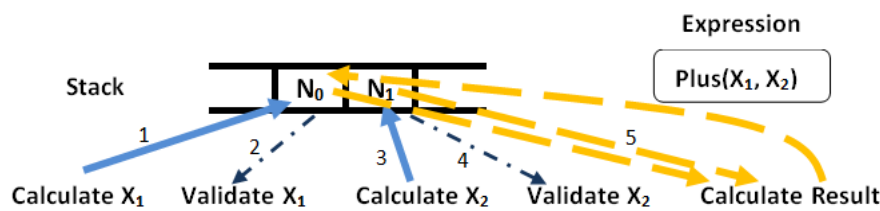


FIGURE 5.8: Execution flow illustration of CompileNew method in Plus function

5.6 XML-Based Serialization of Geometrical Constructions

We can say that all DGS use some formal languages for describing geometrical objects, e.g. GeoGebra .ggb, where is hidden XML format. XML is widely used, because it is strict and has verification mechanisms, along with a large number of supporting tools (Vidaković and Racković, 2006; Dimić and Surla, 2009; Janičić, 2010). Therefore, we have chosen xml-based format for representation of geometrical constructions. Listing 5.27 represents a point $A = (2,3)$ shown on the GeoCanvas.

Converting from a DGS language to xml, would be performed by a specific converter which serializes variables from the engine into XML. Converting from xml to a FLG language is more complicated, and Engine is needed to deserialize given XML file. All expressions are loaded in proper order, maintaining the dependencies. Each function is mapped with its name, unique name and class that represent it.

LISTING 5.27: XML representation of the drawing with one point

```
<Root>
<Assemblies>
  <Assembly Index="1" Name="SLGCore, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
</Assemblies>
<TypeMappings>
  <Mapping Name="$Grid" UniqueName="$Grid" Class="SLGeometry.Shapes.FGrid" />
  <Mapping Name="Logical" UniqueName="Logical" Class="SLGeometry.Expressions.Logical" />
  <Mapping Name="CString" UniqueName="CString" Class="SLGeometry.Expressions.CString" />
  <Mapping Name="CPoint" UniqueName="CPoint" Class="SLGeometry.Shapes.CPoint" />
  <Mapping Name="Number" UniqueName="Number" Class="SLGeometry.Expressions.Number" />
  <Mapping Name="Point" UniqueName="Point" Class="SLGeometry.Shapes.FPoint" />
</TypeMappings>
<Variables>
  <Var Name="$Grid">
    <Fn UniqueName="$Grid">
      <Args />
    </Fn>
    ...//Visual properties
  </Var>
  <Var Name="A">
    <Fn UniqueName="Point">
```

```

    <NamedArgs>
      <Arg Name="X">
        <Number Value="2" />
      </Arg>
      <Arg Name="Y">
        <Number Value="3" />
      </Arg>
    </NamedArgs>
  </Fn>
  <VisualProperties>
    <NamedArgs>
      <Arg Name="$Visible">
        <Logical>True</Logical>
      </Arg>
      <Arg Name="$Label">
        <CString></CString>
      </Arg>
      <Arg Name="$LabelOffset">
        <CPoint>
          <X>
            <Number Value="0" />
          </X>
          <Y>
            <Number Value="0" />
          </Y>
        </CPoint>
      </Arg>
      <Arg Name="$Selected">
        <Logical>False</Logical>
      </Arg>
      <Arg Name="Shape">
        <Number Value="0" />
      </Arg>
      <Arg Name="Size">
        <Number Value="8" />
      </Arg>
    </NamedArgs>
  </VisualProperties>
</Var>
</Variables>
</Root>

```

5.7 Summary

In this chapter we have described the main contributions of the dissertation: a metadata-supported object-oriented extension of DGS with a lazy evaluation mechanism. Our initial focus has been on expressing metadata, as an alternative for .NET attributes, that are structurally designed, easy to use and accessible via reflection. Metadata defined in this way allow the use of non-CLR types values.

Further, this metadata structure directly contributes to an extensibility mechanism, such as plug-ins. This mechanism enables adding new types, operations, functions and visuals. Structural relationships are correctly represented, and correctness is checked in the constructors of metadata classes.

Finally, the property activation infrastructure, using lazy evaluation and optimization, reducing the evaluation costs, is devised. Geometric constructions are kept in XML format.

Part III

Validation

Chapter 6

Subject-Specific Components in DGS

Computer-aided visualization provides meaningful insight into teaching, which helps improve comprehension. With dynamic geometry software, a teacher can easily create interactive mathematical learning materials. To fully benefit from DGS applied to subjects other than mathematics, their functionality must be extended with features which are inherent to those subjects. We demonstrate how a DGS, extended with components, can be applied to geography as well as mathematics teaching. Through experiments, we confirmed the benefits of our approach in practice.

SLGeometry can import and use software components from DLL files. These components are either interactive visual controls (UI controls) or sequential behavior controllers. UI controls represent objects such as buttons, light bulbs, clocks, geographic maps, etc. Behavior controllers contain control logic, which is employed to control the behavior of interactive drawings. We implemented a set of components, which facilitate development of interactive materials, especially mathematical games.

In this section we present a pattern for building mathematical games in SLGeometry, and describe the features of our components, which help teachers develop games without programming. We tested our approach using our experimental DGS platform SLGeometry (Herceg, Radaković, and Herceg, 2012; Radaković and Herceg, 2013; Radaković and Herceg, 2018; Herceg et al., 2019) in two separate experiments. The first experiment had two stages:

1. Graduate students of mathematics were required to create dynamic drawings which contained geographic maps, to be used as learning materials in a high school classroom;
2. High school pupils were assigned tasks based on the materials prepared in the first experiment.

One of the tasks was performed twice, first using a DGS without components and then using a DGS with components. The second experiment involve use of sequential behavior controllers, which make the development of games without programming possible. The students' and pupils' performance, reactions and comments were recorded and compared.

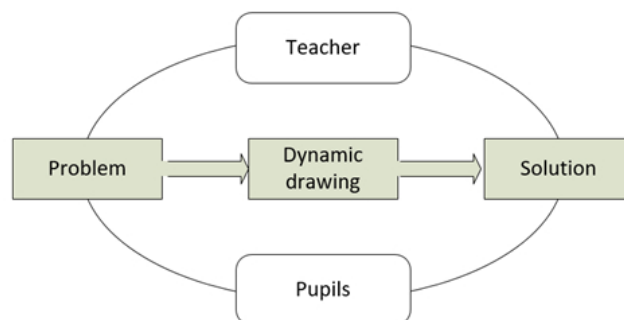


FIGURE 6.1: Straightforward path to the solution

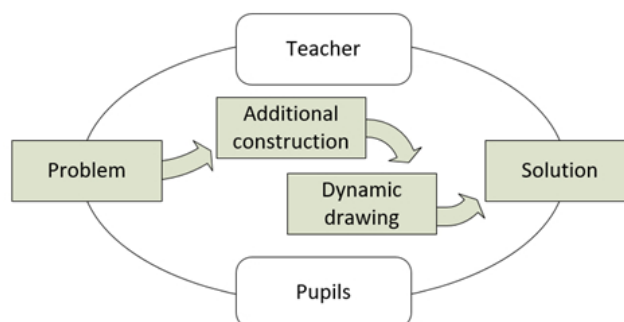


FIGURE 6.2: Deviating from the original problem to construct auxiliary visuals

6.1 Motivation

Creation of mathematical learning materials in DGS is easy and straightforward. Due to their interactive nature, DGS are also used to implement learning materials for other subjects (Blum and Niss, 1991; Herceg and Herceg-Mandić, 2013). Many authors, for example Sarama and Clements (2009) stress that geometry is the most relevant mathematical subject which lies at the heart of physics, chemistry, biology, geology, geography, art and architecture. However, DGS have certain shortcomings when applied to subjects other than mathematics, as they are designed to operate primarily with numerical data, geometric objects and mathematical functions. Setting up a DGS to show interactive drawings for subjects other than mathematics can be a tedious and time consuming task. For example, to create an interactive drawing which contains a geographical map, the author must temporarily set aside his primary goal and focus on the mathematical construction of the map. Furthermore, when a finished map is included in the dynamic drawing, it consists of many objects which can draw the pupils' attention away from the main focus of the drawing. The same limitation also appears when creating materials for other subjects. Thus, one may find that creation of learning materials for subjects other than mathematics, while feasible, is not easily achieved in a DGS, except where examples are essentially mathematical.

In the ideal case, teacher and pupils consider a mathematical problem, create a dynamic drawing to visualize the problem and develop a solution from there (Figure 6.1). For other subjects, the original problem must be set aside while the auxiliary

construction is performed, and only when this is finished, can the original task be resumed (Figure 6.2). In the classroom, when time is limited, this can pose a significant problem.

The aforementioned problem can be solved by implementing subject-specific components and their corresponding functions in a DGS. The visual appearance and behavior of each component is programmed in advance, so that authors are freed of the unnecessary work and complexity is removed from the dynamic drawings. The pupils can regard the components as they would their real life counterparts: for example, a map in real life corresponds to a map component on the screen.

6.2 Components in DGS

A software component is a self-contained unit, which is developed independently and can be loaded and executed in the host software. Components in SLGeometry are either functions or visual objects, which are added to the existing functions and visual objects. When considering the use of components, one should have in mind several important points.

Without components:

1. One complex object is constructed from many geometric shapes;
2. Many variables required to hold these shapes;
3. No logical connection between the shapes;
4. Construction can be completed in DGS alone;
5. Duplicating the complex object requires duplication of all its constituent parts.

With components:

1. One complex object is represented by one component;
2. One variable is required to hold the component;
3. One-to-one correspondence between the component and its real life counterpart;
4. Component is developed by an experienced developer, outside the DGS;
5. Duplicating the complex object creates only one additional variable.

As one can see, the components have many positive aspects. However, there is one significant drawback – each component must be purposefully developed in C# by an experienced developer, outside of SLGeometry. This means that the majority of ordinary users do not possess the necessary skills for component development. On the other hand, given the proper instructions and guidelines, a reasonably skilled developer could make a number of components and make them available for download, for example via a dedicated Web site or from within the DGS.

Simplicity is probably the most important property of DGS, which has made them popular among teachers. Components bring simplicity back into DGS, when they are used in subjects other than mathematics. Considering the subject of geography, there is a pronounced need for maps of all kinds. A map component can draw a country map, while other components can draw locations of cities, rivers and other geographical objects.



FIGURE 6.3: Mainland Italy, drawn by the Country component

Components may extend the reach of DGS even more. For example, by adding interfaces to educational hardware, such as Arduino and micro:bit, the concept of the "generic organizer", as defined by Tall (1986), can be extended into the realm of tangible tools. Learning materials based on dynamic geometry software complemented with educational hardware can be a valuable asset in "blended learning", as defined by Kashefi, Ismail, and Yusof (2012).

6.3 The Country and City Components

The Country function returns a list of points, which represent geographic coordinates of the border of a country. When a polygon is created from the list, a visual representation of the country appears on the screen. Country has two arguments, the country name and the part index – useful for countries which consist of more than one continuous territory (Listing 6.1). The City function returns geographic location of the city (Listing 6.2). It also has two arguments, the first being the city name and the second index, useful when there are multiple cities with the same name.

LISTING 6.1: Country component - Italy

```
italy = Country("Italy", 0)
italy.$Visible = false
pItaly = Polygon(italy)
```

The Country function returns a list of points on the Italian border. The points are then made invisible, and a polygon is drawn. The resulting image on the screen (Figure 6.3) represents mainland Italy, shown at correct geographic coordinates.

6.3.1 The Experiments

For the experiments, we chose the topic 'Age of discovery', which is an important part of the high school geography curriculum. Graduate students of mathematics were required to create dynamic drawings which contained geographic maps, and high school pupils were assigned three tasks based on the learning materials:

1. Measuring the length of a river,
2. Tracing the voyages of the great explorers, Ferdinand Magellan and James Cook,
3. Tracing the students' /pupils' own travels.

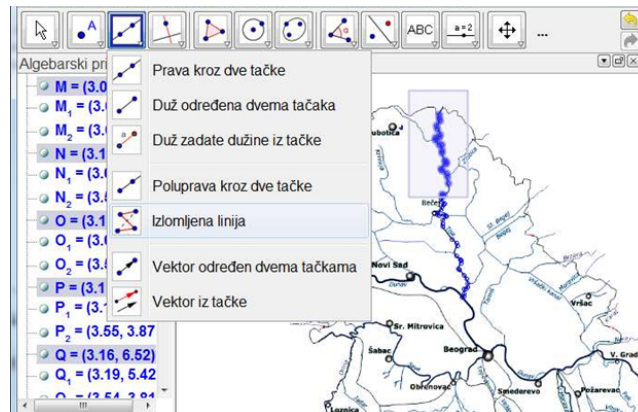


FIGURE 6.4: Measuring the length of a river

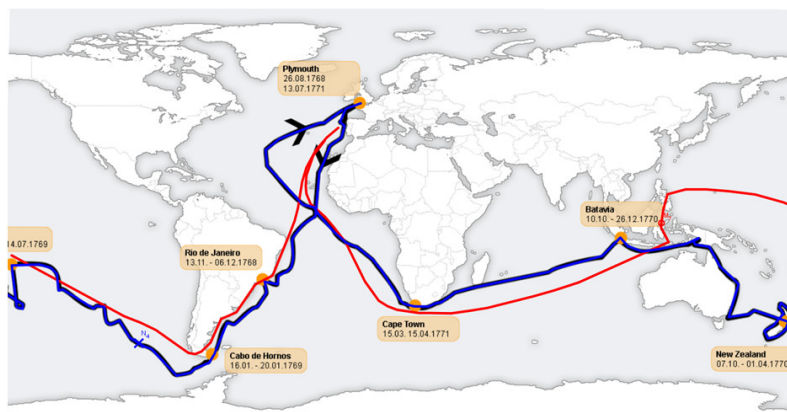


FIGURE 6.5: Tracing the voyages of the great explorers Magellan and Cook

6.3.1.1 Task 1 – Measuring a River’s Length

The first task is to measure the length of a river. In the first step, a map showing the assigned river must be found on the Web. The map is then imported into GeoGebra. In the next step the river is traced with the polygonal line (Figure 6.4). Since the coordinates on the image are arbitrary and unrelated to geographic coordinates, in the final step the actual, real life length of the polygonal line must be calculated. This is accomplished by obtaining the scale of the map: a linear distance between two cities on the map is found and divided by the real distance. The length of the polygonal line is then divided by the scale to calculate the approximate real life length of the river.

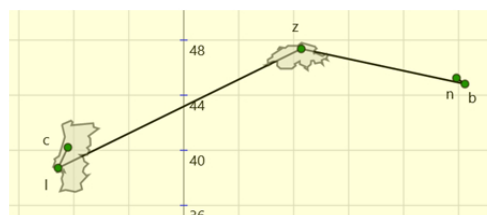


FIGURE 6.6: Tracing a voyage using the Country and City components

Both students and high school pupils were required to perform all the steps in this task within one class, i.e. 45 minutes. The DGS of choice was GeoGebra.

6.3.1.2 Task 2 – Tracing the Great Explorers' Voyages

The second task assigned was to trace the voyages of Ferdinand Magellan and James Cook on the world map (Figure 6.5). Two lists of geographic coordinates, each corresponding to one explorer, were provided in annotated text files. The files contained geographic locations, expressed as pairs of coordinates, and some of the locations were annotated with city names and dates. The students were required to import the coordinates in chronological order into GeoGebra and construct the polylines from both lists. Durations of the voyages and distances traveled were also requested. The pupils were handed out the prepared drawings, without the locations and dates of the significant places. They were given a list of locations and asked to mark them on the map.

6.3.1.3 Task 3 – Tracing One's Own Travels

The students and the pupils were tasked with creating a map of their vacation travels. This task was different from the previous ones, in the sense that each person had traveled to a different place and there were no materials prepared in advance. Being an open-ended problem, the task was discussed with both groups, and the consensus was reached, that the resulting drawing must contain, as a minimum, the countries and cities traveled. The task was first done in GeoGebra, and later in SLGeometry, using the Country and City components. An example solution is shown in Figure 6.6.

LISTING 6.2: Country and City components

```
n = City("NoviSad", 0)
b = City("Belgrade", 0)
z = City("Zurich", 0)
l = City("Lisbon", 0)
c = City("Coimbra", 0)
path = Polyline({n, b, z, l, c})

sw = Country("Switzerland", 0)
sw.$Visible = false
psw = Polygon(sw)

pt = Country("Portugal", 0)
pt.$Visible = false
ppt = Polygon(pt)
```

6.3.2 Results and Comments

Both groups were allotted a time limit of 45 minutes for each test. The percentage of students and pupils who have successfully finished the tasks was recorded and is shown in Table 6.1. Being future mathematics teachers, the students showed greater proficiency and motivation to finish the tasks. As expected, the pupils were more successful in simpler tasks, and quickly lost interest when faced with a difficult and repetitive task (Test 3 without components).

TABLE 6.1: Percentage of finished tasks

Test	Students	Pupils
T1	100% finished on time	65% finished on time
T2	72% finished on time	80% finished on time
T3 – without components	72% finished on time	30% finished on time
T3 – with components	100% finished on time	80% finished on time

Given authors' experience in teaching with GeoGebra and previous tests conducted with SLGeometry, obtained results were more or less expected. After the experiments, we conducted interviews with both groups and asked for their opinions on the tests and suggestions on how the DGS they used could be improved. The students were satisfied with GeoGebra, and, knowing how it operates internally, did not have many suggestions for radical changes. For example, one student remarked that she would like to have a collection of GeoGebra drawings representing maps of all relevant countries, which she would be able to import into her own drawings. Another one wished for a special version of GeoGebra aimed specifically at geography. Several students, however, took a slightly passive attitude and only regretted performing too slow in the experiments. The pupils, on the other hand, expressed more radical ideas, such as having an AI assistant which would recommend the tools based on the task at hand, or being able to download and choose add-ons, in the manner of the popular Web browsers. Both the students and the pupils found the Country and City components very helpful, and one pupil suggested that, instead of having to look for city and country names in the documentation, the DGS should automatically suggest possible argument values – similar to the way AutoCorrect works in programming environments. An insightful comment came from a pupil who said that, with components, he did not have to waste precious time drawing a map, and could focus at the real problem instead. One particular conclusion was not heard in any of the groups, however – duplicating a complex map in a DGS without components leads to the duplication of all the objects it consists of, while duplicating a map component introduces only one additional object. After this conclusion was explained to them, the members of both groups agreed with it.

6.4 Sequential Behavior Controllers in SLGeometry

Further extending our previous work with custom components, we developed behavior controller components for SLGeometry, which act in a sequential manner, i.e. they have inputs, memory and outputs (Table 6.2). With controllers, it is possible to achieve the most common behavior needed in mathematical games, without the need for programming. Instead, the user places controllers on the drawing and sets their properties accordingly. Each type of controller performs a specific type of action, based on input values, and produces the appropriate output. The action does not occur immediately, however. Instead, it is triggered by assigning the logical value True to the special Trigger property of a controller. Controller's outputs are updated immediately after-wards. A special output property, named Done is then briefly set to True and then again to False. The controllers can be daisy-chained

together, by connecting the Done output of one controller to the Trigger input of another. If used correctly, this enables users to create sequential behavior without the need for writing program code.

TABLE 6.2: Sequential behavior controllers and their properties

Controller/property	Type	Description
Sequencer		A counter which cycles through an interval of integer values
Min	Number	Lower bound of the interval
Max	Number	Upper bound of the interval
Value	Number	Current value of the Sequencer
Trigger	Logical	On transition from False to True, the trigger causes Value to increase by 1. After Max is reached, the Sequencer starts again from Min.
Done	Logical, read only	Transitions from False to True to False after Value is updated
Randomizer		Produces a random value from the specified interval
Min	Number	Lower bound of the interval
Max	Number	Upper bound of the interval
Trigger	Logical	On transition from False to True, causes the Randomizer to generate a new Value
Done	Logical, read only	Transitions from False to True to False after Value is updated
Scorekeeper		Adds points to the total score when triggered
Points	Number	Score in the current round of a game
Score	Number, read only	Total score
Trigger	Logical	On transition from False to True, the value of Points is added to Score.
Done	Logical, read only	Transitions from False to True to False after Value is updated

The following examples demonstrate the practical use of the controllers.

6.4.1 Example 1 – Arithmetic Sum

The following example (Figure 6.7) demonstrates the use of triggers and their propagation from one component to another. The goal of this example is to calculate the arithmetic sum of numbers from 1 to 9. We use the Sequencer to produce values from 1 to 9, and the Scorekeeper to add them together. A Pushbutton triggers the Sequencer, and the Sequencer's Done property then triggers the Scorekeeper. The current value of the Sequencer is fed into the Points property of the Scorekeeper.

Clicking the button causes the Sequencer to cycle through numbers from 1 to 9, which are then added to the score, thereby producing an arithmetic sum as a result. In order to achieve this behavior, the user needs only to connect the controllers to each other, as shown in Table 6.3.

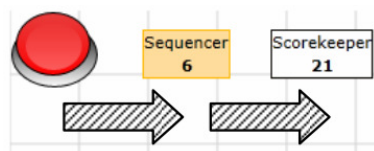


FIGURE 6.7: Propagation of triggers between components - Arithmetic Sum example

TABLE 6.3: Daisy chaining the components via triggers

s.Trigger = p.Pressed	The Sequencer's trigger is connected to the button, so that it advances each time the button is clicked
k1.Points = s.Value	The current value of the Sequencer is connected to the input of the Scorekeeper
k1.Trigger = s.Done	The Scorekeeper adds Points to the sum right after the Sequencer finishes

6.4.2 Example 2 – A Simple Game

This example demonstrates (Figure 6.8) the implementation of the 'guess the midpoint' game in SLGeometry. The game is set up in a similar way as in GeoGebra, with one important difference – the behavior is controlled without script programs. Instead, the game logic is implemented by using controllers. Randomizers r1 and r2 provide random coordinates for point B. Sequencer q keeps track of how many times the game has been played. PushButton p signals that the user has placed the marker and wants to check his/her solution. When the button is clicked, Scorekeeper increases by 1 if the user's solution is correct. Scorekeeper's Done property then triggers the Sequencer and the Randomizers, thus moving point B to a new random location and starting a new round of the game.

The complete code that produces the game is shown in Table 6.4. It should be noted that the code is provided only for the sake of completeness, as the users will create the game using toolbars and mouse. Important commands, which provide sequential behavior, are shown in boldface.

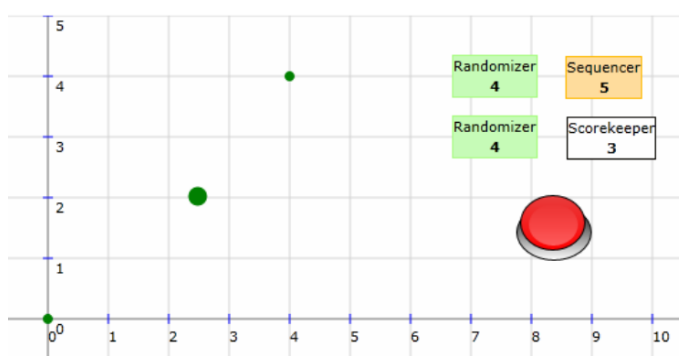


FIGURE 6.8: The "guess the midpoint" game in SLGeometry

TABLE 6.4: Definition of the 'guess the midpoint' game

r1 = Randomizer()	Two Randomizers are created and their intervals set.
r2 = Randomizer()	
r1.Min = 2	
r1.Max = 8	
r2.Min = 1	
r2.Max = 7	
A = (0,0)	Point A is fixed.
B = (r1.Value, r2.Value)	Point B is placed randomly.
X = (2,2)	Point X is supposed to be moved by the user, so we make it appear bigger.
X.Size = 15	
f = Segment(Segment(A,B).Midpoint, X).Length	f is the distance between the point X and the midpoint of the segment AB.
p = PushButton()	A PushButton, a Sequencer and a Scorekeeper are created.
q = Sequencer()	
s = Scorekeeper()	
s.Points = If(f<1, 1, 0)	Current score is fed into the Scorekeeper.
s.Trigger = p.Pressed	Sequential behavior is defined here. The PushButton activates the Scorekeeper, which adds the current score.
q.Trigger = s.Done	
r1.Trigger = s.Done	When the Scorekeeper finishes, it activates the sequencer and both randomizers, preparing the game for the next round.
r2.Trigger = s.Done	

6.4.3 Experiment

In order to test our behavior controllers in practice, we conducted an experiment with a group of 23 first year students of mathematics. We looked for answers to the following questions:

- Are behavior controllers easier to learn and use than script programming?
- Given the choice, how many participants would choose behavior controllers over script programs to implement a mathematical game?
- How well can the students solve a problem, which was not previously demonstrated, using the behavior controllers?
- Do the students find the behavior controllers significantly easier to use than scripting?

6.4.3.1 Assignments

The experiment was conducted during two classes, in a computer lab. In the first class, we taught the students script programming and behavior controllers, and presented the two examples from Section 6.4. Then we asked them to reproduce the examples, with our help. One week later, in the second class, the students were given two assignments:

1. Implement the "Guess the midpoint" game from Example 2, with two random points A and B:
 - (a) using script programming;
 - (b) using behavior controllers.
2. Implement the three-digit counter, which consists of three Sequencers. The counter should advance when a PushButton is pressed. (*A car mileage meter was shown as an example.*)

The second assignment, which was not presented before, was to implement a multi-digit counter using the Sequencer control. Both assignments were graded from 1 (fail) to 4 (excellent). Finally the students were polled on their opinions about the behavior controllers.

6.4.4 Results and Comments

The students were graded in the following manner:

- First, the students which had solved their assignments correctly, were graded 'Excellent';
- The students which had made minor syntax errors were shown how to correct them. Such students were graded 'Good';
- Finally, we gave additional instructions to the rest of the students. Then, the ones which successfully solved the problems were graded 'Satisfactory', and the rest received 'Fail'.

TABLE 6.5: Students' test scores

	Assignments		
	A1a	A1b	A2
Fail	6	2	0
Satisfactory	2	4	3
Good	8	7	6
Excellent	7	10	14

TABLE 6.6: Poll results

Question	Yes	No
Are behavior controllers easier to use than script programs?	23	0
Do you find one class enough to learn behavior controllers?	20	3
Do you find one class enough to learn script programming?	6	17
If you had to choose only one method of controlling the behavior of your interactive drawing, would you choose behavior controllers over scripting? Explain.	15	8
Your opinion on behavior controllers, in writing?	-	-

Test results are shown in Table 6.5. Columns 'A1a' and 'A1b' contain results from the first assignment, solved using scripting and behavior controllers, respectively. Column 'A2' contains results from the second assignment.

Regarding the first assignment, success rate was better with behavior controllers than with scripts. While eight students struggled with script programming and subsequently six of them failed to complete the assignment, only two failed to complete the same assignment using controllers. The number of excellent results was also bigger with the controllers (10 versus 7).

The amount of help we had to provide was also different in favor of controllers. In A1a, we had to explain both the syntax and semantics of the script commands, while in A1b we only had to remind the students how to correctly address controller properties.

The second assignment was successfully completed by all the students. As with the previous assignment, we helped the students, which scored 'Good', by correcting minor syntax errors. The three students which scored 'Satisfactory' could not formulate the correct criterion for triggering the next digit by themselves.

We polled the students on their opinions about our behavior controllers. The questions and answers are presented in Table 6.6.

We conclude from the answers that all the participants recognized the simplicity that the behavior controllers offer. Most of the students learned how to use them without difficulty. The opposite holds for script programming. Learning how to program was difficult for students, especially for those who had no previous programming knowledge. However, approximately one third of the students recognized that scripting offers more choice and versatility. They stated that they were confident enough in their programming skills to use script programming instead of controllers.

The participants answered the last question in writing, and later we had a discussion with them, which resulted in some interesting opinions and observations:

- 'I can actually see what the controllers do. Scripts are invisible and not very clear to me.'
- 'With controllers, I don't have to memorize commands.'
- 'Scripting is more powerful, but too difficult for me.'
- 'I wish you created more behavior controllers for more complex games.'

While most students agreed that the controllers are a useful addition to a DGS, some argued that they did not offer enough flexibility. On the other hand, more than half of the students said that they were able to create a simple mathematical game with controllers, while they could not do the same with scripting. As the students of mathematics are the future teachers of mathematics, we find that this conclusion justifies the use of behavior controllers.

6.5 Summary

Components in dynamic geometry software bring new functionality without the need for programming. Instead of constructing complex objects from geometric objects, authors can simply place a visual component on the drawing surface and use it as an atomic object. Components allow authors to focus on the main task instead of looking for ways to make DGS do things they weren't designed for in the first place. Also, components enable application of DGS in subjects other than mathematics and help authors who are not proficient in development of learning materials. Besides the many benefits of the components, there is one significant shortcoming – components must be implemented by programmers, outside of the DGS, which puts component development out of the reach of ordinary DGS users. This problem may be alleviated by developing an online component library.

We identified a pattern of behavior, common to many mathematical games created in GeoGebra. The behavior of such games is usually controlled by script programs, which can be difficult to write and maintain. We developed several sequential behavior controllers, which help in creation of mathematical games in SLGeometry DGS without the need for programming. It is our aim to develop additional controllers, to enable the users to create games with more complex behavior. The use of controllers needs not be limited to games only.

In our experiments we found that our results are, in general, in accordance with the conclusions of other authors, see, for example, Hohenwarter et al. (2008), Hohenwarter and Preiner (2007), Mott et al. (2008), Prensky (2007), Tatarczak and Medrek (2017). As others have also noted, the students take a more active role when learning is supported with a DGS. Blended learning and phenomenon-based learning (Kashefi, Ismail, and Yusof, 2012; Lavicza et al., 2018) in geography were made possible by DGS, and the approach we took by extending SLGeometry with components was well received by our pupils and students. Regarding the implementation aspect, SLGeometry is a component-based system as outlined by Crnkovic, Chaudron, and Larsson (2005), Jahn et al. (2013), Radaković and Herceg (2018). Separation of development cycles for the host application and the components enables independent creation of extensions for various subjects. We can conclude that a DGS, extended with components, can easily be adapted for specific subjects and hardware (Tomaschko

and Hohenwarter, 2017), or connected to other software (Quaresma, Santos, and Baeta, 2018; Quaresma et al., 2008).

Chapter 7

Testing Lazy Evaluation

In the previous chapters, Chapter 4 and Chapter 5, we have presented the SLGeometry framework architecture and its components, along with the proposed structural metadata infrastructure and the property activation mechanism which implements lazy evaluation.

But to be sure, that the given infrastructure also benefits, both, the evaluation speed and memory footprint, we need an experimental verification. In other words, the main motivation for the experimental work presented in this chapter is the proof that our infrastructure is not slower, or with larger number of created objects, compared to traditional functional solutions.

To achieve that, we compare three different expression tree evaluation strategies:

1. eager,
2. functional, and
3. lazy evaluation.

For each of these approaches we developed a separate project with the same core, but different evaluation engines. The performances of all evaluation schemes were compared. Since in the eager evaluation scheme, all calculated properties are instantiated and evaluated, regardless of whether they are referenced or not, this approach is considered as the 'worst case' scenario, to better highlight advantages of the other two approaches.

Certainly, it is supposed that the lazy evaluation brings the best results, although it could be easily expected that the overhead of objects happened, in particular with objects with big number of properties.

Evaluation speed and memory footprint were observed as performance indicators. The experiment was based on repeated evaluation of drawings with different number of objects with calculated properties (Figure 7.1), using all three evaluation strategies.

Next, we argument why we have chosen the triangle for the geometrical shape that we use in our experiments. Section 7.2 gives the list of properties which are added to the triangle and calculated. Experimental setup is presented in Section 7.3. We compare and discuss performance and memory footprint of all tree evaluation schemes in last two sections.

7.1 Why the Triangle?

Since the triangle is one of the basic geometrical shapes that is taught from early childhood, a great number of interactive examples have been developed in many DGS or other mathematical software (*GeoGebra materials: triangle*, 2019; *Cinderella Gallery: Altitudes in a triangle*, 2019; Engstrom, 2001; Arzarello et al., 2002; *Cabrilog: Echantillon de Cabri Factory – Parallélogramme*, 2019; *Wolfram Math World: Triangle*, 2019) and many tutorials exist that can be found on the Web, including Wikipedia, YouTube, etc. (Lernpfad, 2019; *Math Open Reference, Triangles*, 2019; *StackExchange – Mathematics*, 2019; *Cut The Knot: The many ways to construct a triangle*, 2019; *Geometrie interactive - le cercle qui tourne le triangle*, 2019; Dahan, 2014). In Euclidean geometry, special points and lines of the triangle, including the incenter, centroid, circumcenter and orthocenter, have always attracted interest. Many researches investigate the construction task in geometry while some concentrate on learning trajectory for the triangle topic, especially triangle construction (Wernick, 1982; Yiu, 2008; Marinković and Janičić, 2012; Marinković, 2015; Zhuravlev and Samovol, 2016; Anwar and Rofiki, 2018).

Clark Kimberling's Encyclopedia of Triangle Centers – ETC (Kimberling, 2019; Kimberling, 1993) counts more than 10 000 triangle centers. Since the triangle is a well-known geometrical shape, and pupils in schools should learn its properties according to the national curricula for mathematics in European countries (*Lehrplan PLUS Mittelschule – Juni 2016, Bayerisches Staatsministerium für Bildung und Kultus, Wissenschaft und Kunst, Staatsinstitut für Schulqualität und Bildungsforschung München*, 2016; *Nastavni plan i program za osnovnu školu 2013, Ministarstvo znanosti i sporta Republike Hrvatske*, 2013; *Progressions pour le cours préparatoire et le cours élémentaire première année: Mathématiques, Ressources pour l'école élémentaire*, 2012; *Nastavni planovi i programi za osnovne i srednje škole, Zavod za unapređenje obrazovanja i vaspitanja, Republika Srbija*, 2016; *Štátny pedagogický ústav, Štátny vzdelávací program, Vzdelávacia oblasť: Matematika a práca s informáciami, Slovenskej republiky, Slovakia*, 2019; *Department for education, Mathematics programmes of study: key stages 1 and 2, National curriculum in England, September 2013*, 2013), we decided to use the triangle and its properties as the base for constructions in the experiments.

7.2 Constructions

Test examples are created starting from a base group of objects, similar to one in Wernick's list (Wernick, 1982), which consists of a triangle with:

- altitude on side a ;
- angle bisector for angle α ;
- perpendicular bisector for side a ;
- circumcenter O ;
- circumcircle cc .

For the extended testing of object and memory consumption, the following properties are added to the triangles:

- triangle perimeter;
- side a ;
- side b ;
- altitude foot of altitude on side c ;
- triangle area;
- inradius;
- median from A ;
- median line through B ;
- median line through C ;
- orthocenter;
- coordinate X from midpoint of side a ;
- coordinate Y from altitude foot on side b ;
- triangle with vertices in midpoints of sides a, b, c ;
- coordinate X from midpoint of side c of the inner triangle;
- coordinate Y of vertex C of the inner triangle;
- coordinate Y from midpoint of side b of the inner triangle;
- coordinate Y of vertex B of the inner triangle;
- perimeter of the inner triangle.

Given the parameter n , additional n groups of objects are created, using the construction steps in Table 7.1 and Table 7.2. The vertices of the base triangle have constant coordinates, and the vertices of the dependent triangles are calculated from them, so that dependent triangles are arranged in rows with five triangles in each row. The dependent triangles are shifted by 1 on the x-axis, and each row is shifted by -1 on the y-axis. The parameter $i = 1, \dots, n$ denotes the index of a dependent triangle, and, the expressions $k = i|5$ and $j = i|5$ specify its position.

For experiments involving lazy and eager evaluation, the constructions were specified in the object oriented syntax. For experiments involving functional evaluation, the constructions were specified using the functional syntax, i.e. for each property a separate function was applied. The constructions were created for $n = 5, 10, 20, 30, 40$.

7.3 Experimental Setup

All experiments were conducted on three computer configurations shown in Table 7.3. The software was run from inside Visual Studio 2015 and its built-in Diagnostic Tools were used to track and measure performance. CPU time was measured on each computer in the following way:

1. For a given n , the construction was created,
2. Coordinates of the point A in the base triangle were changed, causing all the dependent objects to be evaluated,
3. Step 2 was repeated a set number of times and total time was taken.

We started with $n = 5$, by timing 50, 100, 200, 300 and 500 evaluations. Since there were no significant deviations in the results, we performed all subsequent experiments with 100 evaluations. We performed the same measurements for $n = 10, 20, 30, 40$. Memory footprint was measured in the following way:

TABLE 7.1: Constructions used in tests, specified in OO syntax. For dependent triangles, $i = 1, \dots, n; k = i|5; j = i|5$

	Base group	Dependent groups
Base tests	$A = (0.68, 0.83)$	$A_i = (A.X + k, A.Y - j)$
	$B = (0.14, 0.14)$	$B_i = (B.X + k, B.Y - j)$
	$C = (0.79, 0.26)$	$C_i = (C.X + k, C.Y - j)$
	$t = \text{Triangle}(A, B, C)$	$t_i = \text{Triangle}(A_i, B_i, C_i)$
	$uab = \text{Angle}(B, A, C).\text{Bisector}$	$uab_i = \text{Angle}(B_i, A_i, C_i).\text{Bisector}$
	$sab = t.\text{SideA}.\text{Bisector}$	$sab_i = t_i.\text{SideA}.\text{Bisector}$
	$al = t.\text{AltitudeA}$	$al_i = t_i.\text{AltitudeA}$
	$o = t.\text{Circumcenter}$	$o_i = t_i.\text{Circumcenter}$
	$cc = \text{Circle}(o, A)$	$cc_i = \text{Circle}(o_i, A_i)$
	Additional tests	$per = t.\text{Perimeter}$
$a = t.\text{SideA}$		$a_i = t_i.\text{SideA}$
$b = t.\text{SideB}$		$b_i = t_i.\text{SideB}$
$hcf = t.\text{AltitudeFootC}$		$hcf_i = t_i.\text{AltitudeFootC}$
$ar = t.\text{Area}$		$ar_i = t_i.\text{Area}$
$inr = t.\text{Inradius}$		$inr_i = t_i.\text{Inradius}$
$ma = t.\text{MedianA}$		$ma_i = t_i.\text{MedianA}$
$mbl = t.\text{MedianLineB}$		$mbl_i = t_i.\text{MedianLineB}$
$mcl = t.\text{MedianLineC}$		$mcl_i = t_i.\text{MedianLineC}$
$o2 = t.\text{Orthocenter}$		$o2_i = t_i.\text{Orthocenter}$
$amx = a.\text{Midpoint.X}$		$amx_i = a_i.\text{Midpoint.X}$
$afby = t.\text{AltitudeFootB.Y}$		$afby_i = t_i.\text{AltitudeFootB.Y}$
$t2 = \text{Triangle}(t.\text{SideA}.\text{Midpoint}, t.\text{SideB}.\text{Midpoint}, t.\text{SideC}.\text{Midpoint})$		$t2_i = \text{Triangle}(t_i.\text{SideA}.\text{Midpoint}, t_i.\text{SideB}.\text{Midpoint}, t_i.\text{SideC}.\text{Midpoint})$
$scmx = t.\text{SideC}.\text{Midpoint.X}$		$scmx_i = t_i.\text{SideC}.\text{Midpoint.X}$
$t2cmx = t2.C.X$		$t2cmx_i = t2_i.C.X$
$sbmy = t.\text{SideB}.\text{Midpoint.Y}$		$sbmy_i = t_i.\text{SideB}.\text{Midpoint.Y}$
$t2bmy = t2.B.Y$		$t2bmy_i = t2_i.B.Y$
$per2 = t2.\text{Perimeter}$		$per2_i = t2_i.\text{Perimeter}$

1. The number of objects on the heap and heap size were taken after the application was run,
2. For a given n , the construction was created,
3. The number of objects on the heap and heap size were taken after the example was created.

We started with example of triangle with 9 properties (Table 7.1 and Table 7.2, basic test) for all tree approaches. Since lazy and functional evaluations produced close values, we increased the number of properties, starting with 3 and increasing by 3 until 27 properties (Table 7.1 and Table 7.2) for both approaches. These measurements were repeated three times.

As the eager evaluation scheme always calculates all properties of all the triangles and other geometric objects in the drawing, we therefore expect it to perform slowest. On the other side, with the lazy evaluation scheme only referenced properties are calculated. In the functional evaluation scheme for each property a separate function is employed for each property that needs to be calculated. With regards to the postulates, we assume that the lazy evaluation scheme should perform at least

TABLE 7.2: Constructions used in tests, specified in Functional syntax. For dependent triangles, $i = 1, \dots, n; k = i|5; j = i|5$

	Base group	Dependent groups
Base tests	A = (0.68, 0.83)	A_i = (A.X + k, A.Y - j)
	B = (0.14, 0.14)	B_i = (B.X + k, B.Y - j)
	C = (0.79, 0.26)	C_i = (C.X + k, C.Y - j)
	t = Triangle(A, B, C)	t_i = Triangle(A_i, B_i, C_i)
	uab = Bisector(Angle(B, A, C))	uab_i = Bisector(Angle(B_i, A_i, C_i))
	sab = Bisector(Side(1, t))	sab_i = Bisector(Side(1, t_i))
	al = Altitude(1, t)	al_i = Altitude(1, t_i)
	o = Circumcenter(t)	o_i = Circumcenter(t_i)
	cc = Circle(o, A)	cc_i = Circle(o_i, A_i)
	per = Perimeter(t)	per_i = Perimeter(t_i)
	a = Side(1, t)	a_i = Side(1, t_i)
	b = Side(2, t)	b_i = Side(2, t_i)
	hcf = AltitudeFoot(3, t)	hcf_i = AltitudeFoot(3, t_i)
	ar = Area(t)	ar_i = Area(t_i)
inr = Inradius(t)	inr_i = Inradius(t_i)	
Additional tests	ma = Segment(Midpoint(Side(1, t)), A)	ma_i = Segment(Midpoint(Side(1, t_i)), A)
	mbl = Line(Midpoint(Side(2, t)), B)	mbl_i = Line(Midpoint(Side(2, t_i)), B)
	mcl = Line(Midpoint(Side(3, t)), C)	mcl_i = Line(Midpoint(Side(3, t_i)), C)
	o2 = Orthocenter(t)	o2_i = Orthocenter(t_i)
	amx = Coordinate(1, Midpoint(a))	amx_i = Coordinate(1, Midpoint(a_i))
	afby = Coordinate(2, AltitudeFoot(2, t))	afby_i = Coordinate(2, AltitudeFoot(2, t_i))
	t2 = Triangle(Midpoint(Side(1, t)), Midpoint(Side(2, t)), Midpoint(Side(3, t)))	t2_i = Triangle(Midpoint(Side(1, t_i)), Midpoint(Side(2, t_i)), Midpoint(Side(3, t_i)))
	scmx = Coordinate(1, Midpoint(Side(3, t)))	scmx_i = Coordinate(1, Midpoint(Side(3, t_i)))
	t2cmx = Coordinate(1, Vertex(3, t2))	t2cmx_i = Coordinate(1, Vertex(3, t2_i))
	sbmy = Coordinate(2, Midpoint(Side(2, t)))	sbmy_i = Coordinate(2, Midpoint(Side(2, t_i)))
	t2bmy = Coordinate(2, Vertex(2, t2))	t2bmy_i = Coordinate(2, Vertex(2, t2_i))
	per2 = Perimeter(t2)	per2_i = Perimeter(t2_i)

TABLE 7.3: Computer configurations used for testing

Computer	CPU	RAM	OS
Configuration 1	Intel Core i5 2.4 GHz	8 Gb	Windows 10 Pro 64-bit
Configuration 2	Intel Core i5 3.2GHz	16 Gb	Windows 10 Pro 64-bit
Configuration 3	AMD E-450 1.6 GHz	4 Gb	Windows 10 Pro 64-bit

as fast as the functional evaluation scheme, while both should outperform the eager by a great margin. As for memory consumption, the lazy evaluation scheme is expected to require more memory compared to the functional approach, because of the implementation requirements

7.4 Results, Analysis and Discussion

We analyze the performance and memory footprint of all tree evaluation schemes. One-way Analysis of Variance (ANOVA) and non-parametrical Kruskal-Wallis ANOVA are used to test the statistical significance of differences between test results of all three experimental approaches. The statistical significance level is set to $p < 0.05$. For these analyses we used Statistica 13.

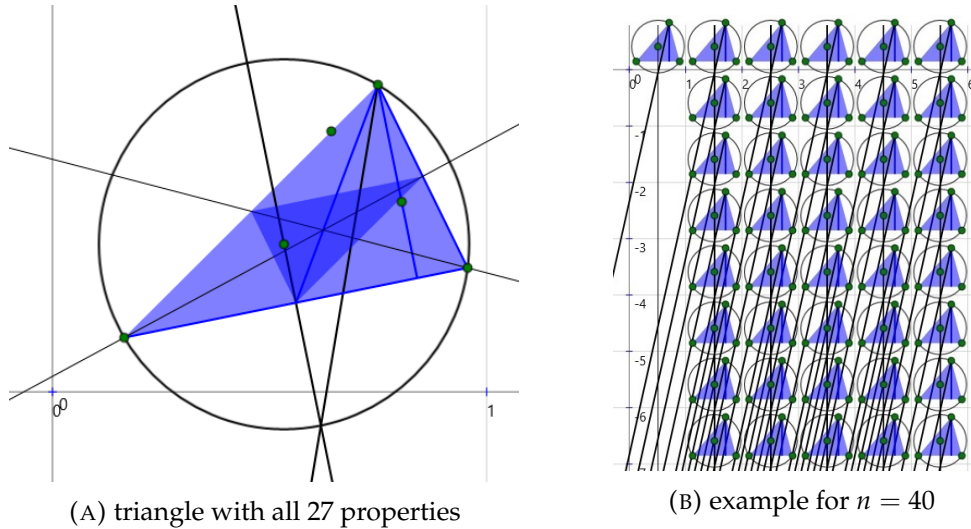


FIGURE 7.1: Single triangle with all 27 properties (A) and an example for $n = 40$ in SLGeometry (B)

The ANOVA testing is used to determine whether at least two groups are different, but without noticing which group-pairs. Therefore, we checked all the pairs between eager, lazy and functional schemes for each measurement, using the Post-hoc Tukey HSD test and obtained that all group-pairs are statistically significantly different from each other with $p < 0.05$.

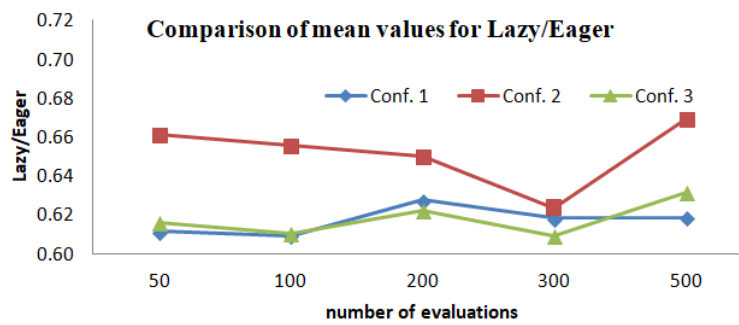
CPU time measurements for all approaches are presented together, and are shown in tables with mean \pm standard deviation values for all approaches and different values of n . To compare the values we also observed the ratio of the mean values for Lazy/Eager and Lazy/Functional evaluation schemes.

The study is divided into three subsections. In the first one, we analyze CPU time tests for $n = 5$. In the second subsection, we widen our analysis to speed tests for $n = 10, 20, 30, 40$. In the third subsection, we perform object count and memory consumption testing. Each group of tests is discussed for each computer configuration separately.

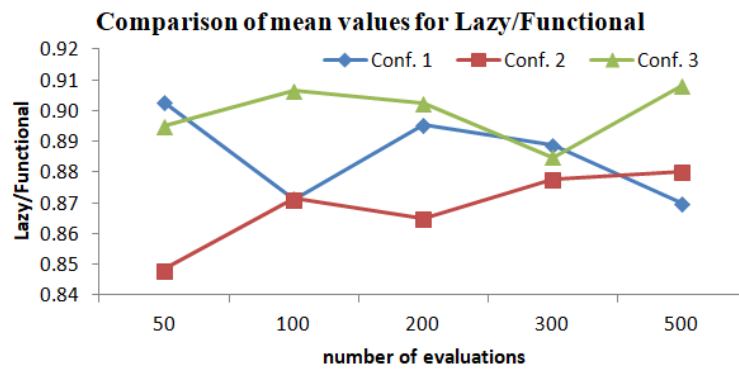
7.4.1 CPU Time Tests for 5 Triangles

We started the testing using the example for $n = 5$ by measuring CPU time for 50, 100, 200, 300 and 500 evaluations. We ran 10 tests for each evaluation scheme on each computer. Since the number of tests is not large and we could not check the assumption of a normal distribution of the results, we used the non-parametrical Kruskal-Wallis ANOVA test to compare the results. In Table 7.4 multiple comparisons of p-values for each approach with $p < 0.05$ are shown (for Configuration 3, 50 evaluations), thus the data samples have statistically significantly different results. The comparisons of p-values for all other configurations and all numbers of triangle movements are similar, i.e. all the data are comparable.

All three configuration tests have the best results for the lazy evaluation scheme, shown in Table 7.5. Eager performed slower than lazy, 37-39% (1 and 3) and 33-38% (2), while functional was slower 10-13% (1), 12-15% (2) and 9-12% (3) (Figure 7.2).



(A) Lazy/Eager



(B) Lazy/Functional

FIGURE 7.2: Comparative graphs of mean values of CPU time for ratio Lazy/Eager (A) and Lazy/Functional (B) on all configuration ($n = 5$)

TABLE 7.4: Multiple Comparisons p values (2-tailed); Results Conf3M50, Independent (grouping) variable: Groups Conf3M50
Kruskal-Wallis test: $H(2, N = 30) = 25.82944$ $p < 0.05$ Computer configurations used for testing

Multiple Comparisons p values (2-tailed); Results Conf3m50			
Results Conf3m50 (Conf3 m all data 2 better) Independent (grouping) variable: Groups Conf3m50			
Depend.: Kruskal-Wallis test: $H(2, N = 30) = 25.83519$ $p = .0000$			
	50eager R:25.500	50lazy R:5.5000	50func R:15.500
50eager		0.000001	0.033255
50lazy	0.000001		0.033255
50func	0.033255	0.033255	

TABLE 7.5: Mean \pm standard deviation values of CPU time on all configurations ($n = 5$)

	Mean values \pm SD	50	100	200	300	500
Conf. 1	Eager	1008.3 \pm 54.052	1972.4 \pm 55.9	3940 \pm 29.38	5916.6 \pm 78.09	9867.5 \pm 92.16
	Lazy	616.5 \pm 14.269	1201.4 \pm 28.98	2472 \pm 25.05	3657.1 \pm 38.31	6104.7 \pm 88.82
	Functional	682.9 \pm 10.969	1379 \pm 18.71	2761.3 \pm 45.83	4114.2 \pm 96.94	7019.4 \pm 121.35
Conf. 2	Eager	648.6 \pm 12.231	1293 \pm 12.96	2576.7 \pm 17.02	4066.8 \pm 19.29	6376.4 \pm 33.13
	Lazy	429 \pm 8.692	847.4 \pm 10.606	1675.5 \pm 13.87	2536.2 \pm 20.35	4267.3 \pm 19.19
	Functional	505.8 \pm 12.164	972.6 \pm 10.814	1937.5 \pm 16.47	2889.4 \pm 19.91	4848 \pm 25.1
Conf. 3	Eager	4414.4 \pm 30.1	8963.3 \pm 67.72	17456.5 \pm 453.6	26591.1 \pm 184.2	43148.9 \pm 343.1
	Lazy	2717.9 \pm 43.76	5472.4 \pm 69.38	10855.6 \pm 88	16198.3 \pm 140.9	27244.2 \pm 296.3
	Functional	3037.2 \pm 43.21	6037.6 \pm 58.8	12030.4 \pm 129.7	18306.1 \pm 249.5	29999.3 \pm 274.8

7.4.2 CPU Time Tests for 100 Evaluations

Since there were no significant deviations in the results presented in Section 7.4.1, performance testing on more complex drawings with $n = 10, 20, 30, 40$ was carried out by measuring CPU time for 100 evaluations. We used one-way ANOVA to analyze our results in these tests. The observed results are consistent with the normal distribution ($N = 30$ for each approach and each computer). Table 7.5 presents the CPU time test results ran on all configurations.

Configuration 1 CPU time tests. Figure 7.3 shows the results in box plot diagrams for $n = 5, 10$. Both plots in Figure 7.3 show the CPU time for all 5 loads of triangles. Plots use the point in the center of the box to show the mean, with its numeric value in the text box; box edge values are mean \pm standard deviation; whiskers denote minimal and maximal values. In the diagrams it is easily seen that the lazy approach is better than eager and functional in all cases. The eager approach is worse than the functional. When we compare the mean values in Figure 7.4, we can say that the lazy approach is better than the functional cca. 11%, and compared with the eager approach up to 46%.

Configuration 2 CPU time tests. In this case, according to better computer configuration all results were faster than Configuration 1 results. For all groups, the lazy approach is the fastest, afterwards the functional comes with cca. 10% delay, while the eager values have a delay ranging from 34% to 57% (Figure 7.4).

Configuration 3 CPU time tests. As this configuration is the weakest; all results were the slowest (Table 7.6). Again the lazy approach was the fastest. This time, the functional approach varied from 10% to 16% delay, while eager delays were 38-54% (Figure 7.4).

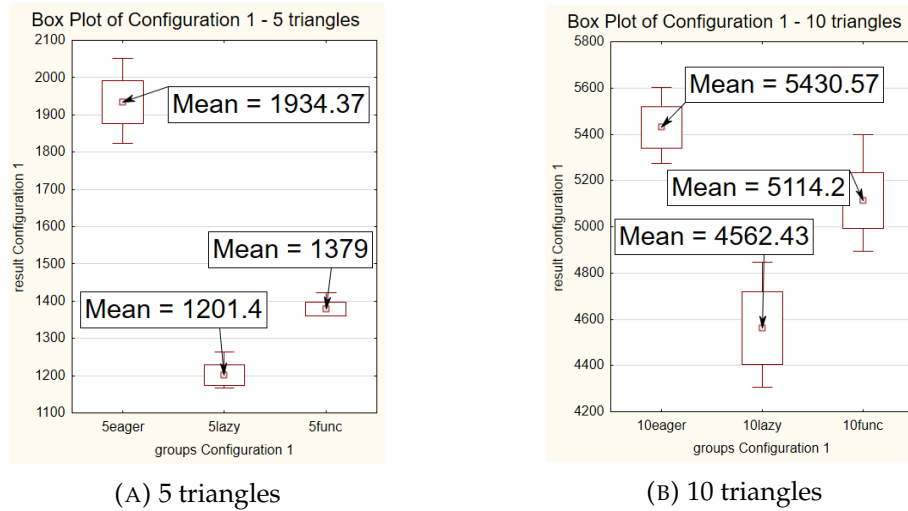
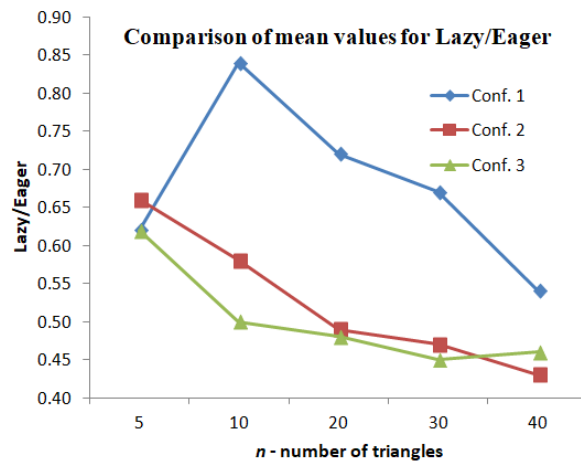
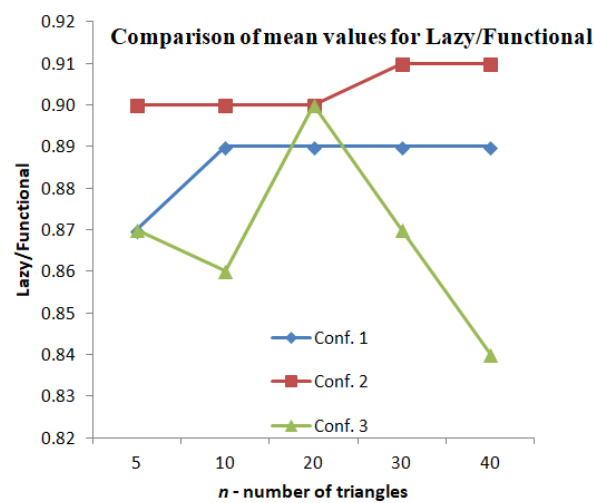


FIGURE 7.3: Box Plot of results made on Configuration 1: (A) 5 triangles (B) 10 triangles



(A) Lazy/Eager



(B) Lazy/Functional

FIGURE 7.4: Comparative graphs of mean values of CPU time of ratio Lazy/Eager (A) and Lazy/Functional (B) on all configuration ($n = 5, 10, 20, 30, 40$)

TABLE 7.6: Mean \pm standard deviation values of CPU time on all configurations ($n = 5, 10, 20, 30, 40$)

Mean values \pm SD	5	10	20	30	40	
Conf. 1	Eager	1934.37 \pm 57.57	5430.57 \pm 98.05	16975.40 \pm 1218.7	35001.30 \pm 1735.1	70502.50 \pm 12461.6
	Lazy	1201.4 \pm 27.97	4562.43 \pm 126.1	12257.60 \pm 364.6	23396.90 \pm 848	38137.90 \pm 1433.9
	Functional	1379 \pm 18.05	5114.20 \pm 119.6	13701.40 \pm 299.5	26338.60 \pm 739	42887.60 \pm 1737.6
Conf. 2	Eager	1315.900 \pm 28.92	3496.07 \pm 17.65	11000.9 \pm 102.9	22489.5 \pm 78	38312.6 \pm 197.6
	Lazy	869.667 \pm 20.704	2012.13 \pm 33.72	5371.2 \pm 98.49	10276.5 \pm 197.2	16546 \pm 372.4
	Functional	962.133 \pm 18.821	2231.57 \pm 24.67	5992.3 \pm 32.27	11270.4 \pm 73.1	18152.9 \pm 80.3
Conf. 3	Eager	8515.10 \pm 418.45	25612.9 \pm 1512.7	87274.3 \pm 1469.8	179995 \pm 146729	283862 \pm 12391
	Lazy	5260.93 \pm 287.94	12717.4 \pm 833.3	41883.4 \pm 129.5	81739.9 \pm 59191	130807 \pm 4461
	Functional	6048.40 \pm 521.23	14866 \pm 1418.8	46928.7 \pm 1193.5	94138.9 \pm 68539.3	155030 \pm 2691

7.4.3 Object Count and Memory Tests

In addition to CPU time testing we conducted memory footprint tests, by measuring the number of objects on the heap and total memory consumption. Tests are divided in two parts, measuring object count and memory consumption for triangles:

1. with 9 properties;
2. starting with triangle vertices and triangle, and adding 3 more properties for each new test.

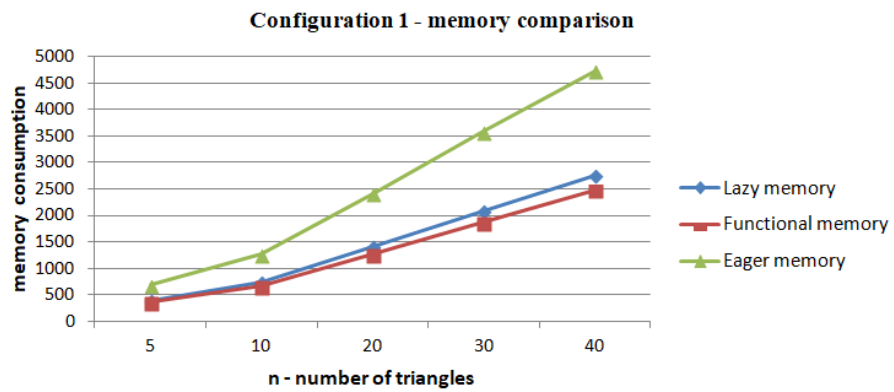
In the first part all tree approaches were compared, and in second only the lazy and functional ones.

7.4.3.1 Measuring for 9 Properties of Triangle

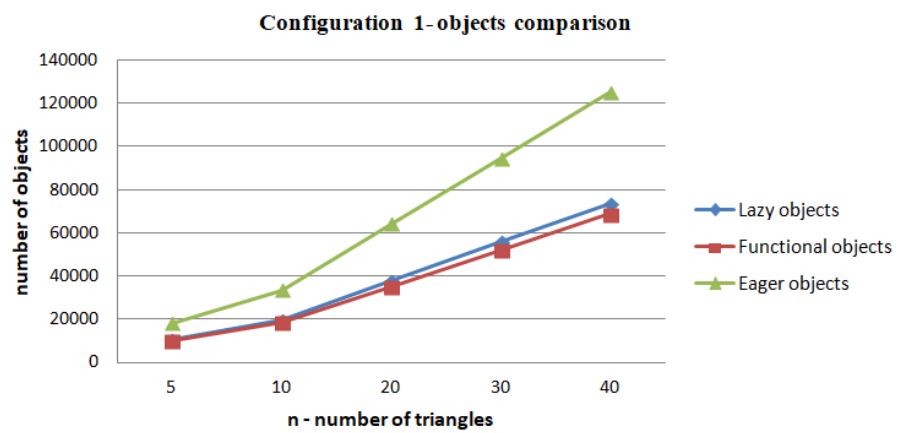
Table 7.7 shows the average difference values of objects and used memory, before and after a test example is loaded from file, on Configuration 1. The values differ between evaluation schemes due to different evaluation engine implementations. With the increase of the number of loaded objects, the number of objects on the heap and memory consumption also increases. Figure 7.5 shows average differences between object count (a) and memory consumption (b) values before and after loading of test examples for Configuration 1. The graphs for other configurations are very similar (with only up to 0.5% difference). In all cases the eager approach shows worst results. The best values for object count are obtained with the functional approach and compared to lazy evaluation the results are 7% better. Similar results are obtained for memory consumption, with functional requiring up to 10% less memory.

TABLE 7.7: Average difference between heap object count and memory consumption before and after the test examples are loaded for Lazy, Functional and Eager testing (Config. 1, 9 properties)

n	Lazy		Functional		Eager	
	Objects	Memory (Kb)	Objects	Memory (Kb)	Objects	Memory (Kb)
5	10649.33	397.65	9911.00	354.95	18158.00	686.57
10	19673.33	738.38	18312.00	661.52	33464.00	1268.59
20	37702.67	1405.74	35120.67	1274.47	64092.00	2420.19
30	55727.00	2088.75	51913.67	1877.09	94714.00	3587.52
40	73762.67	2749.47	68832.00	2490.43	125369.67	4733.99



(A) Memory comparison



(B) Object comparison

FIGURE 7.5: Memory (A) and object (B) comparison for Configuration 1 (9 properties)

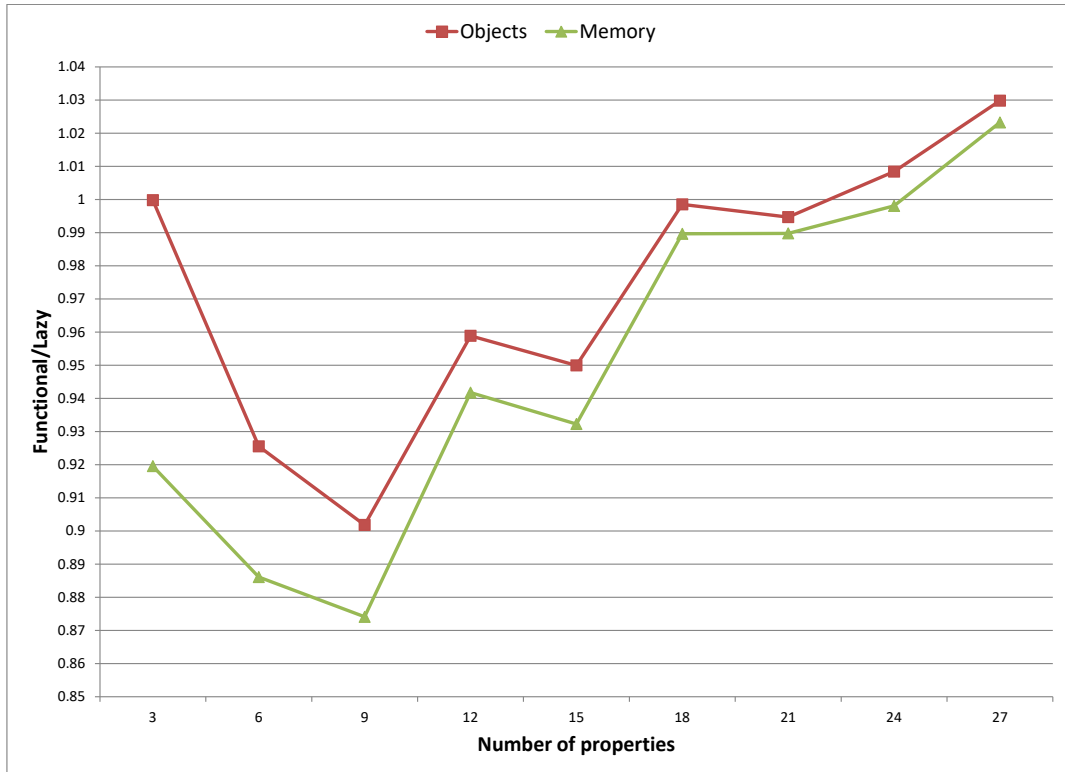


FIGURE 7.6: Average memory and object comparison Functional/Lazy for all 27 properties of the triangle

7.4.3.2 Measuring for 27 Properties of Triangle

According to the Table 7.1 and Table 7.2 first comparison of the number of objects on the heap and total memory consumption was for triangle with its vertices. Each next step we added 3 more properties. As the number of properties increases, the properties are more complicated in a sense that they need more property activations and the activations occur on several levels, as explained in Section 4.

Figure 7.6 shows the ratio of average values for each group of properties of Functional/Lazy memory and object consumption. The values >1 show cases when Lazy approach is better than Functional, what is our aim prove that they exist after some conditions are achieved. It is easy to see that the object consumption is equal for starting triangle. When triangles have 6 and 9 properties Lazy approach takes 8-10% more objects and with 12 and 15 properties that backlog is better cca. 5%, while with 18-21 properties it is just 0.5-0%. The turnaround is reached after having 24 and 27 properties, when Lazy approach gives better results for cca. 1-2%.

The memory ratio Functional/Lazy consumption looks similar with the exception of the starting triangle, where Functional approach is better for 8% due to complex metadata in Lazy approach.

7.5 Summary

Bestow to the results from the experiments we can make several observations.

First, the lazy evaluation scheme has the best execution time for all configurations and all experimental settings. The functional evaluation scheme is second best with a delay of 9-12%, while for the eager approach delays range between 33-57%. The advantage of the lazy evaluation scheme over the functional one can be attributed to optimized evaluation of properties, where common intermediate results are calculated only once.

Second, memory consumption and object count tests confirmed that the eager approach yields worst results. Functional evaluation performed slightly better than lazy evaluation regarding memory footprint for triangles without many properties. Looking closely at the results in Table 7.7, however, we can conclude that the slight increase of memory and object consumption is easily offset by the benefits of the object oriented approach, which were discussed in detail in Section 3.4.2.

Third, memory and object consumption for lazy evaluation gives better results for triangles with more complex properties after the number of properties exceeds 24. This value may vary depending on the complexity of the construction and objects involved.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this dissertation we presented a framework for specifying metadata which bears structural and semantic information in the FLG and demonstrated its use on several types of code entities (Chapter 4). The proposed framework offers greater flexibility than attributes, while retaining their functionality and declarative nature. The problem of storing values of custom types in metadata (Section 3.2), as well as structural information, which could not be overcome using attributes, is easily solved using our approach (Section 5.1). We also provided guidelines for implementation of new data types, operations, type conversions and functions.

Thanks to pervasive use of metadata to annotate all aspects of DGS, we have successfully decoupled specific data types, operations, functions and visuals from the core components of the software, which means that the software can be extended with new features, or the default set of features can be completely replaced (Section 3.4.1, Section 3.4.2). By introducing the constant object data type and dot notation, we have avoided complex syntax from the language and unified the objects with their properties (Section 3.4). This achievement benefits the users and developers of SLGeometry alike. For the user, expression syntax has become simpler and more straightforward; for the developer, the implementation of a constant object type and its properties can be unified inside a single C# class instead of being scattered throughout multiple functions. Thanks to the metadata, object properties have become discoverable and various typing and visual aids can be developed, which will assist the user while manipulating objects with properties.

The field of DGS has been developing quickly in the last decade. Due to ease of use and general availability, DGS are widely used by teachers to create and disseminate teaching and learning materials, and increasingly so in the fields other than geometry. Therefore a need has arisen for easy extensibility of DGS. We believe that the principles and methods highlighted in this dissertation can easily be adapted to any DGS developed in C#, Java and similar languages, since the implementation in the Engine is relatively simple, and the main weight of the solution lies with the idea of structured metadata.

Components in dynamic geometry software bring new functionality without the need for programming. Our aim is to develop additional controllers, to enable the users to create games with more complex behavior. Besides the many benefits of the components, there is one significant shortcoming – components must be implemented by programmers, outside of the DGS, what can be allayed by developing

additional component library. In our experiments we noticed the students take a more active role when learning is supported with a DGS (Chapter 6).

We have developed extensions to the computational engine and expression tree definition of SLGeometry DGS, which demonstrate the following improvements:

- Provide support for objects (in the OOP sense) to represent geometric shapes and their properties; this concept is easily extended to supporting user interface elements in SLGeometry, and, in general, any object type that can be implemented in C# (Section 3.4);
- Provide a framework for simple definition and evaluation of object properties (Section 3.3, Chapter 4, Chapter 5);
- Define expression tree extensions and an algorithm for lazy evaluation of object properties, which reduce evaluation time by only evaluating properties that are referenced, and also by calculating common intermediate results only once (Section 5.4);
- Define a result caching scheme which can significantly reduce heap load and speed up rapid consecutive evaluation of expression trees (Section 5.3.1);
- Provide support for context-aware code completion, therefore alleviating complexity and effort often associated with textual input of expressions.

The extensions are implemented on SLGeometry, dynamic geometry software developed as Universal Windows Application in C# on .NET 4.5. We have tested our approach to expression evaluation and compared it to a purely functional evaluation scheme, which is regarded as standard in today's state-of-the-art dynamic geometry software (Chapter 7). CPU time, heap load and RAM consumption were measured and compared.

From experimental results, the proposed algorithm for lazy evaluation gives very good results considering speed, while memory consumption and heap load are also satisfactory. We have found that our approach is comparable with the functional evaluation scheme, having a better CPU time with only a marginal increase in memory footprint (Section 7.4).

The main contribution of this dissertation is the proposed system for efficient management of a set of interactive objects with dynamic properties, which is implemented as an extension of a classical expression tree evaluator. Computational efficiency is provided by the introduced caching scheme, dynamic property activation and lazy property evaluation. From the software development point of view, our proposal enables developers to encapsulate all required metadata and functionality in one source file for each object type. This is a step forward from the functional implementations, where the introduction of a new type means that a number of functions must also be introduced or overloaded in order to support that type. Our solution supports efficient manipulation of complex objects with numerous properties. The presented solution can also be applied to other functional expression evaluators.

8.2 Future Work

Available metadata for registered operations and functions allows for various optimizations of expression trees. One of the most interesting optimizations would be early binding of all operations, which are applied to operands of known types. The generalized binary operation from Listing 5.19 would require a new recursive structure to be created, which would substitute calls to the `BinaryOpLateBinding` method with actual operation evaluation methods, for all operands whose types are known beforehand. This structure would then be held in place of the actual expression tree and evaluated more efficiently.

Considering the available information about return types of functions, parts of expression trees can be identified, which are good candidates for compilation. For example, a subtree which only consists of numerical constants and functions returning numerical values can easily be compiled into more efficient code. By extrapolating this principle, a skeleton compiler can be written, which could be populated with metadata and separate compiling methods for operations and functions applied to arbitrary data types. Implementers of new data types, operations and functions would only need to write specific compilation methods and register them with metadata, in order for their data types, operations and functions to take part in compilation. The work on a generalized expression tree compiler is presently in progress.

Currently, the all components which are contained in `SLGeometry` framework are mutually dependent, with attempts to make them isolated and autonomous as stand alone framework, and ready for use to other developers.

Bibliography

- Abramovich, S (2013). "Computers in Mathematics Education: An Introduction". In: *Computers in the Schools: Interdisciplinary Journal of Practice, Theory and Applied Research* 30.1-2, pp. 4–11. DOI: [10.1080/07380569.2013.765305](https://doi.org/10.1080/07380569.2013.765305).
- Agarwal, V V (2013). *Using Attributes With C#.NET*. [Online; accessed 02-June-2019]. URL: <https://www.c-sharpcorner.com/UploadFile/84c85b/using-attributes-with-C-Sharp-net/>.
- Agda (2019). [Online; accessed 02-June-2019]. URL: <https://wiki.portal.chalmers.se/agda/pmwiki.php>.
- Aggarwa, CC (2011). *Social Network Data Analytics*. Springer. DOI: [10.1007/978-1-4419-8462-3](https://doi.org/10.1007/978-1-4419-8462-3).
- Aizikovitsh-Udi, E and N Radakovic (2012). "Teaching Probability by Using Geogebra Dynamic Tool and Implementing Critical Thinking Skills". In: *Procedia - Social and Behavioral Sciences* 46. 4th WORLD CONFERENCE ON EDUCATIONAL SCIENCES (WCES-2012) 02-05 February 2012 Barcelona, Spain, pp. 4943–4947. ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2012.06.364>. URL: <http://www.sciencedirect.com/science/article/pii/S1877042812021003>.
- Aktümen, M, T Horzum, and T Ceylan (2013). "Modeling and Visualization Process of the Curve of Pen Point by GeoGebra". In: *European Journal of Contemporary Education* 4.2, pp. 88–99. DOI: [10.13187/ejced.2013.4.88](https://doi.org/10.13187/ejced.2013.4.88). URL: http://ejournal11.com/journals_n/1373090541.pdf.
- Aldrich, J, C Chambers, and D Notkin (2002). "Architectural reasoning in ArchJava". In: *European Conference on Object-Oriented Programming*. Springer, pp. 334–367.
- Alvi, S (2002). *CodeProject - Attributes in C#*. [Online; accessed 02-June-2019]. URL: <https://www.codeproject.com/articles/2933/attributes-in-c>.
- Alvino, S et al. (2009). "An Exploratory Analysis of Subject Metadata in the Digital Public Library of America". In: *Proceedings of ICWL: Advances in Web Based Learning – ICWL 2009, Springer, Berlin, Heidelberg*. Vol. 5686, pp. 58–67. DOI: [10.1007/978-3-642-03426-8_7](https://doi.org/10.1007/978-3-642-03426-8_7).
- Anwar and I Rofiki (2018). "Investigating students' learning trajectory: a case on triangle". In: *Journal of Physics: Conference Series* 1088, p. 012021. DOI: [10.1088/1742-6596/1088/1/012021](https://doi.org/10.1088/1742-6596/1088/1/012021). URL: <https://iopscience.iop.org/article/10.1088/1742-6596/1088/1/012021/pdf>.
- Arbab, F (2005). "Abstract behavior types: a foundation model for components and their composition". In: *Science of Computer Programming* 55.1-3, pp. 3–52.
- Ariola, Z M et al. (1995). "A Call-by-need Lambda Calculus". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. New York, NY, USA: ACM, pp. 233–246. ISBN: 0-89791-692-1. DOI: [10.1145/199448.199507](https://doi.org/10.1145/199448.199507). URL: <http://doi.acm.org/10.1145/199448.199507>.

- Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali (1989). "I-structures: Data Structures for Parallel Computing". In: *ACM Trans. Program. Lang. Syst.* 11.4, pp. 598–632. ISSN: 0164-0925. DOI: [10.1145/69558.69562](https://doi.org/10.1145/69558.69562). URL: <http://doi.acm.org/10.1145/69558.69562>.
- Arzarello, F et al. (2002). "A cognitive analysis of dragging practises in Cabri environments". In: *ZDM: the international journal on mathematics education* 34.3, pp. 66–72. DOI: <http://dx.doi.org/10.1007/BF02655708>.
- Augustsson, L (1984). "A Compiler for Lazy ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. New York, NY, USA: ACM, pp. 218–227. ISBN: 0-89791-142-3. DOI: [10.1145/800055.802038](https://doi.org/10.1145/800055.802038). URL: <http://doi.acm.org/10.1145/800055.802038>.
- Backus, J (1978). "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *Commun. ACM* 21.8, pp. 613–41. DOI: <http://dx.doi.org/10.1145/359576.359579>.
- Barzilay, E and J Clements (2005). "Laziness Without All the Hard Work: Combining Lazy and Strict Languages for Teaching". In: *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education*. FDPE '05. New York, NY, USA: ACM, pp. 9–13. ISBN: 1-59593-067-1. DOI: [10.1145/1085114.1085118](https://doi.org/10.1145/1085114.1085118).
- Bass, Len, Paul Clements, and Rick Kazman (2012). *Software Architecture in Practice*. 3rd. Addison-Wesley Professional. ISBN: 0321815734, 9780321815736.
- Benton, N, L Cardelli, and C Fournet (2004). "Modern concurrency abstractions for C#". In: *ACM Transactions on Programming Languages and Systems* 26.5, pp. 769–804. ISSN: 0164-0925. DOI: [10.1145/1018203.1018205](https://doi.org/10.1145/1018203.1018205).
- Berzal, F et al. (2005). "Fuzzy Object-Oriented Modelling with Metadata Attributes in C#". In: *Computational Intelligence, Theory and Applications, International Conference 8th Fuzzy Days in Dortmund, Germany*. Springer Berlin Heidelberg, pp. 253–62.
- Bhagat, K K and C Chang (2015). "Incorporating GeoGebra into Geometry Learning- A lesson from India". In: *Eurasia Journal of Mathematics, Science & Technology Education* 11.1, pp. 77–86. ISSN: 1305 - 8223.
- Bidinger, P et al. (2005). "Dream types: a domain specific type system for component-based message-oriented middleware". In: *International Conference on Software Engineering: Proceedings of the 2005 conference on Specification and verification of component-based systems*. Vol. 5. 06.
- Bird, Richard and Philip Wadler (1988). *An Introduction to Functional Programming*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd. ISBN: 0-13-484189-1. URL: https://usi-pl.github.io/lc/sp-2015/doc/Bird_Wadler.%20Introduction%20to%20Functional%20Programming.1ed.pdf.
- Bloss, A, P Hudak, and J Young (1988). "Code optimizations for Lazy Evaluation". In: *Lisp and Symbolic Computation* 1.2, pp. 147–64.
- Blum, W and R Borromeo Ferri (2009). "Mathematical Modelling: Can It Be Taught And Learnt?" In: *Journal of Mathematical Modelling and Application* 1.1, pp. 45–58.
- Blum, W and M Niss (1991). "Applied mathematical problem solving, modeling, applications, and links to other subjects – state, trends and issues in mathematics instruction". In: *Educational Studies in Mathematics*, Kluwer Academic Publishers, Dordrecht 22, pp. 37–68.

- Bose, D (2010). *Component Based Development*. [Online; accessed 02-June-2019]. URL: <https://arxiv.org/ftp/arxiv/papers/1011/1011.2163.pdf>.
- Botana, F and M A Abánades (2014). "Automatic deduction in (dynamic) geometry: Loci computation". In: *Computational Geometry* 47.1, pp. 75–89. ISSN: 0925-7721. DOI: <https://doi.org/10.1016/j.comgeo.2013.07.001>. URL: <http://www.sciencedirect.com/science/article/pii/S0925772113000928>.
- Botana, F and J.L. Valcarce (2001). "Cooperation between a Dynamic Geometry Environment and a Computer Algebra System for Geometric Discovery". In: *CASC 2001*. Ed. by Vorozhtsov E.V. (eds) Ganzha V.G., Mayr E.W. Springer, Berlin, Heidelberg, pp. 63–74. DOI: https://doi.org/10.1007/978-3-642-56666-0_5.
- Botana, F et al. (2015). "Automated Theorem Proving in GeoGebra: Current Achievements". In: *J. Autom. Reason.* 55.1, pp. 39–59. ISSN: 0168-7433. DOI: [10.1007/s10817-015-9326-4](https://doi.org/10.1007/s10817-015-9326-4). URL: <http://dx.doi.org/10.1007/s10817-015-9326-4>.
- Bourque, P and R E Fairley (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press. ISBN: 0769551661, 9780769551661. URL: www.swebok.org.
- Bruneton, E et al. (2004). "An open component model and its support in java". In: *International Symposium on Component-based Software Engineering*. Springer, pp. 7–22.
- Buriková, S, W Steingartner, and M.A.M. Eldojali (2016). "A Foundation Towards the Classification of Categories for Component Oriented Programming". In: *EEI 7: proceedings of the Faculty of Electrical Engineering and Informatics of the TUKE*, pp. 20–5.
- Burkhardt, H (2013). "Curriculum design and systemic change". In: *Mathematics curriculum in school education*, pp. 13–34. DOI: https://doi.org/10.1007/978-94-007-7560-2_2.
- (2018). "Towards Research-based Education". In: *Shell Centre for Mathematical Education Publications Ltd*. Pp. 1–25. URL: https://www.mathshell.com/papers/pdf/hb_2018_research_based_education.pdf.
- Cabri (2019). [Online; accessed 02-June-2019]. URL: <http://cabri.com/en/>.
- Cabrilog: Echantillon de Cabri Factory – Parallélogramme (2019). [Online; accessed 02-June-2019]. URL: <http://y2u.be/0Gd8U87G41g>.
- Caligaris, M G, M E Schivo, and M R Romiti (2015). "Calculus & GeoGebra, an Interesting Partnership". In: *Procedia - Social and Behavioral Sciences* 174. International Conference on New Horizons in Education, INTE 2014, 25-27 June 2014, Paris, France, pp. 1183–1188. ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2015.01.735>. URL: <http://www.sciencedirect.com/science/article/pii/S1877042815007879>.
- Casero, R, M Cesarini, and M Monga (2003). "Managing Code Dependencies in C#". In: *J Obj Techol* 3.2, pp. 47–55.
- C# - Attributes (2019). [Online; accessed 02-June-2019]. URL: https://www.tutorialspoint.com/csharp/csharp_attributes.htm.
- Cazzola, W and E Vacchi (2014). "Java: Bringing a richer annotation model to Java". In: *Computer Languages, Systems & Structures* 40.1. Special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing, pp. 2–18. ISSN: 1477-8424. DOI: <https://doi.org/10.1016/j.cl.2014.02.002>.

- Cepa, V and M Mezini (2004). "Declaring and Enforcing Dependencies Between .NET Custom Attributes". In: *Generative Programming and Component Engineering*. Ed. by G Karsai and E Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 283–297. ISBN: 978-3-540-30175-2. DOI: [10.1007/978-3-540-30175-2_15](https://doi.org/10.1007/978-3-540-30175-2_15).
- Chang, S (2013). "Laziness by Need". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 81–100. ISBN: 978-3-642-37036-6. DOI: [10.1007/978-3-642-37036-6_5](https://doi.org/10.1007/978-3-642-37036-6_5). URL: https://link.springer.com/content/pdf/10.1007%2F978-3-642-37036-6_5.pdf.
- Chen, X et al. (2007). "A Model of Component-Based Programming". In: *International Symposium on Fundamentals of Software Engineering*. Ed. by F Arbab and M Sirjani. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 191–206. DOI: [10.1007/978-3-540-75698-9_13](https://doi.org/10.1007/978-3-540-75698-9_13).
- Chlipala, A (2016). "Ur: Statically-Typed Metaprogramming with Type-Level Record Computation". In: *CACM* 59.8, pp. 93–100. DOI: [10.1145/2958736](https://doi.org/10.1145/2958736).
- Chodarev, S and J Kollar (2016). "Extensible Host Language for Domain-Specific Languages". In: *Comp Inf* 35.1, pp. 84–110. URL: <http://www.cai.sk/ojs/index.php/cai/article/view/1507/745>.
- Chodarev, Sergej et al. (2014). "Abstract syntax driven approach for language composition". In: *Cen Eu J Comput Sci* 4.3, pp. 107–17. DOI: [10.2478/s13537-014-0211-8](https://doi.org/10.2478/s13537-014-0211-8).
- Chou, S C, X Gao, and J-Z Zhang (1994). *Machine proofs in geometry. Automated production of readable proofs for geometry theorems*. English. Vol. 6. Singapore: World Scientific, pp. xvii + 461. ISBN: 981-02-1584-3/hbk; 978-981-279-815-2/ebook.
- Cinderella Gallery: *Altitudes in a triangle* (2019). [Online; accessed 02-June-2019]. URL: <http://antique.cinderella.de/en/demo/gallery/altitudes.html>.
- Coquand, T and G Huet (1988). "The calculus of constructions". In: *Information and Computation* 76.2, pp. 95–120. ISSN: 0890-5401. DOI: [10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3). URL: <http://www.sciencedirect.com/science/article/pii/0890540188900053>.
- Crnkovic, I, M Chaudron, and S Larsson (2005). "Component-based Development Process and Component Lifecycle". In: *Journal of Computing and Information Technology* 13.4, pp. 321–327. DOI: [10.2498/cit.2005.04.10](https://doi.org/10.2498/cit.2005.04.10). URL: <http://cit.fer.hr/index.php/CIT/article/view/1586/1290>.
- Cut The Knot: The many ways to construct a triangle* (2019). [Online; accessed 02-June-2019]. URL: <http://www.cut-the-knot.org/triangle/>.
- Dagiene, V, T Jevsikova, and S Kubilinskiene (2013). "An Integration of Methodological Resources into Learning Object Metadata Repository". In: *Informatika* 24.1, pp. 13–34.
- Dahan, JJ (2014). "Morphing examples in 2D and 3D with TIN'Spire at all levels". In: *T3 Conference, Las Vegas*. URL: <https://www.youtube.com/watch?v=7tjnbBvnoiU>.
- DeJarnette, N (2012). "America's children: Providing early exposure to STEM (science, technology, engineering and math) initiatives". In: *Education* 133.1, pp. 77–84. DOI: [10.1007/s10649-005-9002-4](https://doi.org/10.1007/s10649-005-9002-4).
- Department for education, Mathematics programmes of study: key stages 1 and 2, National curriculum in England, September 2013* (2013). [Online; accessed 02-June-2019].

- URL: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/335158/PRIMARY_national_curriculum_-_Mathematics_220714.pdf.
- Diekmann, L and L Tratt (2014). "Eco: A Language Composition Editor". In: *International Conference on Software Language Engineering*. Springer, Cham, pp. 82–101. DOI: [10.1007/978-3-319-11245-9_5](https://doi.org/10.1007/978-3-319-11245-9_5).
- Diković, Lj (2009). "Applications GeoGebra into Teaching Some Topics of Mathematics at the College Level". In: *Computer Science and Information Systems 6.2*, pp. 191–203. DOI: [10.2298/csis0902191D](https://doi.org/10.2298/csis0902191D). URL: <http://www.comsis.org/pdf.php?id=138-0812>.
- Dimić, Bojana and Dušan Surla (2009). "XML Editor for UNIMARC and MARC 21 cataloguing". In: *The Electronic Library 27.3*, pp. 509–528.
- Drijvers, P et al. (2016). *Uses of Technology in Lower Secondary Mathematics Education*. 1st ed. 2366-5947. The address: Springer International Publishing. ISBN: 978-3-319-33666-4.
- Dutta, B, D Nandini, and G Shahi (2015). "MOD: Metadata for Ontology Description and Publication". In: *Proceedings of International Conference on Dublin Core and Metadata Applications*, pp. 1–9. ISBN: 1939-1366. URL: <http://dcpapers.dublincore.org/pubs/article/view/3758>.
- Eichberg, M, T Schäfer, and M Mezini (2005). "Using Annotations to Check Structural Properties of Classes". In: *Fundamental Approaches to Software Engineering*. Ed. by M Cerioli. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 237–252. ISBN: 978-3-540-31984-9. DOI: [10.1007/978-3-540-31984-9_18](https://doi.org/10.1007/978-3-540-31984-9_18).
- Engstrom, L (2001). "Investigate a Triangle - An Amount of Different Answers Using Cabri?" In: URL: <http://www.tact.fse.ulaval.ca/outils2/Cabri/2001/contributions/Engstrom.pdf>.
- Erdweg, S, PG Giarrusso, and T Rendel (2012). "Language Composition Untangled". In: *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 7:1–7:8. DOI: [10.1145/2427048.2427055](https://doi.org/10.1145/2427048.2427055).
- Erdweg, S et al. (2011). "SugarJ: Library-based Syntactic Language Extensibility". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pp. 391–406. DOI: [10.1145/2076021.2048099](https://doi.org/10.1145/2076021.2048099).
- Fabresse, L, C Dony, and M Huchard (2008). "Foundations of a simple and unified component-oriented language". In: *Computer Languages, Systems & Structures 34.2*, pp. 130–149. ISSN: 1477-8424. DOI: [10.1016/j.cl.2007.05.002](https://doi.org/10.1016/j.cl.2007.05.002).
- Freixas, Marc, Robert Joan-Arinyo, and Antoni Soto-Riera (2010). "A constraint-based dynamic geometry system". In: *Compr-Aid Des 42.2*, pp. 151–61. DOI: [http://dx.doi.org/10.1016/j.cad.2009.02.016](https://doi.org/10.1016/j.cad.2009.02.016).
- Friedman, DP and DS Wise (1976). "Technical Report TR44: CONS should not Evaluate its Arguments". In: *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh, pp. 257–84.
- Frost, R A and S Karamatos (1993). "Supporting the attribute grammar programming paradigm in a lazy functional programming language". In: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992 McMaster University, Hamilton, Ontario, Canada*. Ed. by Peter E

- Lauer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 278–295. ISBN: 978-3-540-47776-1. DOI: [10.1007/3-540-56883-2_13](https://doi.org/10.1007/3-540-56883-2_13).
- A review of the best pre-made interactive GeoGebra activities* (2016). Vol. 28. Electronic Proceedings of the Twenty-eighth Annual International Conference on Technology in Collegiate Mathematics. Atlanta, Georgia. URL: <http://archives.math.utk.edu/ICTCM/VOL28/A015/paper.pdf>.
- Gelernter, H. (1963). “Computers & Thought”. In: ed. by Edward A. Feigenbaum and Julian Feldman. Cambridge, MA, USA: MIT Press. Chap. Realization of a Geometry-theorem Proving Machine, pp. 134–152. ISBN: 0-262-56092-5. URL: <https://pdfs.semanticscholar.org/2edc/8083073837564306943aab77d6dcc19d0cdc.pdf>.
- (1995). “Computers & Thought”. In: ed. by Edward A. Feigenbaum and Julian Feldman. Cambridge, MA, USA: MIT Press. Chap. Realization of a Geometry-theorem Proving Machine, pp. 134–152. ISBN: 0-262-56092-5. URL: <http://dl.acm.org/citation.cfm?id=216408.216418>.
- Gelernter, H., J. R. Hansen, and D. W. Loveland (1960). “Empirical Explorations of the Geometry Theorem Machine”. In: *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '60 (Western). New York, NY, USA: ACM, pp. 143–149. DOI: [10.1145/1460361.1460381](https://doi.org/10.1145/1460361.1460381). URL: <https://pdfs.semanticscholar.org/2edc/8083073837564306943aab77d6dcc19d0cdc.pdf>.
- GeoGebra* (2019). [Online; accessed 02-June-2019]. URL: <https://www.geogebra.org/>.
- GeoGebra Materials* (2019). [Online; accessed 02-June-2019]. URL: <https://www.geogebra.org/materials/>.
- GeoGebra materials: triangle* (2019). [Online; accessed 02-June-2019]. URL: <https://www.geogebra.org/search/perform/search/triangle/materials/>.
- Geometrie interactive - le cercle qui tourne le triangle* (2019). [Online; accessed 02-June-2019]. URL: https://youtu.be/iDsjZThy_4s (visited on 04/17/2018).
- Gilst, F A van and P M van den Broek (1995). “A New Programming Technique for Lazy Functional Languages”. In: *Sci. Comput. Program.* 24.1, pp. 63–81. ISSN: 0167-6423. DOI: [10.1016/0167-6423\(94\)00024-9](https://doi.org/10.1016/0167-6423(94)00024-9). URL: [http://dx.doi.org/10.1016/0167-6423\(94\)00024-9](http://dx.doi.org/10.1016/0167-6423(94)00024-9).
- Ginsburg, D et al. (1997). *The History of the Calculus and the Development of Computer Algebra Systems*. [Online; accessed 02-June-2019]. URL: <http://www.math.wpi.edu/IQP/BVCalcHist/calc5.html>.
- Gonzalez, H B and J J Kuenzi (2012). *Science, technology, engineering, and mathematics (STEM) education: A primer*. URL: <https://fas.org/sgp/crs/misc/R42642.pdf>.
- Granström, J G (2012). “A new paradigm for component-based development”. In: *Journal of Software* 7.5, pp. 1136–1148.
- Greaves, D and S Singh (2008). “Using C# Attributes to Describe Hardware Artefacts within Kiwi”. In: *Specification, Verification and Design Languages*. IEEE, pp. 239 – 40.
- Guerra, E and C Fernandes (2013). “A Qualitative and Quantitative Analysis on Metadata-Based Frameworks Usage”. In: *Computational Science and Its Applications – ICCSA 2013, Lecture Notes in Computer Science* 7972, pp. 375–90. DOI: [10.1007/978-3-642-39643-4_28](https://doi.org/10.1007/978-3-642-39643-4_28).

- Guerra, E, C Fernandes, and F Fagundes Silveira (2010). "Architectural Patterns for Metadata-based Frameworks Usage". In: *Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP '10)*, ACM, New York, NY, USA, 4:1–4:25. DOI: [10.1145/2493288.2493292](https://doi.org/10.1145/2493288.2493292).
- Guzzi, P H (2019). "Ontology-Based Annotation Methods". In: *Encyclopedia of Bioinformatics and Computational Biology*. Ed. by S Ranganathan et al. Oxford: Academic Press, pp. 867 –869. ISBN: 978-0-12-811432-2. DOI: <https://doi.org/10.1016/B978-0-12-809633-8.20400-7>.
- Hall, J and G Chamblee (2013). "Teaching Algebra and Geometry with GeoGebra: Preparing Pre-Service Teachers for Middle Grades/Secondary Mathematics Classrooms". In: *Computers in the Schools: Interdisciplinary Journal of Practice, Theory, and Applied Research* 30, pp. 12–29. DOI: [10.1080/07380569.2013.764276](https://doi.org/10.1080/07380569.2013.764276).
- Han, O B et al. (2013). "Computer Based Courseware in Learning Mathematics: Potentials and Constrains". In: *Procedia - Social and Behavioral Sciences* 103, pp. 238 –244. ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2013.10.331>. URL: <http://www.sciencedirect.com/science/article/pii/S1877042813037762>.
- Hazzard, K and J Bock (2013). *Metaprogramming in .NET*. Manning Publ, pp. 1–360. ISBN: 9781617290268. URL: <https://www.manning.com/books/metaprogramming-in-dot-net>.
- Heineman, G T and W T Councill, eds. (2001). *Component-based Software Engineering: Putting the Pieces Together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-70485-4.
- Henderson, P and JH Morris (1976). "A lazy evaluator". In: *Proceeding POPL '76 Proc. of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, ACM New York, USA 1976, pp. 95–103. DOI: [doi:10.1145/800168.811543](https://doi.org/10.1145/800168.811543).
- Herceg, Đ and V Herceg-Mandić (2013). "GeoGebra in a geography class". In: *Acta Didactica Napocensia* 6.1, pp. 61–68. URL: <https://files.eric.ed.gov/fulltext/EJ1053672.pdf>.
- Herceg, Đ, V Herceg-Mandić, and D Radaković (2012). "The Teaching of Geography Using Dynamic Geometry Software". In: *Local Proceedings of the Fifth Balkan Conference in Informatics , BCI2012*, pp. 11–5. URL: <http://ceur-ws.org/Vol-920/p11-herceg.pdf>.
- Herceg, Đ and D Radaković (2011). "The Extensibility of an Interpreted Language Using Plugin Libraries". In: *Numerical Analysis and Applied Mathematics ICNAAM 2011, AIP Conf. Proc. 2011*. Vol. 1389, pp. 837–40.
- Herceg, Đ, D Radaković, and D Herceg (2012). "Generalizing the Extensibility of a Dynamic Geometry Software". In: *Numerical Analysis and Applied Mathematics ICNAAM 2012, AIP Conf. Proc. 2012*. Vol. 1479, pp. 482–5.
- Herceg, Đ et al. (2019). "Subject-specific components in dynamic geometry software". In: *International Journal for Technology in Mathematics Education* 26.2, pp. 97–102. ISSN: 1744–2710. DOI: [10.1564/tme_v26.2.07](https://doi.org/10.1564/tme_v26.2.07).
- Herceg, D and Đ Herceg (2009). "The Definite Integral and Computer". In: *The Teaching of Mathematics* 12.1, pp. 33–44. URL: <http://elib.mi.sanu.ac.rs/files/journals/tm/22/tm1215.pdf>.
- Hohenwarter, J et al. (2009). "Introducing dynamic mathematics software to secondary school teachers: The case of GeoGebra". In: *The Journal of Computers in*

- Mathematics and Science Teaching* 28.2, pp. 135–46. DOI: 10.14221/ajte.2013v38n12.6. URL: https://archive.geogebra.org/static/publications/2009-Hohenwarter_Lavicza_IntroducingDynMathSoft-GeoGebra.pdf.
- Hohenwarter, M and J Preiner (2007). “Dynamic mathematics with GeoGebra”. In: *Journal of Online Mathematics and its Applications* 7. URL: https://www.maa.org/external_archive/joma/Volume7/Hohenwarter/index.html.
- Hohenwarter, M et al. (2008). “Teaching and learning calculus with free dynamic mathematics Software GeoGebra”. In: *Proceeding of International Conference in Mathematics Education, ICME 11, Monterrey, Mexico*. URL: <https://archive.geogebra.org/static/publications/2008-ICME-TSG16-Calculus-GeoGebra-Paper.pdf>.
- Seldin, J P, Lambda Calculus Hindley J R: To H.B. Curry: Essays on Combinatory Logic, and Formalism, eds. (1980). *The formulae-as-types notion of construction*. (1969) unpublished manuscript. Academic Press, New York, 479—490. URL: <https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf>.
- Hudak, Paul et al. (2007). “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, pp. 12–1–12–55. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856. URL: <http://doi.acm.org/10.1145/1238844.1238856>.
- Hughes, J (1990). “Why Functional Programming Matters”. In: *In D. Turner, editor, Research Topics in Functional Programming, Addison Wesley*, pp. 17–42.
- Jackiw, N and W Finzer (1993). “Watch What I Do”. In: ed. by Allen Cypher et al. Cambridge, MA, USA: MIT Press. Chap. The Geometer’s Sketchpad: Programming by Geometry, pp. 293–307. ISBN: 0-262-03213-9.
- Jahn, M et al. (2013). “Composing User-specific Web Applications From Distributed Plug-ins”. In: *Comput Sci - Res Dev* 28.85, pp. 1–21. DOI: 10.1007/s00450-011-0182-0.
- Janićić, P, J Narboux, and P Quaresma (2012). “The Area Method - A Recapitulation”. In: *J. Autom. Reasoning* 48.4, pp. 489–532. DOI: 10.1007/s10817-010-9209-7. URL: <https://doi.org/10.1007/s10817-010-9209-7>.
- Janićić, Predrag (2010). “Geometry constructions language”. In: *Journal of Automated Reasoning* 44.1-2, p. 3.
- Janićić, Predrag (2019). *GCLC, Mathematical Tool GCLC (Geometry Constructions -> LaTeX Converter)*. [Online; accessed 02-June-2019]. URL: <http://poincare.matf.bg.ac.rs/~janicic/gclc/>.
- Janićić, Predrag (2010). “Geometry Constructions Language”. In: *J. Autom. Reason.* 44.1-2, pp. 3–24. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9135-8. URL: <http://dx.doi.org/10.1007/s10817-009-9135-8>.
- Jezdimirović, J (2014). “Computer Based Support for Mathematics Education in Serbia”. In: *International Journal of Technology and Inclusive Education (IJTIE)* 3.1, pp. 277–285. ISSN: 2046-4568. DOI: 10.20533/ijtie.2047.0533.2014.0036. URL: <https://infonomics-society.org/wp-content/uploads/ijtie/volume-3-2014/Computer-Based-Support-for-Mathematics-Education-in-Serbia.pdf>.
- Johnsson, T (1984). “Efficient Compilation of Lazy Evaluation”. In: *SIGPLAN Not.* 19.6, pp. 58–69. ISSN: 0362-1340. DOI: 10.1145/502949.502880. URL: <http://>

- citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.4238&rep=rep1&type=pdf.
- Joosten, S, K Van Den Berg, and G Van Der Hoeven (1993). "Teaching functional programming to first-year students". In: *Journal of Functional Programming* 3.1, 49–65. DOI: [10.1017/S0956796800000599](https://doi.org/10.1017/S0956796800000599).
- Kashefi, H, Z Ismail, and Y M Yusof (2012). "Supporting Engineering Students' Thinking and Creative Problem Solving through Blended Learning". In: *Procedia - Social and Behavioral Sciences* 56, pp. 117–125. ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2012.09.638>. URL: <http://www.sciencedirect.com/science/article/pii/S1877042812041006>.
- Kaufmann, H and D Schmalstieg (2002). "Mathematics and Geometry Education with Collaborative Augmented Reality". In: *ACM SIGGRAPH 2002 Conference Abstracts and Applications*. SIGGRAPH '02. San Antonio, Texas: ACM, pp. 37–41. ISBN: 1-58113-525-4. DOI: [10.1145/1242073.1242086](https://doi.org/10.1145/1242073.1242086). URL: <http://doi.acm.org/10.1145/1242073.1242086>.
- Kendall, A (2016). *Metadata-Driven Design: Creating an User-Friendly Enterprise DSL*. [Online; accessed 02-June-2019]. URL: <https://www.infoq.com/articles/mdd-creating-user-friendly-dsl1>.
- Kent Recursive Calculator* (2019). [Online; accessed 02-June-2019]. URL: <http://krc-lang.org/>.
- Khalil, M et al. (2018). "The Development of Mathematical Achievement in Analytic Geometry of Grade-12 Students through GeoGebra Activities". In: *Eurasia Journal of Mathematics, Science and Technology Education* 14.4, pp. 1453–63. ISSN: 1305 - 8223. DOI: <http://dx.doi.org/10.29333/ejmste/83681>.
- Kimberling, C (1993). "Triangle centers as functions". In: *Journal of Mathematics* 23.4, pp. 1269–86.
- (2019). *Clark Kimberling's Encyclopedia of Triangle Centers – ETC*. [Online; accessed 02-June-2019]. URL: <http://faculty.evansville.edu/ck6/encyclopedia/ETC.html>.
- Kiselyov, O, S Peyton-Jones, and A Sabry (2012). "Lazy v. Yield: Incremental, Linear Pretty-Printing". In: *Programming Languages and Systems, Lecture Notes in Computer Science* 7705, pp. 190–206. DOI: [10.1007/978-3-642-35182-2_14](https://doi.org/10.1007/978-3-642-35182-2_14). URL: <http://okmij.org/ftp/continuations/PPYield/yield-pp.pdf>.
- Knuth, DE (1968). "Semantics of context-free languages". In: *Mathematical Systems Theory* 2.2, pp. 127–45. ISSN: 0025-5661. DOI: [10.1007/BF01692511](https://doi.org/10.1007/BF01692511).
- (1971). "Semantics of context-free languages: correction". In: *Mathematical Systems Theory* 5.2, pp. 95–6. ISSN: 0025-5661. DOI: [10.1007/BF01702865](https://doi.org/10.1007/BF01702865).
- Kortenkamp, U (1999). "Foundations of dynamic geometry". PhD thesis. Swiss Federal Institute of Technology Zurich. URL: <https://www.research-collection.ethz.ch/mapping/eserv/eth:23347/eth-23347-02.pdf>.
- Laborde, C and B Capponi (1994). "Cabri-géomètre constituant d'un milieu pour l'apprentissage de la notion de figure géométrique". In: *Recherches en didactique des mathématiques* 14.1.2, pp. 165–210. URL: <https://rdm.penseesauvage.com/Cabri-geometre-constituant-d-un.html>.
- Lakos, J (1996). *Large-scale C++ Software Design*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0-201-63362-0.

- Lavicza, Z et al. (2018). "Mathematics Learning Through Arts, Technology and Robotics: Multi-and Transdisciplinary Steam Approaches". In: *8th ICMI-East Asia Regional Conference on Mathematics Education 7-11 May 2018, Taipei, Taiwan*, pp. 110–122.
- Lehrplan PLUS Mittelschule – Juni 2016, Bayerisches Staatsministerium für Bildung und Kultus, Wissenschaft und Kunst, Staatsinstitut für Schulqualität und Bildungsforschung München (2016). [Online; accessed 02-June-2019]. URL: <http://www.lehrplanplus.bayern.de/sixcms/media.php/107/LehrplanPLUS%20Mittelschule%20-%20Juni%202016.pdf>.
- Lernpfad, SF (2019). *S. Fink. Lernpfad: Mathe online - Points, lines and circles associated with a triangle*. [Online; accessed 02-June-2019]. URL: http://www.mathe-online.at/lernpfade/triangle_fink/?kapitel=1.
- Lilis, Y and A Savidis (2015). "An Integrated Implementation Framework for Compile-time Metaprogramming". In: *Softw Pract Exper* 45, pp. 727–63. DOI: 10.1002/spe.
- Ljajko, E and V Ibro (2013). "Development of ideas in a GeoGebra – aided mathematics instruction". In: *Mevlana International Journal of Education (MIJE)* 3.3, pp. 1–7. DOI: 10.13054/mije.si.2013.01. URL: <https://files.eric.ed.gov/fulltext/ED544150.pdf>.
- Löberbauer, M et al. (2010). "Testing the Composability of Plug-and-Play Components". In: *8th IEEE International Symposium on Intelligent Systems and Informatics, SISY 2010*, pp. 413–8. DOI: 10.1109/SISY.2010.5647368. URL: http://ase.jku.at/publications/2010/SISY2010_Composability_Testing.pdf.
- Maeder, R (1993). "The Mathematica Programmer: Object-Oriented Programming". In: *The Mathematica Journal* 3.1, pp. 23–31.
- Mainali, B R and M B Key (2012). "Using dynamic geometry software GeoGebra in developing countries: A case study of impressions of mathematics teachers in Nepal". In: *International Journal for Mathematics Teaching and Learning*, pp. 1–16. ISSN: 1473 - 0111. URL: <http://www.cimt.org.uk/journal/mainali.pdf>.
- Marinković, V (2015). "On-line compendium of triangle construction problems with automatically generated solutions". In: *The Teaching of Mathematics* 18.1, pp. 29–44. URL: <http://www.teaching.math.rs/vol/tm1814.pdf>.
- Marinković, V (2016). "ArgoTriCS – automated triangle construction solver". In: *Journal of Experimental & Theoretical Artificial Intelligence*, pp. 1–25. DOI: <http://dx.doi.org/10.1080/0952813X.2015.1132271>.
- Marinković, Vesna and Predrag Janičić (2012). "Towards Understanding Triangle Construction Problems". In: *Proceedings of the 11th International Conference on Intelligent Computer Mathematics. CICM'12*. Bremen, Germany: Springer-Verlag, pp. 127–142. ISBN: 978-3-642-31373-8. DOI: 10.1007/978-3-642-31374-5_9.
- Maskeliūnas, R et al. (2018). "IDO: modelling a serious educational game based on hands on approach for training dementia carers". In: *International Journal of Engineering & Technology* 7.2.28, pp. 143–146. ISSN: 2227-524X. DOI: 10.14419/ijet.v7i2.28.12898. URL: <https://www.sciencepubco.com/index.php/ijet/article/view/12898>.
- Math Open Reference, Triangles* (2019). [Online; accessed 02-June-2019]. URL: <http://www.mathopenref.com/tocs/triangletoc.html>.
- MathWorks - Computer Algebra System* (2019). [Online; accessed 02-June-2019]. URL: <https://www.mathworks.com/discovery/computer-algebra-system.html>.

- McAuley, J and J Leskovec (2012). "Image Labeling on a Network: Using Social-Network Metadata for Image Classification". In: *European Conference on Computer Vision*, pp. 828–41. URL: <http://cs.stanford.edu/people/jure/pubs/image-eccv12.pdf>.
- Mernik, M (2013). "An object-oriented approach to language compositions for software language engineering". In: *J Syst Softw* 86, pp. 2451–64. DOI: [10.1016/j.jss.2013.04.087](https://doi.org/10.1016/j.jss.2013.04.087).
- Milner, R (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3, pp. 257–84. URL: <https://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/milner-type-polymorphism.pdf>.
- Montes, A and M Wibmer (2014). "Software for Discussing Parametric Polynomial Systems: The Gröbner Cover". In: *Mathematical Software – ICMS 2014*. Ed. by H Hong and C Yap. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 406–413. ISBN: 978-3-662-44199-2.
- Mössenböck, Hanspeter (2010). *The Compiler Generator Coco/R -User Manual*. University of Linz, pp. 1–42. URL: <http://www.ssw.uni-linz.ac.at/coco/Doc/UserManual.pdf>.
- Mott, J et al. (2008). "Making a Significant Difference: A Goal-Driven Approach to Improving Teaching & Learning with Technology". In: *2008 Annual Proceedings - Orlando, On the Practice of Educational Communications and Technology*. Vol. 2. URL: https://members.aect.org/pdf/Proceedings/proceedings08/2008I/08_16.pdf.
- MSDN (2019). *Creating Custom Attributes*. [Online; accessed 02-June-2019]. URL: <https://msdn.microsoft.com/en-us/library/sw480ze8.aspx>.
- Mulansky, M and K Ahnert (2011). "Metaprogramming Applied to Numerical Problems". In: *AIP Conference Proceedings , Numerical Analysis and Applied Mathematics*, pp. 1582–5. DOI: [10.1063/1.3637933](https://doi.org/10.1063/1.3637933).
- Narbox, J (2007). "A Graphical User Interface for Formal Proofs in Geometry". In: *J Autom Reas* 39.2, pp. 161–80. DOI: <https://doi.org/10.1007/s10817-007-9071-4>.
- Nastavni plan i program za osnovnu školu 2013, Ministarstvo znanosti i sporta Republike Hrvatske* (2013). [Online; accessed 02-June-2019]. URL: <http://public.mzos.hr/figs.axd?id=20542>.
- Nastavni planovi i programi za osnovne i srednje škole, Zavod za unapređenje obrazovanja i vaspitanja, Republika Srbija* (2016). [Online; accessed 02-June-2019]. URL: <http://www.zuov.gov.rs/poslovi/nastavni-planovi/nastavni-planovi-os-i-ss/?lng=lat>.
- Nikhil, R S., K Pingali, and Arvind (1986). *Id Nouveau*. Tech. rep. AIM-349. MIT. URL: <http://csg.csail.mit.edu/pubs/memos/Memo-265/Memo-265.pdf>.
- Niss, M (2012). "Models and Modeling in Mathematics Education". In: *EMS Newsletter December 2012*, pp. 49–52. URL: http://www.euro-math-soc.eu/ems_education/Solid_Findings_Modelling.pdf.
- Noguera, C and L Duchien (2008). "Annotation Framework Validation Using Domain Models". In: *Model Driven Architecture – Foundations and Applications*. Ed. by I Schieferdecker and A Hartman. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 48–62. ISBN: 978-3-540-69100-6. DOI: [10.1007/978-3-540-69100-6_4](https://doi.org/10.1007/978-3-540-69100-6_4).

- Noguera, C and R Pawlak (2007). "AVal: An Extensible Attribute-oriented Programming Validator for Java". In: *Research Articles. J Softw Maint Evol* 19.4, pp. 253–75. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.220.5933&rep=rep1&type=pdf>.
- Nosál', M, M Sulír, and J Juhár (2016). "Language Composition Using Source Code Annotations". In: *Comput Sci Inf Syst* 13.3, pp. 707–29. DOI: [10.2298/CSIS160114024N](https://doi.org/10.2298/CSIS160114024N).
- Nosál', M, J Porubán, and M Sulír (2017). "Customizing host IDE for non-programming users of pure embedded DSLs: A case study". In: *Comput Lang Syst Struct* 49.Supplement C, pp. 101–18. ISSN: 1477-8424. DOI: <https://doi.org/10.1016/j.cl.2017.04.003>.
- Palmer, Zachary and Scott F. Smith (2011). "Backstage Java". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11* 46.10, p. 939. ISSN: 03621340. DOI: [10.1145/2048066.2048137](https://doi.org/10.1145/2048066.2048137).
- Pech, P (2012). "How integration of DGS and CAS helps to solve problems in geometry". In: *17th Asian Technology Conference in Mathematics, Suan Sunandha Rajabhat University, Bangkok, Thailand*. URL: http://atcm.mathandtech.org/EP2012/invited_papers/3472012_19796.pdf.
- Peelar, S (2016). "Accommodating prepositional phrases in a highly modular natural language query interface to semantic web triplestores using a novel event-based denotational semantics for English and a set of functional parser combinators". PhD thesis. Electronic Theses and Dissertations: University of Windsor. URL: <https://scholar.uwindsor.ca/etd/5911>.
- Pham, T M and Y Bertot (2012). "A Combination of a Dynamic Geometry Software With a Proof Assistant for Interactive Formal Proofs". In: *Electronic Notes in Theoretical Computer Science* 285. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10), pp. 43–55. ISSN: 1571-0661. DOI: <https://doi.org/10.1016/j.entcs.2012.06.005>. URL: <http://www.sciencedirect.com/science/article/pii/S1571066112000254>.
- Pinoli, P et al. (2019). "Metadata management for scientific databases". In: *Information Systems* 81, pp. 1–20. ISSN: 0306-4379. DOI: <https://doi.org/10.1016/j.is.2018.10.002>.
- Podwysocki, Matthew (2019). *Matthew Podwysocki's Blog: Object Oriented F# - Creating Classes*. [Online; accessed 02-June-2019]. URL: <https://weblogs.asp.net/podwysocki/object-oriented-creating-classes>.
- Pree, W (1997). "Component-based software development-a new paradigm in software engineering?" In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. IEEE, pp. 523–524.
- Prensky, M (2007). "How to Teach With Technology—keeping both teachers and students comfortable in an era of exponential change". In: *Emerging Technologies for Learning*. Vol. 2. Becta. Chap. 4, pp. 40–46. URL: https://www.calvin.edu/~dsc8/documents/emerging_technologies07.pdf.
- Progressions pour le cours préparatoire et le cours élémentaire première année: Mathématiques, Ressources pour l'école élémentaire* (2012). [Online; accessed 02-June-2019]. URL: http://cache.media.eduscol.education.fr/file/Progressions_

- [pedagogiques/79/2/Progression-pedagogique_Cycle2_Mathematiques_203792.pdf](#).
- Quaresma, P, V Santos, and N Baeta (2018). "Exchange of Geometric Information Between Applications". In: *6th International Workshop on Theorem proving components for Educational software*. EPTCS 267, pp. 108–119. DOI: [10.4204/EPTCS.267.7](#).
- Quaresma, P et al. (2008). "XML-Based Format for Description of Geometrical Constructions and Proofs". In: *Communicating Mathematics in the Digital Era*. CRC Press, Taylor and Francis Group, pp. 183–197. ISBN: 9781568814100. URL: <http://poincare.matf.bg.ac.rs/~milena/publications/xml.pdf>.
- Radaković, D and Đ Herceg (2010). "The Use of WPF for Development of Interactive Geometry Software". In: *Acta Universitatis Matthiae Belii, Series Mathematics* 16, pp. 65–79. URL: <http://actamath.savbb.sk/pdf/acta1606.pdf>.
- (2013). "A Platform for Development of Mathematical games on Silverlight". In: *Acta Didactica Napocensia* 6.1, pp. 77–90. URL: <https://files.eric.ed.gov/fulltext/EJ1053670.pdf>.
- (2017). "Metadata Specification in a Dynamic Geometry Software". In: *Proc. of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2016, 6th SCLIT*. American Institute of Physics, pp. 330006/1–5. DOI: [10.1063/1.4992504](#).
- (2018). "Towards a completely extensible dynamic geometry software with metadata". In: *Computer Languages, Systems and Structures* 52, pp. 1–20. ISSN: 1477 - 8424. DOI: [10.1016/j.cl.2017.11.001](#). URL: <http://www.sciencedirect.com/science/article/pii/S147784241730057X>.
- Radaković, D, Đ Herceg, and M Löberbauer (2010). "Extensible expression evaluator for the dynamic geometry software Geometrijska". In: *XVIII Conference on Applied Mathematics - PRIM 2009*, pp. 95–100. URL: http://ase.jku.at/publications/2009/PRIM09_095_RadakovicHercegLoeberbauer-p95-100.pdf.
- Radović, S, M Marić, and D Passey (2019). "Technology enhancing mathematics learning behaviours: Shifting learning goals from "producing the right answer" to "understanding how to address current and future mathematical challenges"". In: *Education and Information Technologies* 24.1, pp. 103–126. ISSN: 1573-7608. DOI: [10.1007/s10639-018-9763-x](#).
- Reflection* (C#) (2019). [Online; accessed 02-June-2019]. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>.
- Reis, Z A (2010). "Computer supported mathematics with Geogebra". In: *Procedia - Social and Behavioral Sciences* 9. World Conference on Learning, Teaching and Administration Papers, pp. 1449–1455. ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2010.12.348>. URL: <http://www.sciencedirect.com/science/article/pii/S1877042810024535>.
- Retrieving Information Stored in Attributes* (2019). [Online; accessed 02-June-2019]. URL: <https://docs.microsoft.com/en-us/dotnet/standard/attributes/retrieving-information-stored-in-attributes>.
- Ristić, S et al. (2014). "Generic and Standard Database Constraint Meta-Models". In: *Comput Sci Inf Syst* 11.2, pp. 679–96. DOI: <https://doi.org/10.2298/CSIS140216037R>.

- Rouvoy, R and P Merle (2006). "Leveraging Component-Oriented Programming with Attribute-Oriented Programming". In: *In Proc. of the 11th ECOOP International Workshop on Component-Oriented Programming*, pp. 10–8. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.4416&rep=rep1&type=pdf>.
- Sarama, J and D H Clements (2009). *Early Childhood Mathematics Education Research: Learning Trajectories for Young Children*. Studies in Mathematical Thinking and Learning Series. Taylor & Francis. ISBN: 9781135592509. URL: <https://books.google.rs/books?id=Z5K0AgAAQBAJ>.
- (2012). " 'Concrete' Computer Manipulatives in Mathematics Education". In: *Child Development Perspectives* 3.3, pp. 145–150. DOI: 10.1111/j.1750-8606.2009.00095.x. URL: https://www.du.edu/marsicoinstitute/media/documents/dc_concrete_computer_manipulatives.pdf.
- Savidis, A (2006). "Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-programming". In: *N. Guelfi and A. Savidis (Eds.): RISE 2005, LNCS 3943*. New York, NY, USA: ACM, pp. 113–28. DOI: https://doi.org/10.1007/11751113_9.
- Schoenfeld, A H (2014). "What Makes for Powerful Classrooms, and How Can We Support Teachers in Creating Them? A Story of Research and Practice, Productively Intertwined". In: *Educational Researcher* 43.8, pp. 404–412. DOI: 10.3102/0013189X14554450. URL: http://map.mathshell.org/trumath/schoenfeld_2014_ER.pdf.
- Schreck, P et al. (2016). "Wernick's List: A Final Update". In: *Forum Geometricorum* 16, pp. 69–80. URL: <http://icube-publis.unistra.fr/appli.php/2-SMMJ16>.
- Schult, W and A Polze (2002). "Aspect-Oriented Programming with C# and .NET". In: *Proceedings of International Symposium on Object-oriented Real-time distributed Computing, Crystal City, VA, USA*, pp. 241–8.
- Schwarz, D (2004). *Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5*. [Online; accessed 02-June-2019]. URL: <http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html>.
- Seco, J C, R Silva, and M Piriquito (2008). "Component J: A component-based programming language with dynamic reconfiguration". In: *Comput. Sci. Inf. Syst.* 5.2, pp. 63–86.
- Seger, C J H et al. (2005). "An Industrially Effective Environment for Formal Hardware Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.9, pp. 1381–1405. ISSN: 0278-0070. DOI: 10.1109/TCAD.2005.850814.
- Shepard, RN (1978). "The mental image". In: *American Psychologist* 33, pp. 125–137. DOI: 10.1037/0003-066X.33.2.125.
- Sinot, FR (2008). "Complete Laziness: a Natural Semantics". In: *El Not Theor Comput Sci* 204, pp. 129–45. DOI: 10.1016/j.entcs.2008.03.058.
- Siqueira, JL de, FF Silveira, and EM Guerra (2016). "An Approach for Code Annotation Validation with Metadata Location Transparency". In: *Computational Science and Its Applications - ICCSA 2016, Lecture Notes in Computer Science*. Vol. 9789. Springer, Cham, pp. 422–38. DOI: 10.1007/978-3-319-42089-9_30.
- Slodičák, V and P Macko (2011). "Some New Approaches in Functional Programming Using Algebras and Coalgebras". In: *Electronic Notes in Theoretical Computer*

- Science* 279.3, pp. 41–62. URL: https://ac.els-cdn.com/S1571066111001861/1-s2.0-S1571066111001861-main.pdf?_tid=e744506a-32b1-4f12-a7ba-04ecb0a1ab9e&acdnat=1548721948_d1572d1c01b3b3a78de9721cfe34ab0b.
- Song, M and E Tilevich (2015). “Reusing metadata across components, applications, and languages”. In: *Sci Comput Prog* 98.4, pp. 617–44. DOI: [10.1016/j.scico.2014.09.002](https://doi.org/10.1016/j.scico.2014.09.002).
- StackExchange – Mathematics* (2019). [Online; accessed 02-June-2019]. URL: <http://math.stackexchange.com/questions/361412/finding-the-angle-between-three-points>.
- Stein, W (2012). “Sage: Creating a Viable Free Open Source Alternative to Magma, Maple, Mathematica, and MATLAB”. In: *Foundations of Computational Mathematics, Budapest 2011*. Ed. by Felipe Cucker et al. London Mathematical Society Lecture Note Series. Cambridge University Press, 230–238. DOI: [10.1017/CB09781139095402.011](https://doi.org/10.1017/CB09781139095402.011). URL: <https://wstein.org/papers/focm11/focm11.pdf>.
- (2019a). *CoCalc - Collaborative Calculation in the Cloud*. [Online; accessed 02-June-2019]. URL: <http://cocalc.com/>.
- (2019b). *SageMath - open source mathematical software*. [Online; accessed 02-June-2019]. URL: <http://www.sagemath.org/>.
- Steingartner, W et al. (2016). “Some properties of coalgebras and their rôle in computer science”. In: *Journal of Applied Mathematics and Computational Mechanics* 16.3, pp. 145–56. ISSN: 2353-0588. DOI: [10.17512/jamcm.2016.4.16](https://doi.org/10.17512/jamcm.2016.4.16).
- Steketee, S (2010). “Comparison of Sketchpad and GeoGebra”. In: *Key Curriculum Press* 3. URL: https://s3.amazonaws.com/keycurriculum.com/PDF/Sketchpad/Detailed_Comparison_of_Sketchpad_and_GeoGebra.pdf.
- Štuikys, V, R Damaševičius, and G Ziberkas (2012). “Understanding of Heterogeneous Multi-Stage Meta-Programs”. In: *Inf Technol Cont* 41.1, pp. 23–32. DOI: [10.5755/j01.itc.41.1.916](https://doi.org/10.5755/j01.itc.41.1.916).
- Sulír, M, M Nosál’, and J Porubän (2016). “Recording concerns in source code using annotations”. In: *Comput Lang Syst Struct* 46, pp. 44–65. DOI: [10.1016/j.cl.2016.07.003](https://doi.org/10.1016/j.cl.2016.07.003).
- Sussman, G J and G L Steele Jr (1975). *Scheme: An interpreter for extended lambda calculus*. Tech. rep. AIM-349. MIT. URL: <https://dspace.mit.edu/handle/1721.1/5794>.
- (1998). “Scheme: A Interpreter for Extended Lambda Calculus”. In: *Higher Order Symbol. Comput.* 11.4, pp. 405–439. ISSN: 1388-3690. DOI: [10.1023/A:1010035624696](https://doi.org/10.1023/A:1010035624696).
- Sutherland, I E (1963). *Sketchpad: A Man-Machine Graphical Communication System*. Tech. rep. The address of the publisher: Lincoln Lalboatory, Massachusetts Institute of Technology. URL: <https://apps.dtic.mil/dtic/tr/fulltext/u2/404549.pdf>.
- Takači, Dj, G Stankov, and I Milanovic (2015). “Efficiency of learning environment using GeoGebra when calculus contents are learned in collaborative groups”. In: *Computers & Education* 82, pp. 421–431. ISSN: 0360-1315. DOI: <https://doi.org/10.1016/j.compedu.2014.12.002>. URL: <http://www.sciencedirect.com/science/article/pii/S0360131514002796>.
- Tall, D O (1986). “Using the computer as an environment for building and testing mathematical concepts: A tribute to Richard Skemp”. In: *Mathematics Education*

- Research Center, Warwick Univ., pp. 21–36. URL: <http://homepages.warwick.ac.uk/staff/David.Tall/pdfs/dot1986h-computer-skemp.pdf>.
- Tankelevičienė, L (2004). “Mobile Technologies for Mobile Students”. In: *Informacinės Technologijos Ir Valdymas* 33.4, pp. 35–40.
- Tansey, W and E Tilevich (2008). “Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications”. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. New York, NY, USA: ACM, pp. 295–312. URL: <http://people.cs.vt.edu/tilevich/papers/rosemary-oopsla.pdf>.
- Targamadžė, A and R Petrauskienė (2010). “Impact of Information Technologies on Modern Learning”. In: *Information Technology and Control* 39.3, pp. 169–75. URL: <http://itc.ktu.lt/index.php/ITC/article/view/12375/6847>.
- Tarver, H et al. (2015). “An Exploratory Analysis of Subject Metadata in the Digital Public Library of America”. In: *Proceedings of International Conference on Dublin Core and Metadata Applications*, pp. 30–40. ISBN: 1939-1366. URL: <http://dcpapers.dublincore.org/pubs/article/view/3761>.
- Tatar, E (2013). “The Effect of Dynamic Software on Prospective Mathematics Teachers’ Perceptions Regarding Information and Communication Technology”. In: *Australian Journal of Teacher Education* 38.12, pp. 1–16. ISSN: 0360-1315. DOI: 10.14221/ajte.2013v38n12.6. URL: <https://ro.ecu.edu.au/cgi/viewcontent.cgi?article=2140&context=ajte>.
- Tatarczak, A and M Medrek (2017). “Educational experience in teaching mathematics online: a case study on the implementation of GeoGebra in an interactive learning environment”. In: *INTED 2017, 11th annual International Technology, Education and Development*, pp. 5416–5424. DOI: 10.21125/inted.2017.1262.
- Štátny pedagogický ústav, Štátny vzdelávací program, Vzdelávacia oblasť: Matematika a práca s informáciami, Slovenskej republiky, Slovakia (2019). [Online; accessed 02-June-2019]. URL: http://www.statpedu.sk/sites/default/files/dokumenty/statny-vzdelavaci-program/matematika_isced1.pdf.
- The Coq Proof Assistant (2019). [Online; accessed 02-June-2019]. URL: <https://coq.inria.fr/>.
- The Geometer’s Sketchpad (2019). [Online; accessed 02-June-2019]. URL: <http://www.keycurriculum.com/sketchpad.1.html>.
- The Interactive Geometry Software Cinderella (2019). [Online; accessed 02-June-2019]. URL: <https://cinderella.de/tiki-index.php>.
- The Metadata Community — Supporting Innovation in Metadata Design, Implementation & Best Practices (2019). [Online; accessed 02-June-2019]. URL: <http://dublincore.org/metadata-basics/>.
- Tomaschko, M and M Hohenwarter (2017). “Integrating Mobile and Sensory Technologies in Mathematics Education”. In: *Proceedings of the 15th International Conference on Advances in Mobile Computing & Multimedia*. MoMM2017. Salzburg, Austria: ACM, pp. 39–48. ISBN: 978-1-4503-5300-7. DOI: 10.1145/3151848.3151866.
- Tomiczková, S and M Lávička (2013). “Computer-Aided Descriptive Geometry Teaching”. In: *Computers in the Schools: Interdisciplinary Journal of Practice, Theory, and Applied Research* 30.1-2, pp. 48–60. ISSN: 0738-0569. DOI: 10.1080/07380569.2013.764480.

- Turner, D A (1982). "Recursion Equations as a Programming Language". In: *Darlington, Henderson, Turner (eds.) Functional Programming and its Applications*, Cambridge University Press, pp. 1–28. DOI: https://doi.org/10.1007/978-3-319-30936-1_24.
- (1985). "Miranda: A Non-strict Functional Language with Polymorphic Types". In: *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., pp. 1–16. ISBN: 3-387-15975-4. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.6357&rep=rep1&type=pdf>.
- Underkoffler, M M (1969). "Computer assisted instruction in college general education mathematics". PhD thesis. Digital Repository @ Iowa State University, <http://lib.dr.iastate.edu/>: Iowa State University. DOI: <https://doi.org/10.31274/rtd-180813-1201>.
- Using Attributes in C#* (2019). [Online; accessed 02-June-2019]. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/attributes>.
- Van Den Berg, K G (1995). "Software Measurement and Functional Programming". PhD thesis. University of Twente Enschede.
- Vidaković, J and M Racković (2006). "Generating content and display of library catalogue cards using XML technology". In: *Software: Practice and Experience* 36.5, pp. 513–524. DOI: [10.1002/spe.707](https://doi.org/10.1002/spe.707).
- Wadler, P L et al (1988). *Introduction to Orwell 5.00*. Programming Research Group. Oxford U. URL: <https://homepages.inf.ed.ac.uk/wadler/papers/orwell/orwell1.pdf>.
- Wang, D (1993). "An elimination method for polynomial systems." English. In: *J. Symb. Comput.* 16.2, pp. 83–114. ISSN: 0747-7171.
- Wang, H (1960). "Toward Mechanical Mathematics". In: *IBM Journal of Research and Development* 4.1, pp. 2–22. ISSN: 0018-8646. DOI: [10.1147/rd.41.0002](https://doi.org/10.1147/rd.41.0002). URL: <http://dx.doi.org/10.1147/rd.41.0002>.
- Warth, A (2007). "LazyJ: Seamless Lazy Evaluation in Java". In: *In Workshop on Foundations of Object-Oriented Languages*. URL: <http://www.cs.cmu.edu/~aldrich/FOOL/FOOLWOOD07/program/warth.pdf>.
- Wernick, W (1982). "Triangle constructions with three located points". In: *Mathematics Magazine* 55.13.4, pp. 227–230. URL: <http://www.polarprof.org/geometriagon/pg/Wernick.pdf>.
- Wester, M (1999). *Computer Algebra Systems: A Practical Guide*. John Wiley & Sons, Ltd, Chicester., pp. 1–436.
- Whitehead, A N and B Russell (1910). *Principia Mathematica. Vol. I*. English. URL: <https://archive.org/details/principiamathema01anwh/page/n8>.
- Windows Presentation Foundation* (2019). [Online; accessed 02-June-2019]. URL: <https://msdn.microsoft.com/en-us/library/ms754130.aspx>.
- Winroth, H (1999). "Dynamic projective geometry". PhD thesis. Stockholms Universitet. URL: <https://www.nada.kth.se/utbildning/forsk.utb/avhandlingar/dokt/winroth990324.pdf>.
- Wolfram Mathematica Comparative Analyses - Computer Algebra Systems* (2019). [Online; accessed 02-June-2019]. URL: <https://www.wolfram.com/products/mathematica/analysis/content/ComputerAlgebraSystems.html>.

- Wolfram Math World: Triangle* (2019). [Online; accessed 02-June-2019]. URL: <http://mathworld.wolfram.com/Triangle.html>.
- Writing Custom Attributes* (2019). [Online; accessed 02-June-2019]. URL: <https://docs.microsoft.com/en-us/dotnet/standard/attributes/writing-custom-attributes>.
- Wu, W T (1978). "On the decision problem and the mechanization of theorem-proving in elementary geometry". In: *Scientia Sinica* 21.2, pp. 159–172. DOI: <https://doi.org/10.1360/ya1978-21-2-159>. URL: <http://engine.scichina.com/publisher/ScienceChinaPress/journal/ScientiaSinica/21/2/10.1360/ya1978-21-2-159>.
- (1999). "Automatic Geometry Theorem-Proving and Automatic Geometry Problem-Solving". In: *Proceedings of the Second International Workshop on Automated Deduction in Geometry*. ADG '98. London, UK, UK: Springer-Verlag, pp. 1–13. ISBN: 3-540-66672-9. URL: <http://dl.acm.org/citation.cfm?id=647692.731041>.
- Yiu, P (2008). "Conic Construction of a Triangle from the Feet of Its Angle Bisectors". In: *Journal for Geometry and Graphics* 12.2, pp. 171–182. URL: <http://www.heldermann-verlag.de/jgg/jgg12/j12h2yiu.pdf>.
- Zbiek, RM and A Conner (2006). "Beyond Motivation: Exploring Mathematical Modeling as a Context for Deepening Students' Understandings of Curricular Mathematics". In: *Educ Stud Math* 63.1, pp. 89–112. DOI: [10.1007/s10649-005-9002-4](https://doi.org/10.1007/s10649-005-9002-4).
- Zhuravlev, V and P Samovol (2016). "A new list of triangle construction problems or supplementing Wernick". In: *Mathematics Competitions* 29.1, pp. 31–64. URL: <http://www.wfnmc.org/Journal%202016%201.pdf>.

Prošireni izvod (Extended Abstract in Serbian)

Predgovor

Još od izuma Žakardovog razboja¹ ljudi žele razviti automate i programe koji će im pomoći u svakodnevnom životu. U današnje vreme su računari u svakodnevnoj upotrebi u svim segmentima naših života. Obrazovanje je u našem životu jedan od najvažnijih segmenata. Stoga, možemo reći da je jedna od glavnih svrha računara njihova upotreba u obrazovanju. No, nisu samo računari potrebni i dovoljni za predavanje, takođe je potreban odgovarajući softver. To je razlog što se softver za dinamičku geometriju razvija godinama sve intezivnije.

Softver za dinamičku geometriju (DGS) dozvoljava korisnicima da manipulišu geometrijskim objektima, vukući ih, i na taj način pretvarajući u druge oblike, ili menjajući im poziciju. Dakle, korisnik na taj način doživljava šta se dešava sa konstrukcijom. Međutim, DGS se ne koristi samo za podučavanje geometrije, već i drugih predmeta, kao što su fizika, geografija, itd.

U aktuelnim verzijama softvera za dinamičku geometriju smo vremenom uočili dva nedostatka: softver za dinamičku geometriju nije dobro prilagođen univerzalnim primenama, uglavnom sadrže geometrijske objekte, i ti objekti sadrže samo osnovne osobine, da bi bili lakši i efikasniji za izračunavanje. Imajući to u vidu, naš cilj je razvoj softvera za dinamičku geometriju koji bi bio baza za eksperimente koji bi rezultirali poboljšanjem razvoja materijala za podučavanje i igara.

Ova disertacija predstavlja nov pristup anotacijama izvornog koda, na fleksibilan i jednostavan način, koji sadrži kompleksnu strukturalnu informaciju metapodataka, koja dozvoljava upotrebu vrednosnih tipova, koji nisu CLR tipovi, a dostupan je koristeći refleksiju. Kreirali smo generički proširiv softver za dinamičku geometriju SLGeometry, napisan u C# jeziku na .NET Framework-a, sa ciljem da se poštuju zahtevi dizajna takvog softvera i predloži održiva implementacija proširivog okvira zasnovanog na metapodacima.

Glavne komponente razvijenog okvira su:

- Parser izraza;
- Evaluator izraza (Engine);
- Grafička površina (GeoCanvas).

Opisan pristup donosi u SLGeometry okvir sledeće značajne osobine:

- Proširivost novim tipovima, funkcijama i vizuelnim objektima;
- Ujedinjenje objekata sa njihovim osobinama pristupajući im koristeći dot notaciju;
- Obogaćena strukturalna specifikacija metapodataka softvera za dinamičku geometriju sa optimizacijom i mehanizmom za lenjo izračunavanje.

Sadržaj disertacije je organizovan u tri dela: uvodna razmatranja, implementacija, i validacija. Delovi su organizovani u poglavlja na sledeći način.

¹<https://www.computerhope.com/jargon/j/jacquard-loom.htm>

Deo I - Uvodna razmatranja daje pregled osnovnih pojmova kao i trenutnog stanja postojećih rešenja. Poglavlje 1 daje kratak uvod predstavljajući: formulaciju problema koje želimo da rešimo, glavne ciljeve u razvoju okvira, kao i listu eksplicitnih doprinosa ove disertacije. U Poglavlju 2 daje se kratak pregled trenutnog stanja dobro poznatih softvera za dinamičku geometriju, lenjog izračunavanja, metapodataka i objektno-orijentisanih proširenja, te upotrebe komponenti.

Deo II - Implementacija predstavlja nekoliko motivacijskih primera koji naglašavaju probleme u implemetaciji atributa i korišćenje softvera za dinamičku geometriju sa čisto funkcionalnim jezicima, te daje detalje implemetacije datog rešenja zajedno sa pregledom arhitekture sistema. Poglavlje 3 daje: pregled atributa u .NET-u i prikazuje njihove nedostatke; diskutuje o prednostima objekata sa osobinama i dot (‘.’) notacije; uvodi semantičke ekstenzije i kompozicije jezika zasnovane na funkcionalnom domen-specifičnom jeziku (FLG). U poglavlju 4 su opisani SLGeometry okvir i njegova arhitektura, predstavljen je opis FLG jezika i infrastruktura klasa izraza. Detaljnija opisana implemetacija daje glavne doprinose koje se tiču objektno-orijentisanog proširenja softvera za dinamičku geometriju podržanog metapodacima, i mehanizma za lenjo izračunavanje izračunatih osobina. Takođe su date konverzije tipova, operacije sa keširanjem rezultata, parcijalno kompajliranje stabala izraza i XML reprezentacija crteža u Poglavlju 5.

Deo III - Validacija se bavi validacijom datih koncepata i mehanizama koje smo uveli u prethodnom delu. Obrazac za kreiranje matematičkih igara u SLGeometry ukombinovan sa svojstvima naših komponenti je predstavljen u Poglavlju 6. Osim toga, sproveli smo nekoliko eksperimenata sa učenicima i studentima gde smo testirali naše pristupe u učionici. Za eksperimentalnu verifikaciju predložene infrastrukture metapodataka sa lenjim izračunavanjem, korišteni su rezultati ispitivanja upoređivanja brzina izvršavanja i broja objekata i zauzeća memorije prezentovani u Poglavlju 7. Eksperimenti su sprovedeni u tri različite strategije izračunavanja stabla izraza: eager, funkcionalno i lenjo izračunavanje. Poglavlje 8, na kraju, zaključuje disertaciju i diskutuje o trenutnim rezultatima i budućim pravcima za daljnja istraživanja.

Relevantni radovi koji su objavljeni tokom pisanja disertacije su sledeći: Radaković and Herceg (2010), Radaković, Herceg, and Löberbauer (2010), Herceg and Radaković (2011), Herceg, Herceg-Mandić, and Radaković (2012), Herceg, Radaković, and Herceg (2012), Radaković and Herceg (2013), Steingartner et al. (2016), Radaković and Herceg (2017), Herceg et al. (2019) and Radaković and Herceg (2018).

Uvod

Softver za dinamičku geometriju (DGS) naširoko prihvataju nastavnici širom sveta, na svim razinama obrazovanja, kao alat za kreiranje, demonstraciju i širenje interaktivnih nastavnih materijala u obliku dinamičkih crteža. To značajno utiče na način na koji se geometrija predaje u školama, primenjujući moderno učenje koristeći DGS, koji su intuitivni i laki za korišćenje (Abramovich, 2013; Tankelevičienė, 2004; Targamadzė and Petrauskienė, 2010). Nekoliko dobrih DGS-a je izdržalo test vremena: *GeoGebra* (2019), *The Interactive Geometry Software Cinderella* (2019), *Cabri* (2019), and *The Geometer’s Sketchpad* (2019).

Interaktivnost u DGS-u proizlazi iz jednostavnog principa: dok se jedan deo dinamičkog crteža menja, automatski se preračunavaju i svi delovi koji su zavisni o njemu. Dinamički crteži su definisani izrazima koji su napisani domen-specifičnim funkcionalnim jezicima, te je stoga nastala potreba za njihovom proširivosti i primenom u drugim oblastima, osim prvobitno predviđene geometrije.

Upotreba atributa je preferirani mehanizam, koji povezuje deklarativne informacije sa C# kodom, no oni imaju određene restrikcije koje limitiraju njihovu upotrebu kod reprezentacije kompleksno struktuiranih metapodataka. Naime, samo podskup ugrađenih CLR tipova podataka su dozvoljeni da se koriste u osobinama atributa (*Using Attributes in C#*, 2019; Alvi, 2002). Ovo istraživanje prvenstveno želi prevladati ovaj problem razvojem infrastrukture podataka koja je nezavisna od atributa, te na taj način prevazići opisane nedostatke, bazirajući se na C# jezik i .NET platformu.

Novi koncepti u ovom istraživanju omogućuju proširenje jednostavnih i kompleksnih tipova podataka, unarnih i binarnih operatora, konverzije tipova, funkcija i vizuelnih objekata, i na taj način omogućujući programerima, da implementirajući ih kao C# klase koje su označene metapodacima, lako dodaju nove funkcionalnosti u SLGeometry. Jedna od prednosti ovog koncepta je uvođenje alternative za .NET attribute, daljnji razvoj specifikacije metapodataka pogodnih za opis složenih hijerarhijskih struktura i njihovih međusobnih zavisnosti, implementirajući tipski i operacijski nezavisnu optimizaciju izračunavanja, a metapodaci su razvijeni kao plug-in elementi.

Metapodaci i tehnike kao što metaprogramiranje, atribut-orijentisano programiranje i komponentno-objektno programiranje (razvoj zasnovan na komponentama) su veoma zastupljeni zadnjih nekoliko decenija (Chlipala, 2016; Štuikys, Damaševičius, and Ziberkas, 2012; Lilis and Savidis, 2015; Hazzard and Bock, 2013). Atribut-orijentisano programiranje je tehnika obeležavanja programa koja omogućava programerima da deklarativno obogate program koristeći metapodatke. Dakle, programeri mogu da označe elemente programa kao što su klase, interfejsi, metode, polja, itd., sa atributima (anotacijama), ukazujući da održavaju domen-specifičnu semantiku (Noguera and Pawlak, 2007).

Dinamički crtež sadrži vizuelne objekte kao što su geometrijski likovi i slični objekti, a po konvenciji se može opisati skupom imenovanih izraza, napisanih u domen-specifičnom funkcionalnom jeziku, koji se čuvaju u promenljivima. Zavisnosti među promenljivima, ustanovljena referencama promenljivih, formira uređen acikličan graf koji se prolazi tokom ponovnog izračunavanja da bi se održala konzistentnost crteža. Svi vizuelni objekti su generisani funkcijama koje proizvode konstante određenog tipa, koje zatim prouzrokuju pojavu vizuelnog objekta na ekranu. Funkcije možemo podeliti na: vizulene funkcije koje vraćaju geometrijske oblike, funkcije osobina koji vraćaju ili izračunavaju osobine geometrijskih oblika ili druge funkcije kao što su matematičke, logičke, manipulacije stringovima, itd.

Vremenom smo uočili da postojeći DGS imaju sledeće nedostatke: (1) DGS nisu pogodni za univerzalnu primenu, jer uglavnom sadrže geometrijske objekte i funkcije koje njima rukovode, i (2) geometrijski objekti sadrže samo osnovne minimalno potrebne količine podataka, dok se dodatne osobine izračunavaju primenjujući posebne funkcije na te objekte. Takođe, kreiranje kompleksnih crteža, koji su sastavljeni od mnoštva osnovnih geometrijskih oblika, je vrlo nezgodno. Isto tako je nezgodno i

pristupanje njihovom osobinama, koje zbog složenosti konstrukcije vodi kompoziciji složenih izraza, zbog semantike funkcionalnih jezika. Stoga je naš cilj bio da se ujedine objekti sa svim svojim osobinama, kojima bi se pristupalo uvodeći objektno-orijentisan pristup, dot notacije za pristup osobinama objekata.

Osnovni cilj istraživanja je specifikacija metapodataka za softvere za dinamičku geometriju koja prevazilazi ograničenja atributa u vidu primenljivosti isključivo na CLR tipove podataka, uz zahtev da sadrže podatke o tipu i vrednosti svojih osobina. Uz to je kreirana strategija optimizacije izračunavanja, i algoritam za aktivaciju i deaktivaciju osobina koje treba da se izračunavaju u nekom objektu. Predmet istraživanja je objektno-orijentisano proširenje softvera za dinamičku geometriju (DGS) podržano metapodacima i zasnovano na evaluatoru izraza koji podržava lenjo izračunavanje i parcijalno kompajliranje.

Istraživanje je fokusirano na kreiranju platforme sa:

1. Ujedinjenim objektima sa svim osobinama;
2. Realizacija pristupa osobinama bez upotrebe specijalnih funkcija;
3. Definisane mehanizma za proširenje novim tipovima, funkcijama i vizuelnim objektima;
4. Usvajanje strategija za verifikaciju, izračunavanje i optimizaciju;
5. Specifikacija metapodataka za DGS.

Dobijeni naučni doprinosi disertacije su:

- Detaljno je dat opis alternative za .NET attribute, koja je pogodna za predstavljanje metapodataka kompleksne hijerarhijske strukture. Specifikacija metapodataka je data za unarne i binarne operacije i konverzije tipova, objektno tipove podataka, uvodeći zavisne osobine i duboke zavisnosti;
- Predstavljena je proširiva platforma za dinamičku geometriju SLGeometry, čije su glavne komponente: parser, evaluator izraza i grafička podloga, i njena arhitektura
- Rasprostranjena upotreba anotacija metapodataka omogućuje odvajanje uopštenih algoritama od konkretne semantike;
- Implementiran je i diskutovan daljnji razvoj specifikacije metapodataka pogodnih za izražavanje kompleksnih hijerarhijskih struktura i zavisnosti;
- Implementirana je tipski i operacijski neutralna šema optimizacije izračunavanja, a specifikacija metapodataka podržava razvoj i implementaciju novih funkcionalnosti u vidu plug-in komponenti;
- Predstavljen je generički funkcionalan jezik čija je semantika odvojena od jezičke implementacije i realizovana u formi plug-in tipova, operacija i funkcija;
- Uvedeni su objektni tipovi podataka sa osobinama koje se izračunavaju;
- Za korisnika, objektna (dot) notacija smanjuje kompleksnost napisanih izraza;
- Programerima je olakšan koncept logički povezanog koda unutar jedne jedinice;
- U praktičnoj primeni nisu uočeni zaostaci u izvršavanju.

Pregled postojećih rezultata

S kraja šezdesetih godina prošlog veka, nastavnici matematike su počeli da podržavaju upotrebu matematičkih aplikacija u predavanju i učenju matematike. Možemo reći da je sve započelo sa Sutherlandovim uvođenjem Sketchpada (Sutherland, 1963), gde je predstavljen korisnički grafički interfejs. Zajedno sa razvojem automatskih dokazivača (Gelernter, Hansen, and Loveland, 1960; Wang, 1960; Wu, 1978; Chou, Gao, and Zhang, 1994), razvila se potreba za razvojem adekvatnog grafičkog interfejsa koji dozvoljava interaktivnu manipulaciju mišem, i manipulaciju matematičkih izraza u simboličkoj formi.

S vremenom i većom dostupnosti računara, upotreba računara i softvera za edukaciju je ušla u školski program, pogotovo u matematičke i prirodne nauke (Niss, 2012; Underkoffler, 1969; Kaufmann and Schmalstieg, 2002; Drijvers et al., 2016; Blum and Borromeo Ferri, 2009). Upotreba računarskog softvera u edukaciji je polako integrisana u nastavi od osnovne škole pa sve do fakulteta. No problem je u njihovoj dostupnosti, pogotovo u zemljama u razvoju (Mainali and Key, 2012; Bhagat and Chang, 2015; Khalil et al., 2018; Han et al., 2013). Slična situacija je u Srbiji, no sve veći broj nastavnika se trudi da uvede informacijske i komunikacijske tehnologije u učionice (Radović, Marić, and Passey, 2019; Jezdimirović, 2014; Ljajko and Ibro, 2013; Diković, 2009).

Komercijalni proizvodi, kao što su Cabri (Cabri, 2019; Laborde and Capponi, 1994), Cinderella (*The Interactive Geometry Software Cinderella*, 2019) i Geometer's Sketchpad (*The Geometer's Sketchpad*, 2019; Jackiw and Finzer, 1993), pokrivaju školsku matematiku i fiziku zajedno sa algebrom, analizom, geometrijom, mehanikom i optikom, itd. Veliku podršku im daju forumi i tutorijali gde se nalaze mnogobrojni nastavni materijali sa postavkama i rešenjima problemima.

Zadnjih godina, pored već poznatih komercijalnih programa, javljaju se besplatni i otvorenog koda projekti za sisteme za dinamičku geometriju: SageMath (Software for Algebra and Geometry Experimentation) (Stein, 2019b; Stein, 2012), GCLC (Janičić, 2019; Janičić, 2010; Janicic, Narboux, and Quaresma, 2012), ArgoTriCS (Automated Reasoning GrOup Triangle Construction Solver) (Marinković, 2016; Schreck et al., 2016), GeoGebra (*GeoGebra*, 2019; *GeoGebra Materials*, 2019), itd.

Svim ovim softerima za dinamičku geometriju je zajedničko da imaju funkcionalan pristup za kreiranje dinamičkih crteža.

Krajem sedamdesetih i početkom osamdesetih godina prošlog veka, pojavila se ideja za lenjim (lazy) nestriktnim jezicima kao odgovor na Scheme (Sussman and Steele Jr, 1975; Sussman and Steele Jr, 1998), Milnerov meta-language ML (Milner, 1978) i nekim drugim striktnim (call-by-value) jezicima. Lenjo izračunavanje, takođe nazvano call-by-need, se koristi kao strategija izračunavanja u funkcionalnim jezicima. Lenjo izračunavanje je intuitivno; izrazi i njihovi podizrazi se izračunavaju u run-time-u samo kad je to potrebno, i to samo jednom.

Prema Hudak et al. (2007) lenjo izračunavanje je bilo nezavisno uvedeno tri puta:

- Cons konstruktor funkcija predstavljena u tehničkom izveštaju Friedman and Wise (1976);
- Henderson and Morris (1976) su prezentovali algoritam koji odlaže izračunavanje parametara i listi u LISP-u;

- David Turner je razvio Kent Recursive Calculator kao mini verziju SASL (St Andrews Static Language)

Nakon toga je više istraživača isto tako nezavisno dizajniralo njihove čiste lenje jezike: Miranda Davida Turnera (Turner, 1985), G-mašina Johnsson (1984) i Augustsson (1984), Orwell razvijen kao besplatna alternativa Mirandi (Wadler, 1988), i Id nestriktni jezik za protok podataka razvijen na MIT od Arvind, Nikhil, and Pingali (1989).

Mehanizmi koji omogućuju prilaganje proizvoljnih metapodataka nekom delu programa se sve više koriste u programskim jezicima, npr. koristeći attribute u .NET-u ili anotacije u Javi. Njihova primena je veoma raširena (Cazzola and Vacchi, 2014; Löberbauer et al., 2010; Jahn et al., 2013; Schult and Polze, 2002; Berzal et al., 2005; Greaves and Singh, 2008). Neki istraživači su primetili izvesne nedostatke u postojećem mehanizmu specifikacije metapodataka, kao što su validacija ispravnosti metapodataka ili njihova ponovna upotreba (Song and Tilevich, 2015; Noguera and Duchien, 2008; Eichberg, Schäfer, and Mezini, 2005).

U današnje vreme je softver kompleksan i razvoj po delovima je omogućen uvođenjem softverskih komponenti (Heineman and Councill, 2001; Bourque and Fairley, 2014; Chen et al., 2007; Seco, Silva, and Piriquito, 2008). Mnogi moderni programski jezici, razvojni alati i metodologije razvoja podržavaju razvoj softvera po delovima, gde razni timovi saraduju i imaju različite razvojne cikluse za različite delove konačnog softvera.

Mi smo želeli da uvedemo ovaj princip u DGS, time što smo omogućili razvoj komponenti, nezavistan od razvoja samog DGS. Te komponente se uključuju u DGS u runtime-u i imaju isti tretman kao first-class citizens u DGS-u. Prirodni nastavak ovog principa je da se omoguće i vizuelne komponente, koje pored programskih aspekata imaju i grafičku prezentaciju.

Vizualizacija je veoma bitna u modernoj nastavi. Nove generacije žele da vide i osele same što uče. Softveri za dinamičku geometriju omogućuju te koncepte i ohrabruju učeničku autonomiju. Primetili smo da trenutni softveri za dinamičku geometriju koriste funkcionalan pristup, tj. osobine objekata su definisane zasebnim funkcijama, dok mi uvodimo OO principe i dot notaciju u DGS.

Upotreba atributa, kao što su ih koristili Löberbauer et al., 2010, Jahn et al., 2013, Schult and Polze, 2002, Berzal et al., 2005, Greaves and Singh, 2008, Casero, Cesarini, and Monga, 2003, Benton, Cardelli, and Fournet, 2004, je u početku bila zastupljena u SLGeometry, no ispostavila se nedovoljnom (predstavljeno u Poglavlju 3), i kasnije preraslo u naš metamodel (Poglavlje 5) gde smo sledili ideju metapodatka strukturisanih komponenti (*Windows Presentation Foundation*, 2019).

Takođe smo se vodili idejama prezentovanim u Sulír, Nosál', and Porubän (2016), Nosál', Sulír, and Juhár (2016), omogućujući korespondenciju između matematičkih definicija osobina geometrijskih objekata i koda. Isto tako su primenjena pravila vezivanja, tj. svaki označen deo sa metapodacima odgovara nekom programskom elementu, dok se metapodaci mogu pretražiti refleksijom i koristiti se kao plug-in-ovi (Poglavlje 5). Istodobno se očekuje da je veza između metapodataka i njihovih ciljnih elemenata smislena (Sekcije 5.4.1 i 5.4.2). Za razliku od Tansey and Tilevich (2008) naš okvir omogućuje automatsko generisanje plug-in-ova zasnovanih na metapodacima ručno anotiranog koda.

Naša proširiva platforma se vodi principima komponent-objektnog programiranja. Predložili smo apstraktni model evaluatora izraza i skup metapodataka koji obezbeđuju realne funkcionalnosti (Rouvoy and Merle, 2006; Guerra and Fernandes, 2013; Siqueira, Silveira, and Guerra, 2016).

Uvođenje objektno-orijentisanih svojstava u naš sistem (Sekcija 3.4.2) zastupa pravce proučavane od strane Mernik (2013), Erdweg, Giarrusso, and Rendel (2012), Chodarev et al. (2014), dok je deo sa parcijalnim kompajliranjem izraza pratio smer-nice Palmer and Smith (2011).

Lenjo izračunavanje i dalje veoma popularna strategija koju smo koristili zbog objekata sa velikim brojem osobina (npr. trougao Sekcija 7.2). Kad je osobina objekta obeležena za izračunavanje, mehanizam za aktivaciju osobina započinje izračunavanje neophodnih osobine, i nakon što su one sve izračunate, data osobina se može izračunati. Ovaj princip je opisan u Sekciji 5.4.

Motivacija

Kao što je već u uvodnim poglavljima naglašeno, nedostaci atributa u .NET-u i funkcionalni pristup softvera za dinamičku geometriju, doveli su do razvoja infrastrukture podataka u SLGeometry. Za attribute se može reći da su kao pridevi, pošto daju detaljnije informacije o nekom delu programa (*Using Attributes in C#, 2019; Writing Custom Attributes, 2019; C# - Attributes, 2019; Alvi, 2002; Agarwal, 2013*).

Upotreba atributa u .NET okviru je propisana sledećim implementacijama: predefinisanim atributima i prilagođenim (custom) atributima. Atributi prihvataju argumente kao u metodama, i parametri mogu biti pozicionalni, imenovani i neimenovani. Metapodaci se dobijaju koristeći mehanizam refleksije.

Za ilustraciju potrebe uvođenja nove infrastrukture metapodataka koja bi prevazišla nepremostive prepreke koje atributi u .NET-u imaju uzećemo krug. Krug je geometrijski objekat koji se može definisati na više načina. Jedan način je definisanjem centra kruga i njegovog poluprečnika. Sledeći podaci su potrebni za definisanje kruga funkcijom *FCircle* u SLGeometry: ime poziva funkcije 'Circle' i njen rezultat *CCircle*; argumenti: A – centar kruga (tačka), r – poluprečnik (broj); debljina kruga; potpisi: A, r – krug definisan tačkom A i poluprečnikom r , i r – centar kruga je u tački $(0, 0)$ sa poluprečnikom r .

Listing 3.2 predstavlja upotrebu prilagođenog atributa da bi se opisala funkcija za krug u SLGeometry. Dakle, metapodaci koji su potrebni da bi se opisala funkcija su sledeći:

- Ime funkcije i njen povratni tip;
- Tip i imena argumenata, za svaki argument jedna pojava;
- Potpisi funkcija, jedna pojava za svaki potpis.

Na sličan način su korišteni atributi za opisivanje svih funkcija koje su potrebne za bilo koji softver za dinamičku geometriju, ali problem se pojavio kad je trebalo opisati metapodatke koji su kompleksni i imaju podrazumevane vrednosti koji nisu standardni .NET tipovi. U našem slučaju je to podrazumevana vrednosti za centar kruga, tačka $(0, 0)$ predstavljena instancom klase *CPoint* (poslednji atribut u Listingu 3.2), kad je krug zadan samo poluprečnikom. Dok se piše modifikacija atributa

C# kompajler ne daje nikakvu grešku, već se ona javlja pri pokretanju programa: “Attribute constructor parameter 'p' has type 'Expr', which is not a valid attribute parameter type”.

Dakle, nije bilo moguće definisati podrazumevane vrednosti imenovanih argumenata koristeći atribut, te je bilo potrebno koristiti neko drugo rešenje (pogledati Sekciju 5.1). Takođe, još jedan argument protiv atributa je bila njihova nemogućnost da se predstavi njihova hijerarhijska struktura. Ako posmatramo već dati Listing 3.2, vidimo da je *FCircle* anotiran sa šest atributa koji imaju smisla samo ako se posmatraju u celini.

Stoga je rešenje za date nedostatak razvoj prilagođenih C# klasa koje predstavljaju metapodatke. Naše klase metapodataka imaju sledeće prednosti nad atributima:

1. Vrednosti ne-CLR tipova podatka su dozvoljeni u metapodacima;
2. Hijerarhijska struktura je korektno predstavljena;
3. U konstruktoru klasa metapodataka izvršava se provera korektnosti datih metapodataka;
4. IntelliSense i AutoComplete mehanizmi u Visual Studio IDE omogućuju programerima pomoć pri pisanju metapodataka.

Važno je naglasiti da se na ovaj način metapodaci nalaze u jednom *FnInfo* objektu, a imenovani argumentni su dodeljeni statičkim promenljivima koje se lakše mogu kasnije referencirati u C# kodu.

U većini DGS, objektu su lagani i nose samo obavezne osobine (*GeoGebra*, 2019; *Cabri*, 2019; *The Interactive Geometry Software Cinderella*, 2019), dok u SLGeometry objekti imaju sve svoje osobine. Posmatrajmo trougao, kao geometrijski koncept. On ima tri vrha koji su obavezne osobine, a pored toga ima i više izračunavajućih osobina koje se mogu izračunati na osnovu obaveznih osobina posebnim algoritmima. U slučaju kad su izračunavajuće osobine implementirane kao zasebne funkcije, broj funkcija bi vrlo brzo značajno porastao, jer svaki objekat donosi novi broj funkcija.

Ako se gleda sa korisničkog ugla, onda je veliki broj funkcija opterećujući za korisnika pri odabiru kad želi da piše izraze. A, sa programerskog ugla, mora se pratiti korektnost preopterećenja funkcija da bi se izbegla konfuzija sa funkcijama istog imena, ali koje deluju nad različitim tipom podataka. Da bismo izbegli ove probleme, odlučili smo da uključimo izračunavajuće osobine u objekte unutar SLGeometry, tj. osobine su deklarirane u metapodacima datog tipa unutar odgovarajuće C# klase. Za svaku osobinu je definisano ime, tip, zavisnost i delegat evaluatora. Infrastruktura osobina unutar Engine brine o pozivu delegata evaluatora kad je to potrebno. Na ovaj način je postignuto da se izračunavaju samo tražene osobine.

Uvodeći izračunavajuće osobine, potreba za dodatnim funkcijama je eliminisana. Sve osobine se mogu dobiti koristeći *Property* funkciju: *Property(object, "propertyname")*. No, ako se ona koristi za input, dobijaju se glomazni izrazi. Stoga smo uveli dot notaciju iz objektno-orijentisanih jezika, i na taj način uveli u FLG jezičko proširenje (Mernik, 2013; Erdweg, Giarrusso, and Rendel, 2012). Tokom parsiranja izraza, dot notacija je transformisana u funkcionalni ekvivalent, zahvaljujući odgovarajućim semantičkim akcijama (Mössenböck, 2010), koje se mogu primeniti rekursivno.

Zahvaljujući metapodacima, proizvoljan C# kod se može označiti i uvesti kao funkcija u SLGeometry, i na taj način značajno proširiti domen primene softvera za

dinamičku geometriju. *Table* funkcija demonstrira inkluziju proceduralnog koncepta (for petlja, pravljenje iterativnog niza) u FLG (Listing 3.5 i 3.6). No, istovremeno ona je funkcionalnu prirodu FLG-a, pošto vraća listu kao rezultat svakog izračunavanja, sa konkretnim skupom argumenata. Sad proceduralni kod i lokalne promenjive se smatraju internom implementacijom *Table* funkcije. Stoga možemo posmatrati semantiku unutar *Table* funkcije kao “semantic box” unutar FLG, slično kao jezički box (Diekmann and Tratt, 2014), pošto su oba uvedena kao plug-inovi.

U predloženom objektno-orijentisanom jeziku, reference osobina prati *Engine*, i referencirane osobine su aktivirane samo kad je to potrebno. Takođe se aktivirane osobine izračunavaju samo jednom, bez obzira na broj referenci koji ih pozivaju. Čist funkcionalan jezik je jednostavan za izračunavanje. S druge strane, objektna podrška i dot (‘.’) notacija, iako donose benefite korisnicima i programerima, donose kaznu kod kompleksnih algoritma za izračunavanje, što može dovesti do povećanja utrošene memorije i brzine izvršavanja.

Pregled arhitekture sistema

SLGeometry okvir je kompleksan softver za dinamičku geometriju, napisan u C# na .NET platformi. Izvorni kod sadrži više od 200 .cs fajlova sa više od 16500 linija koda. Dinamički crteži u SLGeometry su navedeni izrazima u FLG-u i dodeljeni imenovanim promenjivama. Izrazi mogu zavisiti od drugih izraza. Rezultat toga je interaktivnost, npr. ako korisnik pomeri tačku, sve što je povezano sa njom se isto tako pomera.

Struktura $\tau = \{T, C, O, F, V\}$ sadrži skup tipova (*T*), konverzije tipova (*C*), operacije (*O*), funkcije (*F*) i vizualne objekte (*V*) u FLG-u. Prateći motivaciju datu u Poglavlju 3 SLGeometry sadrži:

- Specifikaciju za tipove, funkcije i vizuelne elemente;
- Računarski algebarski sistem (CAS) Engine;
- Grafičku podlogu;
- Proširivu infrastrukturu;
- JIT podsistem kompajliranja;
- Parser izraza;
- Interaktivne komponente.

Glavne komponente su: parser, evaluator izraza (Engine) i grafička podloga (GeoCanvas) (Slika 4.1). Engine brine o skupu izraza, čuva imenovane promenjive, koje predstavljaju elemente dinamičkih crteža. GeoCanvas prikazuje geometrijske objekte i UI kontrole i odgovara na korisničku interakciju.

Kad se pokrene program, Engine inicijalizuje skladišta promenjivih, registracije imena, operacija i konverzija, te skenira asemblije i registruje ugrađene funkcije koristeći refleksiju. Skladište registracije imena sadrži fabriku funkcija i upravljače metapodataka funkcija, tj. sve funkcije, tipovi, UI kontrole, konstante i objektna konstante su tu registrovani. Registracija vizuelnih elemenata je inicijalizovana u GeoCanvas-u. Sve ove klase se mogu uvesti i registrovati preko spoljašnjih DLL-ova.

Korisnik sa sistemom može da komunicira na dva načina: pišući tekstualne izraze ili koristeći miš za pomeranje objekata po ekranu. Tekstualni unos procesira parser koji koristi fabriku izraza da konvertuje stablo izraza iz tipova *SLGParser* u *GExpression* nasledene objekte. Uveli smo *SLGParser* tipove podataka za čuvanje rezultata parsiranja, gde se čuvaju ime promenjive kojoj se dodeljuje i njena vrednost. Nadalje, tu se takođe čuvaju tipovi rezultata, tj. kad god promenjiva dobije neku vrednost dodeljuje joj se objekat, a ako je vrednost obrisana i osobina je obrisana, ili se samo parsira izraz bez imena promenjive.

Sva se izračunavanja vrše u *Engine-u*, i on sluša događaje u *GeoCanvas-u* i po potrebi preračunava neophodne osobine.

FLG se ponaša agnostički za tipove, operacije i funkcije, tj. sadrži generičke algoritme kojima se svi tipovi, operacije i funkcije uvoze iz C# klasa označenih metapodacima (Slike 4.1 i 4.2). Vizuelni objekti i konverzije tipova se takođe uvoze u *GeoCanvas* preko klasa označenih metapodacima. Koristeći refleksiju, *SLGeometry* proverava postojenje statičkih polja koja sadrže metapodatke svih uvezenih klasa, i registruje ih u τ (Slika 4.1).

FLG prepoznaje sledeće konstrukcije:

1. Konstante jednostavnih tipova, kao što su brojevi, logičke vrednosti i stringovi;
2. Funkcije sa i bez argumenata;
3. Reference promenjivih, tj. imena promenjivih koji se javljaju u izrazima, i oni se u parseru zamenjuju funkcijom *ValueOf("name");*
4. Operatore uobičajenih operacija: sabiranje, oduzimanje, množenje, deljenje i moduo. U parseru su zamenjeni odgovarajućim funkcijama;
5. Dodela promenjivih u $v = expression$ obliku, koji nije deo jezika, ali je interpretiran u *Engine-u* i ne može biti deo izraza;
6. Objektne konstante sa obaveznom i izračunavajućim osobinama, koje su rezultat evaluacije funkcija i ne mogu biti unošene ručno;
7. Pristup osobinama koristeći dot notaciju, koja se u parseru menja sa funkcijom *Property(object, "name");*
8. Dodela osobinama u obliku $v.property = expression$, nije deo jezika, i interpretira se u *Engine-u* kao dodela vizuelnih osobina, te ne može biti deo izraza.

Izrazi se grade iz atomskih vrednosti, objekata, listi i funkcija. *GExpression* je osnovna klasa za sve izraze. Jednostavni tipovi podataka i greške se nasleduju iz *Const* i *CLRConst<T>* klasa, dok se objektni tipovi podataka nasleduju iz *ConstObject* klase. Funkcije nasleduju *Fn* klasu, dok su vizuelne funkcije predstavljene *VisualFn* klasom. Sve bazne klase su predstavljene na Slici 4.7.

Funkcije mogu imati niti jedan ili više argumenata, i mogu biti preopterećene. Povratni tip se može znati unapred, kao što je to slučaj kod osnovnih funkcija (*Abs*, *And*, *Cos*, *Sin*, *Sqrt* itd.), geometrijskih funkcija (*Line*, *Segment*, *Triangle* itd.) i UI kontrola. Neke funkcije, npr. *If* može imati rezultat različitih tipova. Rezultati funkcija mogu biti konstante (*Const*) kao što su brojevi, logičke vrednosti i stringovi, ili objektne konstante (*ConstObject*), kao što su linije, krugovi i trouglovi.

Vizualni elementi u *SLGeometry* su implementirani sa tri klase: funkcijom, koja generiše objektu konstantu; objektom konstantom koji prezentuje stvarni vizalni objekat; i pomoćna klasa koja iscrtava vizualni objekat na *GeoCanvas*. Kada se

funkcija izračuna, GeoCanvas je obavešten da se rezultat promenio i odgovarajući vizualni objekat je nacrtan. Pošto su vizuelne osobine uobičajene .NET osobine, morali smo obezbediti konverziju tipova između .NET (CLR) tipova i FLG tipova podataka.

Detalji implementacije

Sve tipove podataka, operacije, konverzije tipova i funkcije je potrebno označiti sa metapodacima i registrovati sa Engine, dok se vizualni objekti trebaju registrovati sa GeoCanvas-om. Metapodaci su povezani preko statičkih polja, definisanih po konvenciji. FLG je jezik sa anotacijama u smislu koji je prezentovan od Nosál', Sulír, and Juhár (2016).

Specifikacija metapodataka je jasna i intuitivna. Na najvišoj razini, svaki tip izraza ima pridruženu klasu metapodataka (Slika 5.1). Funkcije su opisane sa *FnInfo* (Listing 5.5), dok su jednostavne konstante i konstante objekata opisane sa *TextConstInfo* i *ConstObjectInfo* (Listing 5.6), redom. Vizualne funkcije su opisane sa *VisualInfo* (Listing 5.8), a UI kontole sa *UIControlInfo* (Listing 5.7).

Metapodaci u funkciji sadrže potpise: *SignatureEmpty* za funkcije bez argumenata, *SignatureUnnamed* za funkcije sa jednim ili više argumenata, kao što su liste, i *SignatureNamed* za funkcije sa fiksnim brojem imenovanih argumenata kao što su tačke, linije i krugovi (Listing 5.2, Listing 5.13). Učitani argumenti se proveravaju da li odgovaraju potpisu na sledeći način: upoređuje se broj argumenata; argumenti koji su simbolički moraju biti tipa *TIdent*, i zamenjeni sa *ConstSymbol*; proverava se redom kompatibilnost tipova argumenata, osim onih koji su označeni; i na kraju se, onim argumentima koji nisu određeni u potpisu, dodeljuje podrazumevana vrednost.

Imenovani argumenti su prezentovani sa *ArgNamedInfo* klasom koja sadrži podatke o imenu argumenta, njegovom tipu i podrazumevanoj vrednosti (opciono) (Listing 5.1). Takođe se brine da li argumenti trebaju biti neizračunati, da li se argument direktno veže za imenovanu osobinu vizualne kontrole, ili se argument treba označiti simboličkim. Ukoliko je argument simbolički (Listing 3.6), npr. *Iterator* argument u funkciji *Table*, koristi se *ArgNamedSymbolInfo*.

No, posebnu pažnju treba posvetiti klasi *PropInfo*, koja opisuje osobine tipova konstantnih objekata. Pored imena osobine, vrednosti, njenog tipa i klase kojoj pripada osobina, ona sadrži dodatne informacije koje se koriste tokom izračunavanja i optimizacije (Listing 5.3), tj. sadrži sve tražene osobine koje trebaju da se izračunaju pre te osobine, delegate za izračunavanje, ime lokalne promenljive koja kešira vrednost osobine za brži pristup tokom izračunavanja, delegate za aktivaciju i deaktivaciju. Npr. *SideAProperty.Chain(CSegment.MidpointProperty)* vraća objekat koji sadrži traženu osobinu *SideAProperty* i podosobine *CSegment.MidpointProperty*, koji se prvo moraju izračunati. Takođe se vrši provera tipova koji sadrže podosobine, da li su istog tipa kao i osobina koja se prvobitno izračunava. Dodatno se vrši provera validnosti lanca traženih osobina, tj. da li data osobina traži podosobine koje poseduje.

Unarni i binarni operatori, kao i konverzije tipova podataka su predstavljeni na sličan način (Figure 5.2). Pošto se izrazi u softveru za dinamičku geometriju konstantno menjaju, i samim time izračunavaju, potrebno je bilo uvesti keširanje rezultata da bi se sprečilo nepotrebno gomilanje objekata na hipu. Dakle, kad god se izvrši neka operacija, keširani rezultat se prosleđuje delegatima na ažuriranje. Kod izvršavanja operacija, takođe je važno znati tip podataka operanada. Ukoliko je on poznat vrši se *rano* povezivanje, no ako to nije slučaj potrebno je izvesti *kasno* povezivanje, jer se tipovi operanada saznaju tek u momentu izvršavanja (Tabela 5.2, Listing 5.19).

Objekti sa velikim brojem osobina zahtevaju značajno više vremena da se ažuriraju. Da bi rešili taj problem, uvodimo algoritam za aktivaciju osobina, koji prati paradigmu poziva po potrebi kao što je dato u Sinot (2008), i stoga izračunava samo one osobine koje su referencirane drugim izrazima, i to samo jednom. Infrasktruktura aktivacije osobina u Engine-u prati reference osobina u izrazima, te ih aktivira kad su referencirani, i deaktivira kad se referenca ukloni. Neka osobina se isto tako može aktivirati, ako je ona tražena osobina za izračunavanje druge osobina unutar istog objekta.

Isto tako možemo posmatrati povezanost hijerarhijske strukture konkretnog objekta u jeziku i metapodataka. Duboka zavisnost osobina se može videti na primeru metapodataka za osobinu MedianA, koja zahteva osobinu Midpoint osobine SideA (Slika 5.3). Treba napomenuti da zavisnost osobina može formirati kompleksan graf zavisnosti, npr. CTriangle.Circumcircle prikazan na Slici 5.4.

Nekompajlirano stablo izraza se izračunava rekurzivno pozivajući Eval metod na svakom čvoru, i na taj način se transformišu delovi stabla izraza sa kompajliranim kodom. Koristili smo tri strategije optimizacije pri izvršavanju: stablo izraza sa simuliranim stekom, stablo izraza sa promenljivima, i delegati sa lambda izrazima. Ako je moguće kompajliranje svakog dela elementa stabla izraza, onda se svaki izraz može kompajlirati. Ako deo nekog izraza nije moguće kompajlirati, to ne znači da nije moguća optimizacija, jer je vrlo moguće da postoje delovi koji se mogu sami po sebi kompajlirati. Ukoliko se naiđe na izraz čije je celo stablo moguće kompajlirati, tada je moguće izvršavanje zameniti idealnijim i bržim kodom.

Kao i većina DGS, i mi smo odabrali XML format za prezentaciju crteža. Konvertovanje iz DGS jezika u XML se vrši posebnim konverterom koji serijalizuje promenjive iz Engine u XML, dok je konvertovanje u suprotnom smeru malo komplikovanije, jer se mora paziti da se svi izrazi učitaju u odgovarajućem redosledu, očuvavajući međusobne zavisnosti.

Predmetno-specifične komponente u DGS

Kompjuterska vizuelizacija obezbeđuje smislenost u podučavanju, što pomaže boljem razumevanju gradiva. Koristeći softver za dinamičku geometriju nastavnici lako mogu kreirati interaktivne nastavne materijale iz matematike. No, stvarna korist DGS-a je njihova primena na druge predmete kao što su geografija, fizika, itd. (Blum and Niss, 1991; Herceg and Herceg-Mandić, 2013). Naša želja je da demonstriramo kako se DGS prošireni komponentama mogu primeniti uspešno u nastavi geografije, na jednak način kao i u nastavi matematike.

U SLGeometry se mogu uvesti i koristiti softverske komponente iz DLL fajlova. Ove komponente su ili interaktivne vizualne kontrole (UI kontrole) ili sekvencijalne kontrole ponašanja. UI kontrole mogu predstavljati objekte kao što su dugmad, sijalice, satovi, geografske mape, itd. Kontrole ponašanja mogu sadržati logičku kontrolu, koja se koristi za kontrolu ponašanja interaktivnih crteža.

Koristeći komponente: složeni objekti se mogu predstaviti jednom komponentom, samo jedna promenjiva je dovoljna da sadrži celu komponentu, jedan na jedan je odnos između komponenti i onog što ona predstavlja, za dupliciranje kompleksnih objekata potrebna je samo još jedna promenjiva. No, nedostak je što se ove komponente moraju razviti izvan DGS od strane iskusnih programera.

Naš pristup smo testirali koristeći SLGeometry platformu u dva zasebna eksperimenta. Prvi eksperiment je imao dve faze: studenti matematike su pripremili dinamičke crteže koji su sadržali geografske mape, a srednjoškolci su koristeći te materijale rešavali zadatke.

Zadaci koje su dobili su: merenje dužine reke, praćenje putovanja Magellana i Cooka, te kreiranje mape sopstvenog odmora koji je rađen uz pomoć komponenti i bez njih. Studenti su svoje zadatke izvršili prilično uspešno, dok su srednjoškolci bili malo manje uspešni (Tabela 6.1). I srednjoškolcima i studentima se rad sa komponentama svideo, iako nisu baš za to da ih sami programiraju.

Proširujući naš prethodni rad, razvili smo kontrole ponašanja u SLGeometry koje su sekvencijalnog tipa, tj. imaju ulaz, memoriju i izlaz (Tabela 6.2).

Primer sa Aritmetičkom sumom demonstrira korišćenje okidača i njihovo propagiranje od jedne komponente ka drugoj (Slika 6.7). Koristimo komponente sekvencera (Sequencer) za izbacivanje vrednosti od 1 do 9, i čuvara rezultata (Scorekeeper). Klikanjem na dugme se vrednost sekvencera ciklično menja od 1 do 9, dok se rezultat čuva u Scorekeeper-u. Naravno, potrebno je povezati komponente kao što je to urađeno u Tabeli 6.3.

Drugi eksperiment demonstrira implementaciju pogađanja središnje tačke između dve prouzvoljne tačke (Slika 6.8). Ovde je logika igre implementirana koristeći kontrole, dok se ista postavka u GeoGebri radi pomoću skript programa. Generatori slučajnih brojeva (Randomizer) dodeljuju proizvoljno koordinate tačke B. Sekvencer prati koliko se igara igralo. Dugme za pritiskanje (PushButton) signalizira da je korisnik stavio marker za željeno mesto i želi proveriti njegovo rešenje (Tabela 6.4). Rezultati testova su dati u Tabelama 6.5 i 6.6. Možemo zaključiti da su sudionici prepoznali jednostavnost koju kontrole ponašanja nude, i većina ih je lako savladala, za razliku od pisanja skripti.

Kroz eksperimente sprovedene u nastavi sa učenicima, potvrđujemo prednosti našeg pristupa zasnovanog na kreiranju komponenti u DGS.

Testiranje lenjog izračunavanja

U prethodnim poglavljima smo predstavili SLGeometry arhitekturu i njene komponente, zajedno sa uvedenom kompleksnom hijerarhijskom strukturom metapodataka i predloženim mehanizmom za aktivaciju osobina koji implementira lenjo izračunavanje. Da bismo bili sigurni da data struktura isto tako koristi brzini izvršavanja i memorijskom utrošku, sprovedi smo eksperimentalnu verifikaciju.

Stoga smo upoređivali tri različite strategije izračunavanja stabla izraza: marljivo (eager), funkcionalno i lenjo (lazy). Za svaki od tih pristupa smo razvili poseban projekat sa istim jezgrom, ali različitim evaluatorima izraza. Upoređivali smo performanse sve tri šeme izvršavanja. Pošto marljiva strategija zahteva inicijalizaciju i izračunavanje svih osobina, bilo da su referencirane ili ne. Stoga se ona smatra da je to najgora strategija, koja će jasnije naglasiti prednosti druga dva pristupa. Naravno, pretpostavlja se da će lenjo izračunavanje doneti najbolje rezultate, ali se treba isto tako pripaziti šta bi se moglo desiti kad objekti imaju veliki broj osobina.

Posmatra se brzina izračunavanja, kao i memorijski utrošak, tj. broj objekata i zauzeće memorije. Eksperiment se bazira na višestrukum izračunavanju crteža sa različitim brojem objekata sa osobinama koje treba izračunati (Slika 7.1), koristeći sve tri strategije.

Za objekat sa osobinama smo izabrali trougao, jer je jedan od osnovnih oblika koji se uči od prvih razreda u osnovnoj školi, a i rasprostranjen je u upotrebi i za njega postoji veliki broj interaktivnih primera koji su predstavljeni u raznim DGS-ima (*GeoGebra materials: triangle*, 2019; *Cinderella Gallery: Altitudes in a triangle*, 2019; Engstrom, 2001; Arzarello et al., 2002; *Wolfram Math World: Triangle*, 2019; Lernpfad, 2019; *Math Open Reference, Triangles*, 2019; *Cut The Knot: The many ways to construct a triangle*, 2019; *Geometrie interactive - le cercle qui tourne le triangle*, 2019; Dahan, 2014; Kimberling, 2019; Marinković and Janičić, 2012; *Nastavni planovi i programi za osnovne i srednje škole, Zavod za unapređenje obrazovanja i vaspitanja, Republika Srbija*, 2016).

Test primeri su kreirani počinjući sa osnovnom grupom objekata, slično kao kod Wernick (1982), koji se sastoji od trouglova sa sledećim osobinama: visina na stranicu a , simetrala ugla α , simetrala stranice a , centar opisanog kruga O ; opisani krug cc . Za testiranja zauzeća memorije i količine objekata trouglu su dodavane sledeće osobine: obim trougla; stranica a ; stranica b ; podnožje visine na stranicu c ; površina trougla; poluprečnik upisanog kruga; medijana iz A ; pravac medijane kroz B ; pravac medijane kroz C ; ortocentar; X koordinata sredine stranice a ; koordinata Y podnožja visine na stranicu b ; trougao sa vrhovima na sredinama stranica a, b, c ; koordinata X sredine stranice c unutrašnjeg trougla; koordinata Y vrha C unutrašnjeg trougla; koordinata Y sredine stranice b unutrašnjeg trougla; koordinata Y vrha B unutrašnjeg trougla; obim unutrašnjeg trougla. Za dato n kreiran je dodatni broj objekata koristeći konstrukcijske korake kao u Tabelama 7.1 i 7.2. Vrhovi baznog trougla imaju konstantne koordinate, dok se zavisni trouglovi na osnovu njih izračunavaju. Za eksperimente koji se tiču lenjog i marljivog izračunavanja konstrukcije su date u objektno-orijentisanoj sintaksi, dok su za funkcionalni pristup primenjene odvojene funkcije za svaku osobinu. Testiranja su vršena na tri različite konfiguracije (Tabela 7.3).

CPU vreme je mereno na sledeći način (Eksperiment 1 i Eksperiment2): (1) za dato n kreirana je konstrukcija, (2) koordinate tačke A u baznom trouglu su menjane, izazvavši izračunavanje zavisnih objekata, (3) korak 2 je ponavljan više puta i ukupno vreme izvršavanja je mereno.

Eksperiment 1: Na početku smo za $n = 5$ vršili merenja za 50, 100, 200, 300 i 500 ponavljanja.

Eksperiment 2: Pošto nije bilo devijacija u rezultatima, odlučili smo da se dalji eksperimenti vrše sa 100 ponavljanja. Isto merenje je ponavljano za $n = 10, 20, 30, 40$.

Eksperiment vezan za utrošak memorije i broj kreiranih objekata je sproveden na sledeći način (Eksperiment 3): (1) Izmeren je broj objekata na hipu i veličina hipa nakon što je pokrenuta aplikacija; (2) za dato n učitana je konstrukcija; (3) Izmeren je broj objekata na hipu i veličina hipa nakon što je kreirana konstrukcija.

Na početku smo radili eksperimente sa trouglovima i njihovih 9 osobina, za sva tri pristupa (Eksperiment 1 i Eksperiment 2). Pošto su lenja i funkcionalna strategija imali bliske vrednosti, uporedili smo ta dva pristupa ponovo, ali menjajući broj osobina koji se posmatra. Na početku se krenulo sa 3 osobine, sve dok se nije stiglo do 27 osobina (Eksperiment 3).

Za sve tri konfiguracije tokom Eksperimenta 1 je pokazano da strategija lenjog izračunavanja daje najbolje rezultate (Tabela 7.5, Slika 7.2).

Za sve tri konfiguracije tokom Eksperimenta 2 je pokazano da je lenjo izračunavanje bolje od funkcionalnog od 10% do 16%, dok je u marljiva strategija lošija od lenje od 34% - 57% (Slika 7.4, Tabela 7.6).

Kod Eksperimenta 3 sa početnih 9 osobina su lenjo i funkcionalno izvršavanje bili ujednačeni, pa se prešlo na još finiju podelu krenuvši merenje sa 3 osobine i dodajući u svakom novom merenju još 3 nove osobine.

Upoređujući broj objekata, funkcionalan pristup je bolji za 8%-10% kada objekat ima 6 i 9 osobina, oko 5% bolji za objekte sa 12 i 15 osobina, dok je skoro ujednačen za objekte sa 18 i 21 osobinom. Prekretnica u korist lenjog izračunavanja je nakon što objekti imaju više od 24 osobine. Analogan je odnos poređenja zauzeće memorije (Slika 7.6).

Dakle iz svega priloženog zaključujemo da je izvršena verifikacija koncepata i algoritama opisanih u prethodnim poglavljima.

Zaključak i buduće smernice

U ovoj disertaciji je prezentovan okvir za specifikaciju metapodataka koja nosi u sebi strukturalnu i semantičku informaciju u FLG, te je demonstrirana kroz više tipova. Predloženi okvir nudi veću fleksibilnost od atributa, dok i dalje imaju istu funkcionalnost i deklarativnu prirodu. Problem sa čuvanjem vrednosti ne-CLR vrednosnih tipova u metapodacima, kao i strukturnih informacija, koji se nije mogao prevazići koristeći attribute, je lako rešen koristeći naš pristup. Takođe su date smernice za implementaciju novih vrsta podataka, operacija, konverzije tipova i funkcija.

Zahvaljujući raširenoj upotrebi metapodataka za označavanje svih aspekata softvera za dinamičku geometriju, uspešno smo razdvojili tipove podataka, operacije, funkcije i vizuale objekte od osnovnih komponenti softvera. To znači da se softver lako može proširiti novim funkcionalnostima, ili pak, da se podrazumevani skup funkcionalnosti u potpunosti zameni.

Uvodeći tip podataka objektnih konstanti i dot notaciju, izbegli smo komplikovanu sintaksu, i ujedinili objekte sa njihovim osobinama, što pogoduje i korisnicima i programerima podjednako. Sa korisničke tačke gledišta, sintaksa izraza je olakšana i jasnija za korištenje, dok je programerima ujedinjena implementacija objekata zajedno sa svim svojim osobinama unutar jedne C# klase, umesto da se osobine pokrivaju kroz višestruke funkcije.

Zadnjih decenija se softver za dinamičku geometriju izuzetno brzo razvija, a i sve više upotrebljava u nastavi i pravljenju nastavnih materijala. Stoga je porasla potreba za proširivosti softvera za dinamičku geometriju. Smatramo da se principi i metodi istaknuti u ovoj disertaciji mogu lako adaptirati na bilo koji softver za dinamičku geometriju razvijen u C#, Javi ili nekom sličnom jeziku.

Omogućili smo proširenje evaluatora izraza i stabla izraza u SLGeometry koji demonstrira sledeća poboljšanja: omogućuje podršku za objekte, u OOP smislu, koja predstavlja geometrijske objekte i njihove osobine; omogućuje okvir za jednostavnu definiciju i izračunavanje osobina objekata; definiše proširenje stabla izraza i algoritam za lenjo izračunavanje osobina objekata, koji redukuje vreme izračunavanja, računajući samo osobine koje su referencirane, i računajući ih samo jednom; definisana je šema keširanja rezultata koja značajno redukuje zauzeće hipa i ubrzava izračunavanje stabla izraza.

Testirali smo naš koncept upoređujući ga sa čisto funkcionalnom šemom izračunavanja, koja se smatra standardom za softvere za dinamičku geometriju. Mereno je CPU vreme, zauzeće memorije i broj objekata na hipu. Na osnovu eksperimentalnih rezultata predloženi algoritam za lenjo izračunavanje daje veoma dobre rezultate koji se tiču brzine, dok je zauzeće memorije i opterećenje hipa zadovoljavajuće.

Glavni doprinos ove disertacije je predložen sistem za efikasno upravljanje skupom interaktivnih objekata sa dinamičkim osobinama, koji je implementiran kao proširenje klasičnog evaluatora stabla izraza. Efikasnost izračunavanja je dobijena uvođenjem šeme keširanja rezultata, dinamičkom aktivacijom osobina i lenjim izračunavanjem osobina.

Dalji rad se može zasnovati na optimizaciji binarnih operacija. Takođe se može raditi na odvajanju glavnih komponenti koje bi se onda mogle koristiti za druge projekte kao samostane aplikacije.

Biography

Davorka Radaković was born on October 4, 1977 in Zagreb. She graduated from Elementary School "Petar Pre-radović" and Music Elementary School "Zlatko Baloković" in 1992. She graduated from Secondary School "Svetozar Marković" and Music Secondary School "Isidor Bajić" in Novi Sad in 1996. She enrolled studies at Faculty of Sciences, University of Novi Sad, module B. Sc. in Mathematics in 1996. and graduated in 2001. (G.P.A 8.88/10). At the same faculty, she enrolled M. Sc. Studies in 2001. and has passed all the exams with the highest marks (G.P.A 10). She defended Master Thesis "*Extended modular platform for dynamic geometry*" on September 2, 2010, and she enrolled Ph. D. Studies in Informatics.



Since August 1, 2000. she has been employed at Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, at the following positions:

1. laboratory technician, August 1, 2000. – May 2001.
2. systems analyst, May 2001.-April 10, 2011.
3. assistant, scientific field of computer science, April 11, 2011.- April 10, 2017.
4. teaching associate, April 11, 2017. – now.

She has conducted practical exercises in *Introduction to Programming* (for students from Department of Geography) at Department of Mathematics and Informatics, and *Informatics in Sport/Introduction to Informatics* at Faculty of Sport and Physical Education. At the Chair of Computer Science she conducts practical exercises from the following subjects: *Introduction to Programming, Formal Methods Engineering, Data Structures and Algorithms 1 i Software Lab 2 - Web design*. During the several years she, also, had conducted practical exercises from: *Teaching of Informatics, Software Lab 1 - Office management, Seminar paper A – C# and .NET.*, and *Introduction to eBusiness*.

She trainee as teacher of informatics at High School for Children with special needs „dr Milan Petrović“. For many years, she held Preparatory classes from Mathematics at Faculty of Medicine and Faculty of Sciences.

She has participated in the several dozen domestics and international scientific conferences. She coauthored more than 20 papers as conference proceedings and journal articles.

She received ÖAD Short term research scholarship, April, 2009, RISC Institute, Johannes Kepler Universität, Linz, Austria, supervisor: Hanspeter Mössenböck, and in August and September, the same year, DAAD Intensivsprachkursstipendium für ausländische Studierende, Berlin. In June 2011 she received CEEPUS grant research for stay in Cluj, Romania, "Babes Bolyai" University, and in June 2013, 2016, and 2018, she was at the Technical University of Košice, Slovakia. In 2012 and 2013, she participated DAAD Summer School in Ohrid.

During faculty accreditation in 2007, she was a member of Faculty of Sciences Team Accreditation. She was technical support and an organizing committee member for several scientific conferences. During the Department reconstruction she solved the issues of teaching placement at the other faculties. She maintains the

departments website, and she has been scheduling classes and colloquiums for several years. She participated in the first two International Education Fairs „Putokazi“, and in 2017 at Senta Mini Education Fair.

She is fluent in english, german and french.

Biografija (in Serbian)

Davora Radaković je rođena 4.10.1977. godine u Zagrebu. Završila Osnovnu školu "Petar Preradović" i Osnovnu muzičku školu "Zlatko Baloković" 1992. godine. Gimnaziju "Svetozar Marković" i Srednju muzičku školu "Isidor Bajić" završava u Novom Sadu 1996. godine. Prirodno matematički fakultet u Novom Sadu, smer diplomirani matematičar, upisala je 1996. godine, a završila je 2001. godine sa prosekom 8.88. Poslediplomske studije je upisala na istom fakultetu 2001. godine, te položila sve predviđene predmete sa prosečnom ocenom 10. Magistarsku tezu "Proširiva modularna platforma za dinamičku geometriju" je odbranila 2.9.2010. godine. Doktorske akademske studije informatike upisuje 2010. godine.



Zaposlena je na Departmanu za matematiku i informatiku Prirodno-matematičkog fakulteta, Univerziteta u Novom Sadu, od 1.8.2000. godine i bila u sledećim zvanjima:

1. laborant, 1. avgust 2000. – maj 2001.
2. sistem analitičar, maj 2001.-10. april 2011.
3. asistent, uža naučna oblast računarske nauke, 11. april 2011.- 10. april 2017.
4. saradnik u nastavi, 11. april 2017. – sada.

Držala je vežbe iz predmeta *Uvod u računarstvo* (za studente Departmana za geografiju) na Departmanu za matematiku i informatiku i *Informatiku u sportu/Uvod u informatiku i računarstvo* na Fakultetu sporta i fizičkog vaspitanja. Na Katedri za računarske nauke trenutno drži vežbe iz sledećih predmeta: *Uvod u programiranje, Formalni metodi u inženjerstvu, Strukture podataka i algoritmi 1* i *Softverski praktikum 2 - Veb dizajn*. Tokom godina je držala i vežbe iz predmeta: *Metodika informatike, Softverski praktikum 1 - Kancelarijsko poslovanje, Seminarski rad A – C# i .NET.* i *Uvod u elektronsko poslovanje*. Volontirala je kao nastavnik informatike u Srednjoj školi za decu sa posebnim potrebama „dr Milan Petrović“. Dugi niz godina drži pripremnu nastavu iz matematike na Medicinskom fakultetu i Prirodno-matematičkom fakultetu. Učestvovala je na više desetina domaćih i međunarodnih konferencija. Ima objavljenih više od 20 naučnih i stručnih radova u domaćim i stranim časopisima (od toga jedan na SCI listi) i saopštenja sa međunarodnih i domaćih skupova štampanih u celosti i izvodu.

Tokom aprila 2009. godine boravila je na Institutu za sistemsko inženjerstvo Johannes Kepler Univerziteta u Linzu kao CEEPUS stipendista, a avgusta i septembra iste godine je boravila u Berlinu kao DAAD stipendista. Kao CEEPUS stipendista juna 2011. godine boravi na Univerzitetu "Babes Bolyai" u Klužu, Rumunija, a juna 2013., 2016. i 2018. godine boravi na Tehničkom Univerzitetu u Košicama, Slovačka. Letnje intenzivne DAAD kurseve u Ohridu pohađa 2012. i 2013. godine.

Tokom akreditacije fakulteta 2007. godine bila je član Akreditacionog tima PMF-a. Kao tehnička podrška i član organizacionog odbora sudeluje u organizovanju više različitih naučnih skupova. Za vreme rekonstrukcije Departmana rešava

pitanje smještaja nastave na drugim fakultetima. Održava sajt DMI, a više godina je pravila raspored časova i kolokvijuma. Kao predstavnik DMI sudelovala je na prva dva Međunarodna sajma obrazovanja „Putokazi“ i na Mini-sajmu obrazovanja u Senti 2017. godine.

Služi se engleskim, nemačkim i francuskim jezikom.

University of Novi Sad
Faculty of Science
Key Words Documentation

Accession number:
 NO

Identification number:
 INO

Document type: Monograph documentation
 DT

Type of record: Textual printed material
 TR

Contents code: Doctoral dissertation
 CC

Author: Davorka Radaković, M.Sc.
 AU

Mentor: Dr. Miloš Radovanović
 MN

Title: Metadata-Supported Object-Oriented Extension of
 Dynamic Geometry Software
 TI

Language of text: English
 LT

Language of abstract: Serbian/English
 LA

Country of publication: Serbia
 CP

Locality of publication: Vojvodina
 LP

Publication year: 2019
 PY

Publisher: Author's reprint
 PU

Publ. place: Novi Sad, Trg D. Obradovića 4
 PP

Physical description: 8/175 (xxvi + 149)/241/19/31/0/0
 (no. chapters/pages/bib. refs/tables/figures/graphs/appendices)
 PO

Scientific field: Computer Science
 SF

Scientific discipline: Software Engineering

SD

Subject/Key words: Source code annotations; Metadata; Dynamic Geometry Software; Component development; Functional languages; Lazy evaluation

SKW

UC

Holding data: Library of Department of Mathematics and Informatics, Faculty of Sciences, Trg Dositeja Obradovića 4, Novi Sad, Serbia

HD

Note: None

N

Abstract: Nowadays, Dynamic Geometry Software (DGS) is widely accepted as a tool for creating and presenting visually rich interactive teaching and learning materials, called dynamic drawings. Dynamic drawings are specified by writing expressions in functional domain-specific languages. Due to wide acceptance of DGS, there has arisen a need for their extensibility, by adding new semantics and visual objects (*visuals*). We have developed a programming framework for the Dynamic Geometry Software, SLGeometry, with a genericized functional language and corresponding expression evaluator that act as a framework into which specific semantics is embedded in the form of code annotated with metadata. The framework transforms an ordinary expression tree evaluator into an object-oriented one, and provide guidelines and examples for creation of interactive objects with dynamic properties, which participate in evaluation optimization at run-time. Whereas other DGS are based on purely functional expression evaluators, our solution has advantages of being more general, easy to implement, and providing a natural way of specifying object properties in the user interface, minimizing typing and syntax errors.

SLGeometry is implemented in C# on the .NET Framework. Although attributes are a preferred mechanism to provide association of declarative information with C# code, they have certain restrictions which limit their application to representing complex structured metadata. By developing a metadata infrastructure which is independent of attributes, we were able to overcome these limitations. Our solution, presented in this dissertation, provides extensibility to simple and complex data types, unary and binary operations, type conversions, functions and visuals, thus enabling developers to seamlessly add new features to SLGeometry by implementing them as C# classes annotated with metadata. It also provides insight into the way a domain specific functional language of dynamic geometry software can be genericized and customized for specific needs by extending or restricting the set of types, operations, type conversions, functions and visuals.

Furthermore, we have conducted experiments with several groups of students of mathematics and high school pupils, in order to test how our approach compares to the existing practice. The experimental subjects tested mathematical games using interactive visual controls (UI controls) and sequential behavior controllers.

Finally, we present a new evaluation algorithm, which was compared to the usual approach employed in DGS and found to perform well, introducing advantages while maintaining the same level of performance.

AB

Accepted on Senate: March 8th 2017

AS

Defended:

DE

Thesis Defend Board:

(Degree/first and last name/title/faculty)

DB

President: Dr. Zoran Budimac, full professor,
University of Novi Sad, Faculty of Sciences

Mentor: Dr. Miloš Radovanović, associate professor,
University of Novi Sad, Faculty of Sciences

Member: Dr. Đorđe Herceg, full professor,
University of Novi Sad, Faculty of Sciences

Member: Dr. Milan Vidaković, full professor,
University of Novi Sad, Faculty of Technical Sciences

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Ključna dokumentacijska informacija

Redni broj:
RBR

Identifikacioni broj:
IBR

Tip dokumentacije: Monografska dokumentacija
TD

Tip zapisa: Tekstualni štampani materijal
TZ

Vrsta rada: Doktorska disertacija
VR

Autor: mr Davorka Radaković
AU

Mentor: dr Miloš Radovanović
MN

Naslov rada: Objektno-orijentisano proširenje softvera za
dinamičku geometriju podržano metapodacima

NR

Jezik publikacije: engleski
JP

Jezik izvoda: srpski/engleski
JI

Zemlja publikovanja: Srbija
ZP

Uže geografsko područje: Vojvodina
UGP

Godina: 2019
GO

Izdavač: autorski reprint
IZ

Mesto i adresa: Novi Sad, Trg D. Obradovića 4
MA

Fizički opis rada: 8/175 (xxvi + 149)/241/19/31/0/0
(broj poglavlja/strana/lit. citata/tabela/slika/grafika/priloga)

FO

Naučna oblast: Računarske nauke
NO

Naučna disciplina: Softversko inženjerstvo

ND

Predmetna odrednica

/Ključne reči:

Anotacija izvornog koda; Metapodaci; Softveri za dinamičku geometriju; Razvoj zasnovan na komponentama; Funkcionalni jezici; Lenjo izračunavanje

PO

UDK

Čuva se:

Biblioteka Departmana za matematiku i informatiku, Prirodno-matematički fakultet, Trg Dositeja Obradovića 4, Novi Sad, Srbija

ČU

Važna napomena:

Nema

VN

Izvod:

U današnje vreme softver za dinamičku geometriju (DGS) je široko prihvaćen kao alat za kreiranje i prezentovanje vizuelno bogatih interaktivnih nastavnih materijala i materijala za samostalno učenje, nazvanih dinamičkim crtežima. Kako je raslo prihvatanje softvera za dinamičku geometriju, tako je i rasla potreba da se oni proširuju, dodajući im novu semantiku i vizualne objekte. Razvili smo programsko okruženje za softver za dinamičku geometriju, SLGeometry, sa generičkim funkcionalnim jezikom i odgovarajućim evaluatorom izraza koji čini okruženje u kom su ugrađene specifične semantike u obliku koda označenog metapodacima. Ovo okruženje pretvara uobičajen evaluator stabla izraza u objektno orijentiran, te daje uputstva i primere za stvaranje interaktivnih objekata sa dinamičkim osobinama, koji sudeluju u optimizaciji izvršenja tokom izvođenja. Dok se drugi DGS-ovi temelje na čisto funkcionalnim evaluatorima izraza, naše rješenje ima prednosti jer je uopštenije, lako za implementaciju i pruža prirodan način navođenja osobina objekta u korisničkom interfejsu, minimizirajući kucanje i sintaksne greške.

SLGeometry je implementirana u jeziku C# .NET Framework-a. Iako su atributi preferiran mehanizam, koji povezuje C# kôd sa deklarativnim informacijama, oni imaju određena ograničenja koja limitiraju njihovu primenu za predstavljanje složenih strukturiranih metapodataka. Razvijanjem infrastrukture metapodataka koja je nezavisna od atributa, uspeli smo prevladati ta ograničenja. Naše rešenje, predstavljeno u ovoj disertaciji, pruža proširivost: jednostavnim i složenim vrstama podataka, unarnim i binarnim operacijama, konverzijama tipova, funkcijama i vizuelnim objektima, omogućavajući time programerima da neprimetno dodaju nove osobine u SLGeometry implementirajući ih kao C# klase označene metapodacima.

Takođe, okruženje pruža uvid na koji se način jezik specifičan za funkcionalni domen softvera za dinamičku geometriju može napraviti generičkim i prilagođen specifičnim potrebama: proširivanjem ili ograničavanjem skupa tipova, operacija, konverzija tipova, funkcija i vizuelnih elemenata.

Pored toga, sproveli smo nekoliko eksperimenata sa više grupa studenata matematike i srednjoškolaca, sa ciljem da testiramo kako se naš pristup može uporediti sa postojećom praksom. Tokom eksperimenata testirane su matematičke igre koristeći interaktivne vizualne kontrole (UI kontrole) i kontrolere sekvencijalnog pon-
ašanja.

Na kraju je prikazan i novi algoritam za izračunavanje, koji se pokazao uspešnim u poređenju sa uobičajenim pristupom koji se koristi u DGS-ima, a opet uvodeći prednosti dok je zadržao istu razinu performansi.

IZ

Datum prihvatanja teme od strane Senata: 08.03.2018.

DP

Datum odbrane:

DO

Članovi komisije:

(Naučni stepen/ime i prezime/zvanje/fakultet)

KO

Predsednik: dr Zoran Budimac, redovni profesor,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Mentor: dr Miloš Radovanović, vanredni profesor,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Član: dr Đorđe Herceg, redovni profesor,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Član: dr Milan Vidaković, redovni profesor,
Univerzitet u Novom Sadu, Fakultet tehničkih nauka