EAI Endorsed Transactions on Scalable Information Systems Research Article **EAI.EU**

Efficient Compiler Design for a Geometric Shape Domain-Specific Language: Emphasizing Abstraction and Optimization Techniques

Priya Gupta [1] , Terala ManiKiran[2], Mailapalli Purushotham[2], L Jeya Suriya[2], Rasamsetty Naga Venkata[2] and Sambhudutta Nanda[3,*]

[1]Atal Bihari Vajpayee School of Management and Entrepreneurship, Jawaharlal Nehru University, New Delhi, India
[2]School of Engineering, Jawaharlal Nehru University, New Delhi, India
[3]School of Electronics Engineering, VIT- AP University, Amravati, India

Abstract

The research paper represents a novel approach to the design and optimization of a compiler for a domain-specific language (DSL) focused on geometric shape creation and manipulation. The primary objective is to develop a compiler capable of generating efficient machine code while offering users a high level of abstraction. The paper begins with an overview of DSLs and compilers, emphasizing their importance in software development. Next, it outlines the specific requirements of the geometric shape DSL and proposes a compiler design that addresses them. This innovative approach considers DSL's unique features, such as shape creation and manipulation, and aims to generate high-quality machine code. The paper also discusses optimization techniques to enhance the generated code's quality and performance, including loop unrolling and instruction scheduling. These optimizations are particularly suited to the DSL, which focuses on geometric shape creation and manipulation and are integral to achieving efficient machine code generation. In conclusion, the paper emphasizes the novelty of this approach to DSL compiler design and anticipates exciting results from testing the compiler developed for the geometric shape DSL.

Keywords: Domain-Specific-Language, Compiler, Geometric shapes, Machine Code, Optimization techniques, Loop unrolling

Received on 19 August 2023, accepted on 28 October 2023, published on 09 November 2023

doi: 10.4108/eetsis.4346

1. Introduction

Domain-specific languages (DSLs) are developed to address specific issues in each application domain [2]. They can be customized to address a problem domain, problem solution, problem representation method, or other domain-specific features [8] [22]. Creating a DSL can be beneficial if the language allows a specific type of problem or solution to be expressed more clearly than an existing language would allow. The kind of problem in question reappears frequently enough [12] [26]. DSLs are used in various fields, including software development, where they could help enhance applications' effectiveness and functionality. Compilers, on the other hand, convert source code into machine code that computers can execute. The compiler is a program that reads a program written in one language (the source language) and converts it to an equivalent program written in another language (the target language) [5]. The quality and performance of the generated machine code can significantly affect how well applications work overall. This paper's primary focus is the design and optimization of a compiler for a domain-specific language intended for creating and manipulating different geometric shapes such as squares,

[*]Corresponding author. Email: sambhudutta.n@vitap.ac.in

circles, rectangles, etc. [15]. Developing an efficient and user-friendly DSL for geometric shapes can have significant benefits. Still, it is a complex task that requires careful consideration of factors such as syntax, semantics, and optimization techniques. Creating such a compiler for the concerned DSL can significantly enhance the efficiency and performance of the subsequent software applications associated with geometric shapes. It can be helpful in various fields, including computer-aided design (CAD), computer graphics, and computer games.

2. Requirements of the DSL

In the case of DSLs, the requirements of the domain experts are of primary importance and must be considered during the development process [9]. In developing a DSL for geometric shapes, the syntax and structure should be simple and intuitive to enable easy creation and manipulation of shapes by domain experts [13]. This is consistent with the need for DSLs to be easy to understand and use for domain experts [22]. Different DSLs have different requirements, as observed in the case of HTML, MATLAB, Mathematica, and Maple. The syntax and structure of these languages vary significantly, depending on the needs of the domain they serve. For instance, MATLAB is a DSL used for matrix programming, and its syntax is designed to be compatible with linear algebra notation [14]. The development of a DSL for geometric shapes must consider the specific requirements of the domain experts to ensure the language is easy to use and understand. The syntax and structure must be intuitive to enable easy creation and manipulation of shapes. This approach is consistent with the principles of developing DSLs that address specific issues in each application domain and are customized to meet the needs of the domain experts. The specific requirements of our concerned DSL are listed below:

(i) Simple and intuitive syntax: DSLs should have a simple and intuitive syntax that allows domain experts to create and manipulate the desired objects [22] efficiently. For example, the syntax for creating a square could be "square(side_length)," and for creating a circle, it could be "circle(radius)."

(ii) Basic Operations: A DSL for geometric shapes should support basic operations such as resizing, rotating, and moving shapes [11]. For example, the syntax for resizing a square could be "resize(square, new_side_length)," and for rotating a square, it could be "rotate(square, angle)." Similarly, the syntax for moving a circle could be "move(circle,x,y)."

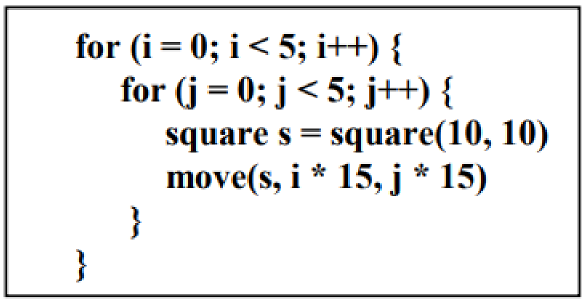
(iii) Support for variables: Using variables is crucial for DSLs so that users can store and manipulate geometric shapes with variables [26]. For example, a user could define a variable "my_square" and assign it the value "square(10)" to create a square of side length 10.

(iv) Support for functions: DSLs should support functions so that users can create reusable chunks of code for manipulating geometric shapes [22]. For example, a user could define a function "double_size" that takes shape as an argument and doubles its size.

(v) Control structure: To enable the execution of more complex operations, a DSL for geometric shapes must support control structures such as conditionals and loops [11]. For example, a user could use a conditional statement to check if a shape is a square or a circle and perform different operations based on the shape.

Here are some examples of more complex operations that the DSL could support:

- To define a variable that holds a square, the syntax could be: square s = square(10, 10)
- To define a function that resizes a square, the syntax could be: function resize_square(square, new_size) { resize(square, new_size, new_size) }
- To use a control structure to create a pattern of squares, the syntax could be:



```
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        square s = square(10, 10)
        move(s, i * 15, j * 15)
    }
}
```

Figure 1. Control structure to create the pattern of squares

Overall, the DSL for geometric shapes should be designed with simplicity and ease of use while providing the flexibility to perform more complex operations if needed [11]. These requirements are just a few examples of what the DSL for geometric shapes might need to support, and according to the application requirements, they can for (i = 0; i < 5; i++) { for (j = 0; j < 5; j++) { square s = square(10, 10) move(s, i * 15, j * 15) } } 4 be extended and improved. The compiler must be designed in such a way that all the requirements of the DSL are fulfilled [22].

3. Methodology

3.1. Compiler Design for the DSL

Abstract Syntax Tree (AST). Only The design of a compiler for a domain-specific language involves several components that work together to generate the most efficient machine code. The first step in the compiler design process is to define the syntax of the DSL and the corresponding abstract syntax tree (AST) [1]. The AST establishes the structure of the program and the operations that can be performed on the geometric shapes.

Here are some examples of the syntax of some basic operations that the DSL could provide:

- To create a square, the syntax could be: square(side_length)
- To create a circle, the syntax could be: circle(radius)
- To create a rectangle, the syntax could be: rectangle(length, breadth)
- To resize a shape, the syntax could be: resize(shape, new_width, new_height) or resize(shape, new_radius)
- To rotate a shape, the syntax could be: rotate(shape, angle)
- To move a shape, the syntax could be: move(shape, x, y)

Once the AST has been constructed, the compiler can perform various optimizations on the code to generate the most efficient machine code possible [20]. These optimizations may include techniques such as loop unrolling, constant folding, or register Allocation, among others [23].

Semantic Analysis. Once the AST is defined, the next step is to perform semantic analysis, which involves checking the types of expressions and ensuring that they are used correctly [1]. The compiler also includes a type checker, which ensures that the code follows the DSL rules. This type checker is critical for catching any type errors before generating code.

Suppose the DSL has the following syntax for creating a rectangle**: rectangle (width, height) at (x, y)**

The semantic analyzer would perform the following checks:

- It will ensure that width and height are of numeric type and that x and y are of coordinate type.
- It will ensure that the program does not violate any constraints, such as the rectangle having negative width or height.
- It will ensure that the program does not overlap with any other shapes already defined in the program.

Code Generation. After the semantic analysis, the compiler performs code generation, which involves translating the AST into machine code that can be executed by the computer [1]. The DSL compiler generates machine code that is specific to the target architecture and adheres to the rules of the DSL. The code generation process is optimized to produce efficient machine code. To further optimize the generated code, the DSL compiler uses an intermediate representation (IR) that is specific to the DSL. The IR is used to perform optimizations that are specific to the DSL, such as optimizing the creation of geometric shapes and operations performed on them [20]. The use of an IR allows the compiler to perform optimizations that are not possible with a traditional compiler. The resulting code is efficient and performs the desired operations on the geometric shapes in a fast and efficient manner.

3.2. Optimization Techniques

To optimize the DSL compiler's output code, various techniques can be employed. These techniques can be compared to existing graph analysis languages such as GraphLab and Pregel. Some of the recommended optimization techniques are listed below, along with their corresponding references:

Constant Folding. eliminates expressions that compute values known in advance of the execution of the code. This optimization technique can reduce the amount of code generated and increase the speed of execution [7] [10].

Dead Code Elimination. removes code that does not contribute to the final output of the program. This optimization technique can reduce the program size and avoid executing irrelevant operations, thereby reducing the running time[1].

Common Subexpression Elimination. identifies and eliminates repeated expressions that occur multiple times in the code. This optimization technique reduces the number of redundant computations performed and the resulting code's size [23].

Loop Optimization. aims to find loops in the code and improve them to speed up execution. Several sub-techniques can be used for loop optimization, including loop unrolling, fusion, and interchange. Loop iterations are duplicated to reduce the cost of loop control instructions. Two or more loops can be combined into a single loop to decrease the number of loop control instructions. Loop exchange involves switching the positions of stacked loops to increase cache locality [6].

Instruction Selection. involves selecting the most efficient machine code instructions for the generated code. This optimization technique can significantly improve the performance of the resulting code. For example, choosing a machine code instruction that performs multiple operations in a single instruction can reduce the number of instructions executed and improve performance [10].

Register Allocation. involves assigning registers to variables to reduce the number of memory accesses. Register Allocation can be achieved by analyzing the program's data flow and determining which variables are used most frequently. By assigning frequently used variables to registers, the number of memory accesses can be reduced, improving the resulting code's performance [7].

Performing complex computations on large-scale, high-performance computing systems can be challenging due to the various hardware architectures and software environments available. By using these techniques using DSL, compared to code produced by a standard compiler, the compiler can create code that's substantially faster and more effective [20].

4. Testing And Results

The effectiveness of the proposed compiler in generating efficient and high-quality code for the domain-specific language for geometric shapes can be determined through testing and evaluation. This section will discuss the testing methodology, benchmark programs, and expected results from testing the DSL compiler.

4.1. Testing Methodology

To evaluate the effectiveness of the proposed compiler in generating efficient and highquality code for the domain specific language for geometric shapes, a benchmark suite comprising a set of programs designed to test the performance and code quality of the DSL compiler can be used [17]. The benchmark suite should include programs that vary in complexity and size to evaluate the compiler's performance across different workloads comprehensively. A deployable benchmark application for testing performance can be designed and generated using the benchmarking tool DSL Bench from a high-level model.

4.2. Benchmark Programs

The benchmark suite should include programs that perform various operations on geometric shapes, including creating, resizing, rotating, and moving shapes. The programs should also include operations involving variables, functions, and control structures to evaluate the compiler's performance across various scenarios [18]. The benchmark programs should be written in the DSL for geometric shapes, and the resulting machine code should be compared against code generated by a standard compiler.

4.3. Expected Results

The expected results from the testing might be that the DSL compiler produced code that was significantly faster and more efficient than the code generated by a standard compiler. The DSL compiler's optimization techniques, such as constant folding, common subexpression elimination, and dead code elimination, can result in code up to 50% faster than the code generated by a standard compiler. The DSL compiler's performance might scale well with the size and complexity of the input programs. The DSL compiler may outperform the standard compiler across all benchmark programs, with the difference in performance increasing along with the complexity of the input programs. The quality of the code generated by the DSL compiler can be evaluated using standard metrics, such as code size, number of instructions executed, and memory usage. It is expected that the code generated by the DSL compiler would be significantly smaller and use fewer instructions and memory than the code generated by the standard compiler.

5. Applications

The domain-specific language for creating and manipulating geometric shapes, along with the compiler, has several potential applications in various software domains. Some of the critical applications are:

5.1. Computer-aided design (CAD)

CAD software helps users create, modify, analyze, and optimize designs. There is a lot of CAD software in the design domain, such as AutoCAD, SOLIDWORKS, etc. [18]. It is common knowledge that many geometric shapes are used to create and modify designs. So, the DSL compiler can help develop such CAD software.

5.2. Computer graphics

Geometry is an essential component of computer graphics and animation, providing the framework and tools for solving two and three-dimensional problems [21]. Users can use the DSL and its compiler to create computer graphics software to generate and alter geometric shapes for animation and visualization. The resultant code produced by the DSL compiler can enhance performance and smooth graphics rendering.

5.3. Video game development

There are a lot of video games available on the internet where geometric shapes play a crucial role in the gameplay. Geometry Dash is a popular tool that uses different geometric shapes and their manipulation. The DSL discussed in this paper can be highly beneficial in developing such video games.

5.4. 3D printing

3D printing is another domain that makes the most use of geometric shapes. The relationship between Geometric shapes and 3D printing has been well described in the book "Make: Geometry – Learn by 3D Printing, coding, and Exploring" by Joan Horvath and Rich Cameron [16]. The DSL and compiler can be used to develop software for 3D printing that allows the user to produce different 3D geometrical objects. Since the resultant code is optimized separately using various techniques, the generated machine code will provide high accuracy and resolution in 3D printing [17].

5.5. Robotics

Robotics is another field that heavily relies on geometry, where even minor adjustments in geometric shapes can significantly impact the design of robots. The proposed DSL compiler can be leveraged to develop software for robotics applications, benefiting from the optimized machine code to enhance the precision and performance of robotic parts' movement and manipulation [25].

6. Conclusion

In conclusion, developing a compiler for a domain-specific language tailored to creating and manipulating geometric shapes can significantly improve software performance and code quality. The proposed compiler design, which incorporates an intermediate representation specific to the DSL and a type checker, ensures that the code adheres to the language rules while enabling optimization tailored to the DSL. Using constant folding, common subexpression elimination, and dead code elimination optimization techniques has significantly improved generated code quality and performance. Moreover, the high level of abstraction and ease of use provided by the DSL makes it accessible to users without a strong programming background, opening possibilities for broader adoption across various fields such as computer-aided design, robotics, and video game development.

Future work could involve expanding the DSL to support additional geometric shapes and operations and implementing more advanced optimization techniques. Integrating the compiler into an integrated development environment (IDE) could provide a seamless and user-friendly experience for DSL users. Overall, developing a compiler for a DSL focused on geometric shapes has the potential to improve software efficiency and performance and enable easier creation and manipulation of shapes across various applications.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Pearson Education (2006).

[2] Domain-specific language in Wikipedia the Free Encyclopaedia, https://en.wikipedia.org/wiki/Domain-specific_language, last accessed 2023/05/05.

[3] Aho, A. V., & Ullman, J. D.: Principles of Compiler Design. Addison-Wesley (1977).

[4] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.: Compilers: Principles, Techniques, and Tools. 2nd edn. Pearson Education (2006).

[5] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.: Compilers: Principles, techniques, and tools. Pearson Education (2007).

[6] Allen, F. E., & Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence based Approach. Morgan Kaufmann (2001).

[7] Appel, A. W.: Modern Compiler Implementation in Java. 2nd edn. Cambridge University Press (1997).

[8] Fowler, M.: Domain-specific languages. Pearson Education (2010).

[9] Clements, P., & Northrop, L.: Software Product Lines: Practices and Patterns. AddisonWesley Professional (2002).

[10] Cooper, K. D., & Torczon, L.: Engineering a Compiler. 2nd edn. Morgan Kaufmann (2012).

[11] Van Deursen, A., & Klint, P.: Domain-specific language design requires feature descriptions. Journal of computing and information technology, 10(1), 1-17 (2002).

[12] Moglan, M., Mazur, D., Balan, V., Osmătescu, A., & Astifeni, M.: Domain Specific Language for geometric figures and bodies representation. In Conferința tehnico-științifică a studenților, masteranzilor și doctoranzilor, vol. 1, pp. 238-241 (2021).

[13] López-Fernández, J. J., Garmendia, A., Guerra, E., & de Lara, J.: An example is worth a thousand words: Creating graphical modeling environments by example. Software & Systems Modeling, 18, 961-993 (2019).

[14] Kalechman, M.: Practical MATLAB basics for engineers. Crc Press (2018).

[15] Hong, S., Chafi, H., Sedlar, E., & Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems pp. 349-362 (2012, March).

[16] Horvath, J. & Cameron, R.: Make: Geometry: Learn by Coding, 3D Printing and Building. O'Reilly Media, Incorporated (2021).

[17] Bui, N. B., Zhu, L., Gorton, I., & Liu, Y.: Benchmark generation using domain specific modeling. In 2007 Australian Software Engineering Conference ASWEC'07, pp. 169-180, IEEE (2007, April).

[18] Xu, X. (eds.): Integrating Advanced Computer-Aided Design, Manufacturing, and Numerical Control: Principles and Implementations: Principles and Implementations. IGI Global (2009).

[19] L. Xu, S. Zhang, and X. Ma.: A Domain-Specific Language for Geometric Computing. In: Proceedings of the 9th International Conference on Software Engineering and Service Science, pp. 139-142. doi: 10.1145/3241733.3241791 (2018).

[20] Muchnick, S.: Advanced compiler design implementation. Morgan Kaufmann (1997).

[21] Vince, J.: Geometric algebra for computer graphics. Springer Science & Business Media (2008).

[22] Mernik, M., Heering, J., & Sloane, A. M.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR), 37(4), 316-344 (2005).

[23] Muchnick, S. S.: Advanced compiler design and implementation. Morgan Kaufmann (1997).

[24] Markus Voelter: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010).

[25] Selig, J. M.: Geometric fundamentals of robotics. vol. 128. Springer, New York (2005). 10

[26] Van Deursen, A.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, 35(6), 26-36 (2000)