LANGUAGE DESIGN FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Dietrich Geisler

August 2024

LANGUAGE DESIGN FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

Dietrich Geisler, Ph.D.

Cornell University 2024

In domains that deal with physical space and geometry, programmers need to track the coordinate systems that underpin a computation. We identify a class of *geometry bugs* that arise from confusing which coordinate system a vector belongs to. These bugs are not ruled out by current languages for vector-oriented computing, are difficult to check for at run time, and can generate subtly incorrect output that can be hard to test for.

We introduce a type system and language that prevents geometry bugs by reflecting the coordinate system for each geometric object. A value's *geometry type* encodes its reference frame, the kind of geometric object (such as a point or a direction), and the computational representation (such as Cartesian or spherical coordinates). We show how these types can rule out geometrically incorrect operations, and we show how to use them to automatically generate correct-by-construction code to transform vectors between coordinate systems. We implement a language for graphics programming, Gator, that checks geometry types and compiles to OpenGL's shading language, GLSL. Using case studies, we demonstrate that Gator can raise the level of abstraction for shader programming and prevent common errors without inducing significant annotation overhead or performance cost.

BIOGRAPHICAL SKETCH

Your biosketch goes here. Make sure it sits inside the brackets.

This document is dedicated to all Cornell graduate students.

ACKNOWLEDGEMENTS

Your acknowledgements go here. Make sure it sits inside the brackets.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

CHAPTER 2

GATOR: GEOMETRY TYPES FOR GRAPHICS PROGRAMMING

CHAPTER 3

ONLINE VERIFICATION OF COMMUTATIVITY

CHAPTER 4

CAIMAN: DSL FOR OPTIMIZING HETEROGENEOUS PROGRAM COMMUNICATION (CAIMAN)

4.1 Introduction

Having a concrete example is helpful when understanding the design and behavior of the Caiman language. For this example, we will be building the `select_sum` function, which takes in three arrays ‘v1‘, ‘v2‘, and ‘v3‘, and returns the sum of either ‘v2‘ or ‘v3‘ depending on the sign of ‘v1‘.

A naive `C` solution for this might look something like the following (for some fixed size of array):

```
int select_sum(int[] v1, int[] v2, int[] v3) {
  if (sum(v1) < 0) {
    return sum(v2);
  }
  else {
    return sum(v3);
  }
}
```

This code is fairly straightforward to understand, and has the significant upside of having its behavior exactly dependent on the abstraction of using an existing (working) function. This approach, however, is not necessarily performant. While we can often rely on an optimizing compiler to work out a reasonably fast arrangement of this code, the compiler cannot always assume the *intent* of the programmer, restricting potential optimizations. As a result, the programmer may need to rewrite this code to change performance, but without changing the semantic meaning of this code:

```
int select_sum(int[] v1, int[] v2, int[] v3) {
  int sum2 = sum(v2);
  int sum3 = sum(v3);
  if (sum(v1) < 0) {
    return sum2;
```

```
  }
  else {
    return sum3;
  }
}
```

4.1.1 Separation of Concerns

4.1.2 Program Decomposability

4.2 Background

WebGPU

CUDA, SYCL

Halide, Scheduling

Continuation Passing Style

Linear Types

4.3 Practical Caiman

To explain how Caiman works, we will work through the example shown in the introduction, namely `select_sum`. For this illustration, we will be using code written the user-facing

Caiman frontend, as opposed to the Caiman IR described in more detail in 4.6.

A Caiman program consists of a series of functions written across two languages: the *specification* language and the *implementation* language. Informally, a function written in the specification language describes the semantic behavior of the program, while a function written in the implementation language describes how that program is implemented on the host machine. This is the core mechanism by which Caiman separates semantic and performance concerns.

Additionally, a Caiman specification must deal with one of three distinct properties of the operation it is specifying: the value calculations, the timeline of the events and synchronizations, and manipulations of existing memory. We give each of these "kinds" of specification functions a unique name, respectively they are the *value*, *timeline*, and *spatial* specifications. Any implementation must implement one of each kind of specification, though specifications may be used by multiple implementation functions.

We will start by showing how to implement `select_sum` in Caiman's specification language(s), before moving onto describing the choice of implementations Caiman provides. We will also examine some intuition of why the Caiman typechecker is able to validate an implementation against a given specification, though the formal model and proof will be deferred to section 4.4.

4.3.1 Specification Languages

Caiman specifications are written as functions with a Rust-like header and declarative bodies. The complete syntax can be summarized as the following:

```
spec_kind name(arg : type) -> return_type {
  var :- expression
```

```
  returns var
}
```

The specific expressions allowed for each variable declaration depend on the kind of specification. All specifications, however, share function calls and ternary conditional expression:

```
// function call
var :- fn_name(expr)
// the usual notion of conditional branches
var :- if cond then e1 else e2
```

The most intuitive Caiman specification to start with is often the *value* function. Our implementation for `select_sum` is as follows:

```
val select_sum(v1: [i64], v2: [i64], v3: [i64]) -> i64 {
  condition :- sum1 < 0
  result :- sum(v2) if condition else sum(v3)
  returns result
}
```

Caiman specifications

Function Equivalence

4.3.2 Implementation Language

Control Flow

4.3.3 Working Example

4.4 Formal Model

4.4.1 Formal Objective

4.4.2 Semantics

4.4.3 Timeline and Spatial Languages

4.5 Explication

4.5.1 Object and Example

4.5.2 Core Algorithm

4.5.3 Engineering

4.6 Caiman Engineering

APPENDIX A

CHAPTER 1 OF APPENDIX

Appendix chapter 1 text goes here

BIBLIOGRAPHY

[1] Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. In *SIGGRAPH*, 2011.

[2] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[3] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Functional programming for compiling and decompiling computer-aided design. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[4] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *SIGGRAPH*, 1990.

[5] Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. In *SIGGRAPH*, 2016.

[6] Dean Jackson and Jeff Gilbert. WebGL specification, 2015. `https://www.khronos.org/registry/webgl/specs/latest/1.0/`.

[7] Andrew J. Kennedy. Dimension types. In *European Symposium on Programming (ESOP)*, 1994.

[8] Andrew J. Kennedy. Relational parametricity and units of measure. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.

[9] The Khronos Group Inc. *The OpenGL ES Shading Language*, 1.0 edition.

[10] Microsoft. Direct3D, 2008. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx`.

[11] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2018.

[12] Jiawei Ou and Fabio Pellacini. SafeGI: Type checking to improve correctness in rendering system implementation. In *Eurographics Conference on Rendering (EGSR)*, 2010.

[13] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[14] Adrian Sampson. Let's Fix OpenGL. In *Summit on Advances in Programming Languages (SNAPL)*, 2017.

[15] Adrian Sampson, Kathryn S McKinley, and Todd Mytkowicz. Static stages for heterogeneous programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[16] Donald J. Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, 15:657–680, 2005.

[17] Mark Segal and Kurt Akeley. *The OpenGL 4.5 Graphics System: A Specification*, June 2017. `https://www.opengl.org/registry/doc/glspec45.core.pdf`.

[18] Sebastian Sylvan. Naming convention for matrix math, 2017. `https://www.sebastiansylvan.com/post/matrix_naming_convention/`.