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

```
select_sum(v1 : [i64; N], v2 : [i64; N], v3 : [i64; N]) -> i64 {
  if sum(v1) < 0 {
    sum(v2)
  }
  else {
    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:

```
select_sum(v1 : [i64; N], v2 : [i64; N], v3 : [i64; N]) -> i64 {
  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 Value Specification

Caiman specifications are written as functions with a Rust-like header and declarative bodies. We emphasize declarative here to mean that declaration order does not matter (as we will see, we are essentially building a dependency graph). The complete syntax can be summarized as follows:

```
// function header, any number of arguments
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 expressions for function calls and ternary conditional expression:

```
// function call
fn_name(expr)
// the usual notion of conditions
if cond then expr1 else expr2
```

The most intuitive Caiman specification to start with is often the *value* function, which describes what calculations are needed for each data value in the program. We will take a deep dive into describing the value specification for `select_sum` before returning to definitions for Caiman's other specification languages in section 4.3.3.

Our value specification for `select_sum` is as follows:

```
val select_sum(v1: [i64; N], v2: [i64; N], v3: [i64; N]) -> i64 {
    sum1 = sum(v1)
    sum2 = sum(v2)
    sum3 = sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

We have written this specification to be more verbose than needed for the sake of providing

a more detailed examination of the semantics being used here. Many of these lines can be condensed or combined (we can just write `if sum(v1) < 0 ...`, for example).

Nevertheless, this declaration more-or-less mirrors our naive C code, notably replacing conditional `if/else` blocks with the ternary expression `if-then-else`. This distinction is more than just syntactic, Caiman's specification language is designed to have no internal control flow, as hinted at by the syntax.

Additionally, since Caiman's operations are unordered (we can freely move or combine each declaration here without changing the specification meaning), we do not have any requirement that these operations must be executed in the order written.

More precisely, this specification gives us constraints on what values this function must produce, but leaves the details up to the actual implementation. We can exactly enumerate these requirements as follows:

- `v1`, `v2`, and `v3` are all arrays of type `i64`, and this function produces data of type `i64` (this is the usual type constraint placed by a function header).
- The specification variables `sum1`, `sum2`, and `sum3` depend on `v1`, `v2`, and `v3`, respectively. For each of these, we must apply `sum` to produce our respective value.
- `condition` depends on having calculated `sum1` already *at some point*, and also have the constant `0` as a value.
- Similarly, `result` depends on having calculated `condition`, `sum1`, and `sum2`, thus requiring the calculations of all of their dependencies to have been done.
- Finally, `returns result` informs us that this function will return the `result` value.

Importantly, however, this specification provides formal requirements; as we will shortly

in subsection 4.3.2, Caiman implementations of this specification which do not produce the specified values will fail to typecheck and will be rejected at compile time. Indeed, we can safely say that each specification variable becomes a type we can refer to while typechecking an implementation function. We must first take a short diversion into examining Caiman function definitions.

Function Equivalence

We need to take a short dive into how the `sum` function is being used in this `select_sum` specification. In Caiman, every function used in a specification must be a *function equivalence class*. Function equivalence classes consist of a name associated with one or more function definitions (which may be defined in Caiman or externally) that the programmer considers equivalent.

The most immediate use for function classes can be seen with the following definition of `sum` that we include in the `select_sum` file:

```
feq sum {
    extern(cpu) pure sum([i64; N]) -> i64
    extern(gpu) pure sum([i64; N]) -> i64
}
```

Here we are simply saying that we consider the `cpu` and `gpu` definitions of `sum` to be equivalent. The key reason for introducing these equivalence classes is to allow for multiple implementations of the same function to coexist in the same program, and to allow the user to fearlessly call any particular version in their implementation of some specification.

Note that `select_sum` must also be in such a function equivalence class – syntactically we name this equivalence class to be the same as our defined value function, in this case

just `select_sum`. In this way, Caiman specification functions can call other specification functions without introducing assumptions about the implementations of those functions.

Importantly, however, Caiman does not attempt to prove this assertion or require any more annotation than exactly the type of the function provided here (the external implementations of `sum` could be buggy, and Caiman makes no claim about preventing this). Similarly, we also make no attempt to verify that an external function implementing a Caiman value specification will match the semantics of that specification, and leave this work to the user. Stylistically, this means that Caiman's guarantees and utility are strongest when a majority of the code being used in a Caiman program is written in Caiman rather than made external.

4.3.2 Implementation Language

With our value specification and function equivalence classes in hand, we can now implement a Caiman program for `select_sum`. As a reminder, all the code we have written so far provides typing information for our actual implementation here, and is not compiled into an executable program without this implementation.

Frontend Caiman implementations more-or-less resemble Rust code, with the addition of specifying which particular specification(s) they are implementing. As noted above, we will only focus on implementing our value specification for now. Our first implementation of `select_sum` is thus as follows:

```
fn select_sum_impl(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    if sum(v1) < 0 {
        sum(v2)
    }
    else {
        sum(v3)
    }
```

```
}
```

This is a valid Caiman implementation of this program, meant to show that in many cases, the programmer can essentially implement code similar to standard languages, where information can mostly be inferred. For understanding the typechecking work Caiman does on this program, however, it is perhaps more informative to show the explicit Caiman type annotations we can provide for such a program. When we hand-write these annotations, we produce the following (equivalent) program:

```
fn select_sum_impl(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    let sum1 : i64 @ node(val.sum1) = sum(v1);
    let condition : bool @ node(val.condition) = sum1 < 0;
    if condition {
        let sum2 : i64 @ node(val.sum2) = sum(v2);
        sum2
    }
    else {
        let sum3 : i64 @ node(val.sum3) = sum(v3);
        sum3
    }
}
```

4.3.3 Value Specification

4.3.4 Working Example

4.4 Formal Model

In this section, we will be describing a formal model of the Caiman language. We will use this to prove our assertion that the Caiman typechecker will ensure a given (typed) implementation has the same (observational) semantics as a specification it implements. We will also be more formally specifying these terms to precisely narrow our claim, and

$$
\begin{aligned}
&\psi \in \mathbf{S} \\
&\phi \in \mathbf{C} \\
&\tau \in \mathbf{T} \\
&d ::= d_1.d_2 \mid \tau\ \psi \text{ :- } \phi(\psi_1) \mid \tau\ \psi \text{ :- if } \psi_1 \text{ then } \psi_2 \text{ else } \psi_3 \\
&p ::= d.\text{returns } \psi
\end{aligned}
$$

Figure 4.1: Hatchling Specification Syntax

$$
\begin{aligned}
&\psi \in \Psi \\
&\tau \in \mathbf{T} \\
&x \in \mathbf{V} \\
&f \in \mathbf{F} \\
&c ::= c_1; c_2 \mid x \leftarrow f_\psi(x_1) \mid \\
&\qquad x = \text{alloc}\,\tau \mid x \leftarrow \text{copy}\,x_1 \mid x \leftarrow \text{ref}\,x_1 \mid \\
&\qquad x \leftarrow \text{select}_\psi(x_1, x_2, x_3) \\
&p ::= c; \text{return}\,x
\end{aligned}
$$

Figure 4.2: Hatchling Implementation Syntax

addressing the limitations of this presented approach.

To do this, we will start by describing a subset of the Caiman IR, a language called *Hatchling*. Hatchling has operations similar to the types and control flow of the Caiman IR, with the goal of describing exactly the type guarantees made by Caiman.

As with the Caiman IR, Hatchling has a specification and an implementation. Our primary goal will be to setup a proof of the following theorem (although the detailed proof itself will be left to Appendix TODO):

Theorem 1. *Any well-typed Hatchling implementation program with types matching those of a well-typed Hatchling specification program must, for all inputs, either fail to terminate or produce equivalent values as that specification program.*

As an observation about this proof, the term *equivalent outputs* relies on a precise

notion of equivalence and values in this context. Specifically, equivalence is defined exactly as any function that is a member of its function class (as described in subsection **??**), while value refers to the series of operations that define some result. In this sense, we observe that the operation 1 + 1 is not considered to be the same value as the constant 2, unless an equivalence is explicitly stated.

The syntax for each of these Hatchling sub-languages are defined in figures 4.1 and 4.2, respectively. These syntax rely on external sets of distinct symbols, namely **T**, **S**, **P**, **V**, and **F**. These sets have no explicit meaning, but intuitively represent the sets of types, specification variables, function classes, implementation variables, and function names, respectively. Function calls in Hatchling only allow a single argument – the extension of these proofs to multiple arguments is straightforward but mechanically irksome. We assume that the set of types **T** includes the type unit, which we use in the typechecking semantics. We also assume the set of specification variable symbols **S** and the set of function classes **C** includes **0**, a special character that cannot be assigned to and indicates a lack of dependence.

A Hatchling specification program additionally requires 2 contexts, 1 dynamic and 1 static:

- $\Psi : \mathbf{S} \mapsto \mathbf{T} \bigtimes (\mathbf{C} \bigtimes (\mathbf{S} \bigtimes \mathbf{S} \bigtimes \mathbf{S}))$, which is dynamic and maps from a specification variable to both its type and its dependencies. A variable depends either on a function or no function (semantically represented with $\top$), and has exactly 0, 1, or 3 dependencies (0 for an input, 1 for a function call, or 3 for a condition).
- $\Phi : \mathbf{C} \mapsto \mathbf{T} \bigtimes \mathbf{T}$, which is static and maps from function classes to the input type and the output type of that function.

A Hatchling implementation program, on the other hand, similarly requires only 1

dynamic context, but requires 3 constant global contexts (where Δ and Ψ are built to be derived from the specification associated with the current implementation). These contexts are as follows:

- $\Gamma : \mathbf{V} \mapsto \mathbf{T} \bigtimes (\mathbf{S} \cup \top)$, the sole dynamic context, which maps from an implementation pointer to the type of the data being referred to by that pointer. This context also includes the associated value if one has been written to the location pointed at by this variable.
- $\Psi : \mathbf{S} \mapsto \mathbf{T} \bigtimes (\mathbf{C} \bigtimes (\mathbf{S} \bigtimes \mathbf{S} \bigtimes \mathbf{S}))$, which must be derived from the result of applying the typechecking rules to a well-typed Hatchling specification. When typechecking an implementation, however, Ψ is global and constant.
- $\Phi : \mathbf{C} \mapsto \mathbf{T} \bigtimes \mathbf{T}$, which is the same as with the specification program
- $\Xi : \mathbf{F} \mapsto \mathbf{C}$, which maps from functions to their owning function class

Finally, both Hatchling programs must specify an output type. For the Hatchling specification program, we denote this as τ_{out}, while for the Hatchling implementation program, we denote this as ψ_{out}. Note that we assume that a function being a member of Φ implies that it is a well–typed function.

4.4.1 Typing Semantics

TODO: the reference semantics need a linear extension, but I don't wanna figure out the best way to condense that right now.

$$\frac{\Psi, \Psi' \vdash d_1 \qquad \Psi', \Psi'' \vdash d_2}{\Psi, \Psi'' \vdash d_1.d_2} \qquad \frac{\Psi, \Psi' \vdash d_2 \qquad \Psi', \Psi'' \vdash d_1}{\Psi, \Psi'' \vdash d_1.d_2}$$

$$\frac{\psi \notin \Psi \qquad \Psi(\psi_1) = (\tau_1, \_) \qquad \Phi(\phi) = (\tau_1, \tau_2)}{\Psi, \Psi \sqcup [\psi \mapsto (\tau_2, (\phi, (\psi_1, \mathbf{0}, \mathbf{0})))] \vdash \tau\, \psi :\text{-}\, \phi(\psi_1)}$$

$$\frac{\psi \notin \Psi \qquad \psi_1 \in \Psi \qquad \Psi(\psi_2) = (\tau, \_) \qquad \Psi(\psi_3) = (\tau, \_)}{\Psi, \Psi \sqcup [\psi \mapsto (\tau, (\top, (\psi_1, \psi_2, \psi_3)))] \vdash \text{if}\, \psi_1\, \text{then}\, \psi_2\, \text{else}\, \psi_3}$$

$$\frac{\Psi, \Psi' \vdash d \qquad \Psi'(\psi) = (\tau_{\text{out}}, \_)}{\Psi, \Psi' \vdash d.\text{return}\, \psi}$$

Figure 4.3: Hatchling Specification Typing Judgment

$$\frac{\Gamma(x) = (\tau, \psi)}{\Gamma, \Gamma[x/(\tau, \top)]} \qquad \frac{\Gamma, \Gamma' \vdash c_1 \qquad \Gamma', \Gamma'' \vdash c_2}{\Gamma, \Gamma'' \vdash c_1; c_2}$$

$$\frac{\Psi(\psi) = (\_, (\phi, (\psi_1, \mathbf{0}, \mathbf{0}))) \qquad \Xi(f) = \phi \qquad \Gamma(x_1) = (\tau, \psi_1)}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x \leftarrow f_\psi(x_1)}$$

$$\frac{}{\Gamma, \Gamma[x/(\tau, \top)] \vdash x = \text{alloc}\, \tau} \qquad \frac{\Gamma(x_1) = (\tau, \psi) \qquad \Gamma(x) = (\tau, \_)}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x = \text{copy}\, x_1}$$

$$\frac{\Gamma(x_1) = (\tau, \psi) \qquad \Gamma(x) = (\tau, \_)}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x = \text{ref}\, x_1}$$

$$\frac{\Gamma(x_i) = (\_, \psi_i) \qquad \Psi(\psi) = (\_, (\top, (\psi_1, \psi_2, \psi_3)))}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x \leftarrow \text{select}_\psi(x_1, x_2, x_3)} \qquad \frac{\Gamma, \Gamma' \vdash c, \Gamma'(x) = \psi_{\text{out}}}{\Gamma, \Gamma' \vdash c; \text{return}\, x}$$

Figure 4.4: Hatchling Implementation Typing Judgment

4.4.2 Operational Semantics

4.5 Explication

4.5.1 Object and Example

4.5.2 Core Algorithm

4.5.3 Engineering

4.6 Caiman Engineering

4.6.1 WebGPU Target and Codegen

4.6.2 Stages of Compilation

4.6.3 High Level Caiman

Translation to Caiman Assembly

4.7 Results

4.7.1 Translating Examples

4.7.2 Timing

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/`.