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

[TODO: update from Gator Abstract] 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. xxx

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

This is an introduction.

CHAPTER 2

GATOR: GEOMETRY TYPES FOR GRAPHICS PROGRAMMING

2.1 Introduction

Applications across a broad swath of domains use linear algebra to represent geometry, coordinates, and simulations of the physical world. Scientific computing workloads, robotics control software, and real-time graphics renderers all use matrices and vectors pervasively to manipulate points according to linear-algebraic laws. The programming languages that express these computations, however, rarely capture the underlying *geometric* properties of these operations. In domains where performance is critical, most languages provide only thin abstractions over the low-level vector and matrix data types that the underlying hardware (i.e., GPU) implements. A typical language might have a basic `vec2` data type for vectors consisting of two floating-point numbers, for example, but not distinguish between 2D vectors in rectangular or polar coordinates—or between points in differently scaled rectangular coordinate systems.

This paper focuses on real-time 3D rendering on GPUs, where correctness hazards in linear algebra code are particularly pervasive. The central problem is that graphics code frequently entangles application logic with abstract geometric reasoning. Programs must juggle vectors from a multitude of distinct coordinate systems while simultaneously optimizing for performance. This conflation of abstraction and implementation concerns makes it easy to confuse different coordinate system representations and to introduce subtle bugs. Figure **??** shows an example: a coordinate system handling bug yields incorrect visual output that would be difficult to catch with testing.

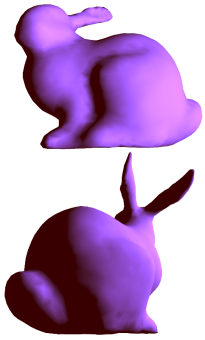

Figure 2.1: Correct implementation.

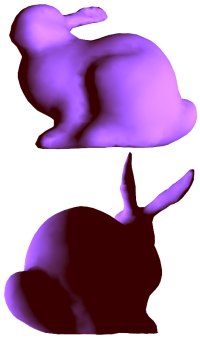

Figure 2.2: Incorrect implementation.

Figure 2.3: Objects rendered with an implementation of the diffuse component of Phong lighting [13], without (a) and with (b) a coordinate system transformation bug. The root cause is an incorrect spatial translation of the light source. The problem is only visible from one side of the model.

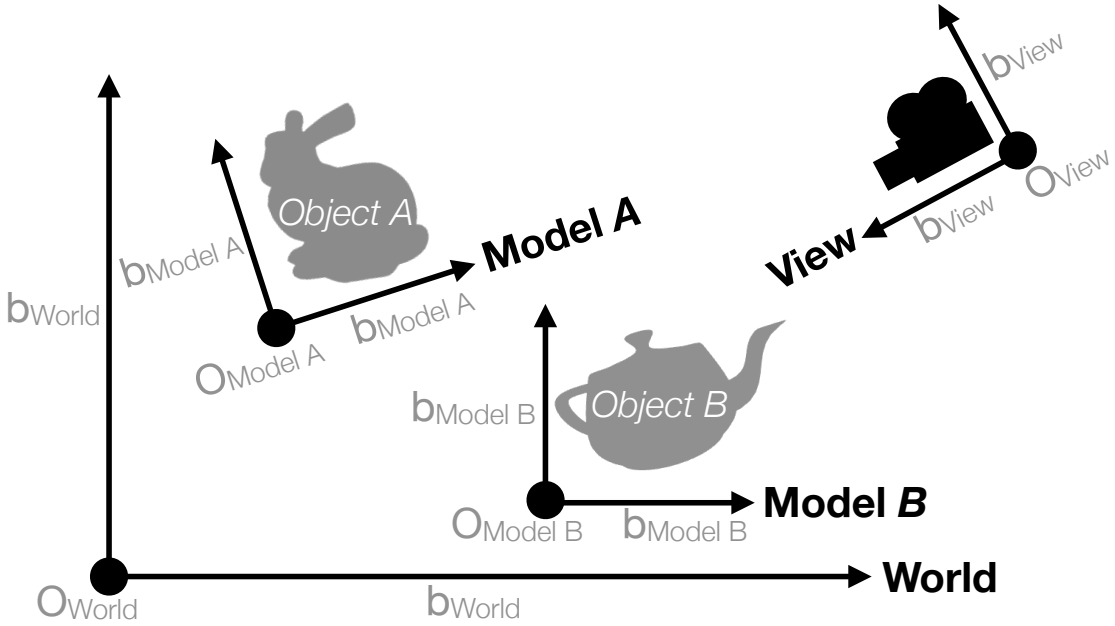

Figure 2.4: Coordinate systems in graphics code. **Model A**, **Model B**, **World**, and **View** are coordinate systems. A coordinate system is defined by its basis vectors b and origin O. The **View** represents the perspective of a simulated camera.

2.1.1 The Problem

Coordinate systems proliferate in graphics programming because 3D scenes consist of many individual objects. Figure 2.4 depicts a standard setup for rendering two objects in a

single scene. Each object comes specified as a *mesh*, which consists of coordinate vectors for each vertex position. The mesh provides these vectors in a local, object-specific coordinate system called *model* space. The application positions multiple objects relative to one another in *world* space, and the simulated camera's position and angle define a *view* space.

Renderer code needs to combine vectors from different coordinate systems, such as in this distance calculation:

```
float dist = length(teapotVertex - bunnyVertex);
```

This code may be incorrect, however, depending on the representation of the `teapotVertex` and `bunnyVertex` vectors. If the values come from the mesh data, they are each represented in their respective model spaces—and subtracting them yields a geometrically meaningless result. A correct computation needs to convert the operands into a common coordinate system using *affine transformation* matrices:

```
float dist = length(teapotToWorld * teapotVertex -
        bunnyToWorld * bunnyVertex);
```

Here, the `teapotToWorld` and `bunnyToWorld` matrices define the transformations from each model space into world space.

Geometry bugs are hard to catch. Mainstream rendering languages like OpenGL's GLSL [17] cannot statically rule out coordinate system mismatches. In GLSL, the variables `teapotVertex` and `bunnyVertex` would both have the type `vec3`, i.e., a tuple of three floating-point numbers. These bugs are also hard to detect dynamically. They do not crash programs—they only manifest in visual blemishes. While the buggy output in Figure 2.2 clearly differs from the correct output in Figure 2.1, it can be unclear what has gone wrong—or, when examining the buggy output alone, that anything has gone wrong at all. Accordingly, writing assertions or unit tests to catch this kind of bug

can be challenging: specifying the behavior of a graphics program requires formalizing how the resulting scene should be perceived. Viewers can perceive many possible outputs as visually indistinguishable, so even an informal specification of what makes a renderer "correct," for documentation or testing, can be difficult to write.

Geometry bugs in the wild. Even among established graphics libraries, geometry bugs can remain latent until a seemingly correct API change reveals the bug. This bug lay dormant while, to quote one of the maintainers, "there was a change in the rendering method that amplified the problems caused by this." The maintainer then noted that they needed to "go backfill this fix to all the docs/examples that have the broken version." Because their effects are hard to detect, geometry bugs can persist and cause subtle inaccuracies that grow as code evolves.

We found similar issues that arise when APIs fail to specify information about vector spaces. The root cause in both cases is that the programming language affords no opportunity to convey vector space information.

This paper advocates for making geometric spaces manifest in programs themselves via a type system. Language support for geometric spaces can remove ambiguity and provide self-documenting interfaces between parts of a program. Static type checking can automatically enforce preconditions on geometric operations that would otherwise be left unchecked.

2.1.2 Geometry Types

We introduce a type system that can eliminate this class of bugs, and we describe a mechanism for automatic transformation that can rule out some of them by construc-

tion. *Geometry types* describe the coordinate system representing each value and the transformations that manipulate them. A geometry type encodes three components: the *reference frame*, such as model, world, or view space; the *geometric object*, such as a point or a direction; and the *coordinate scheme*, such as Cartesian or spherical coordinates. Together, these components define which geometric operations are legal and how to implement them.

The core contribution of this paper is that all three components of geometry types are necessary. The three aspects interact in subtle ways, and real-world graphics rendering code varies in each component. Simpler systems that only use a single label [12] cannot express the full flexibility of realistic rendering code and cannot cleanly support automatic transformations. We show how encoding geometry types in a real system can help avoid and eliminate realistic geometry bugs. We will explore further how these components are defined and interact to provide operation information in Section 2.3.

We design a language, Gator, that builds on geometry types to rule out coordinate system bugs and to automatically generate correct transformation code. In Gator, programmers can write `teapotVertex in world` to obtain a representation of the `teapotVertex` vector in the `world` reference frame. The end result is a higher-level programming model that lets programmers focus on the geometric semantics of their programs without sacrificing efficiency.

We implement Gator as an overlay on GLSL [9], a popular language for implementing shaders in real-time graphics pipelines. Most GLSL programs are also valid in Gator, so programmers can easily port existing code and incrementally add typing annotations to improve its safety. We formalize a geometry type system and show that erasing these types preserves soundness. In our evaluation, we port rendering programs from GLSL to qualitatively explore Gator's expressiveness and its ability to rule out geometry bugs. We

also quantitatively compare the applications to standard GLSL implementations and find that Gator's automatic generation of transformation code does not yield meaningfully slower rendering time than hand-tuned (and unsafe) GLSL code.

This paper's contributions are:

- We identify a class of geometry bugs that exist in geometry-heavy, linear-algebra-centric code such as physical simulations and graphics renderers.
- We design a type system to describe latent coordinate systems present in linear algebra computations and prevents geometry bugs.
- We introduce a language construct that builds on the type system to automatically generate transformation code that is type correct by construction.
- We implement the type system and automatic transformation feature in Gator, an overlay on the GLSL language that powers all OpenGL-based 3D rendering.
- We experiment with case studies in the form of real graphics rendering code to show how Gator can express common patterns and prevent bugs with minimal performance overhead.

We begin with some background via a running example before describing Gator in detail.

2.2 Running Example: Diffuse Shading

This section introduces the concept of geometry bugs via an example: we implement *diffuse lighting*, a component of the classic Phong lighting model [13].[1] We assume some basic linear algebra concepts but no background in graphics or rendering.

[1] Appendix A gives a complete GLSL implementation of the Phong model.

2.2.1 Gentle Introduction to Shader Programming

Shader programs are code, typically written in C-like languages such as GLSL or HLSL, that runs on the GPU to render a graphics *scene*. The GPU executes a pipeline of shader programs, where each shader is specialized to transform a certain property of a graphical object. The shader pipeline consists of several stages. The most notable of these stages are the vertex shader, which outputs the position of each vertex as a pixel and the fragment shader, which outputs the color of each *fragment* corresponding to an on-screen pixel.

In graphics, the *scene* is a collection of objects. The shape of an object is determined by mesh data consisting of *position vectors* for each vertex, denoting the spatial structure of the object, and *normal vectors*, denoting the surface orientation at each vertex.

The kind of transformation each graphics shader applies to a graphical object depends on the pipeline stage. We focus on the vertex and fragment shader, the most common user-programmable stages of the graphics pipeline.

2.2.2 Diffuse Lighting

Diffuse lighting is a basic lighting model that simulates the local illumination on the surface of an object. Given a point on an object, the intensity of its diffuse component is proportional to the angle between the position of the light ray and the local surface normal. The diffuse model first computes the direction of the light by subtracting the mesh (surface) position, *fragPos*, from the light position:

$$lightDir = \mathsf{normalize}(lightPos - fragPos)$$

We normalize the vector, which preserves the angle but sets the magnitude to 1. We calculate the resulting diffuse intensity at this fragment as the angle between the incoming

light ray and the fragment normal using the vector dot product (which is algebraically the sum of the product of vector components):

$$\mathit{diffuse} = \max(\mathit{lightDir} \cdot \mathit{fragNorm}, 0.)$$

The max function used here prevents light from passing through the object by rejecting reflection angles greater than perpendicular.

2.2.3 Where Things Go Wrong: GLSL Implementation

To implement the diffuse lighting model, we must write a GLSL shader program that operates on a per-fragment basis. This section shows how this seemingly simple program translates to surprisingly complex code. We identify pitfalls in this implementation process that our type system will address.

GLSL has vector and matrix types, with names like `vec3` and `mat4`, along with built-in vector functions that make an initial implementation of the diffuse component seem straightforward:

```
float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
  vec3 lightDir = normalize(lightPos - fragPos);
  return max(dot(lightDir, normalize(fragNorm)), 0.);
}
```

Although `lightPos` and `fragPos` have the same type, they are not geometrically compatible: real renderers need to represent them with different reference frames and coordinate schemes. While this incorrect code directly reflects the mathematical description above, the output is nonetheless incorrect: it produces the buggy output in Figure 2.2.

Coordinate Systems The underlying problem is that software needs to represent different vectors in different coordinate systems. Information needed to render the shape of a single graphical object, the positions and normal vectors, lies in the object's *model space*, as can be seen in Figure 2.4. A model space represents the coordinates local to a single object in the scene. The origin of this space is centered in the model, with basis vectors matching the model orientation and scale. Both may change dynamically as time passes in the scene; however, each is fixed during a single iteration of the shader. *World space* gives the absolute coordinates for the entire scene, so the basis vectors and origin of world space are typically fixed.

Mesh data is scene independent, so we represent mesh parameters such as `fragPos` and `fragNorm` initially in model space, independent of the object's current relative position within the scene. In contrast, we represent the position of a light source relative to the entire scene—so `lightPos` is in world space. As a result, the subtraction expression `lightPos - fragPos` attempts to compare vectors represented in different spaces, yielding a geometrically meaningless result. This bug produces the incorrect output seen in Figure 2.2.

Transformation Matrices To fix this program, the shader needs to *transform* the two vectors to a common coordinate system before subtracting them. Mathematically, coordinate systems define an affine space, and thus geometric transformations on coordinate systems can be linear or affine. Affine transformations can change the origin and basis vectors, which can represent translation, while linear transformations affect only the basis vectors, which can represent rotation and scale.

These geometric transformations are represented in code as *transformation matrices*. To apply a transformation to a vector, shader code uses matrix-vector multiplication.

For example, the shader application may provide a matrix `uModel` that defines the transformation from model to world space using matrix multiplication:

```
vec3 lightDir = normalize(lightPos - uModel * fragPos));
```

Homogeneous Coordinates Unfortunately, this matrix multiplication implementation introduces another bug. Transforming `fragPos` from model to world space requires both a linear scaling and rotation transformation and a translation to account for change of origins. This linear transformations with translation is represented by an *affine transformation matrix*. This is a problem: an affine transformation matrix for 3D vectors must be represented as a 4×4 matrix. To multiply this matrix by `fragPos` (which is a 3-dimensional vector), we need a sensible representation of `fragPos` as a 4-dimensional vector. It is thus not immediately clear by what vector we need to multiply:

```
vec3 lightDir = normalize(lightPos - vec3(uModel *?));
```

Because a 3×3 Cartesian transformation matrix on 3-dimensional vectors can only express linear transformations, graphics software typically uses a second kind of coordinate system called *homogeneous coordinates*. An n-dimensional vector in homogeneous coordinates uses $n + 1$ values: the underlying Cartesian coordinates and a *scaling factor*, w. A 4×4 transformation matrix in homogeneous coordinates can express *affine* transformations on the underlying 3-dimensional space, including translation.

To convert from Cartesian to homogeneous coordinates, a vector $[x, y, z]$ becomes $[x, y, z, 1.]$; in the opposite direction, the homogeneous vector $[x, y, z, w]$ becomes $[x/w, y/w, z/w]$. To fix our example to use the 4-dimensional affine transformation `uModel`, we can extend `fragPos` into a homogeneous `vec4` value:

```
vec3 lightDir = normalize(
  lightPos - vec3(uModel * vec4(fragPos, 1.))
);
```

The GLSL functions `vec4` and `vec3` extend a 3-dimensional vector with the given

component and truncate a 4-dimensional vector, respectively. We now have a `lightDir` in a consistent coordinate system, namely in the world space.

The final calculation of the diffuse intensity uses this expression:

```
max(dot(lightDir, normalize(fragNorm)), 0.)
```

Here, `fragNorm` resides in model space and should be transformed into world space. One tricky detail, however, is that `fragNorm` denotes a *direction*, as opposed to a *position* as in `fragPos`. These require different geometric representations, because a direction should not be affected by translation. Fortunately, there is a trick to avoid this issue while still permitting the use of our nice homogeneous coordinate representation. By extending `fragNorm` with $w = 0$, affine translation is not applied.

```
return max(dot(lightDir, normalize(
  vec3(uModel * vec4(fragNorm, 0.))
));
```

This subtle difference is a common source of errors, particularly for novice programmers. Finally, we have a correct GLSL implementation of `diffuse`. This version results in the correct output in Figure 2.1.

2.3 Geometry Types

The problems in the previous section arise from the gap between the abstract math and the concrete implementation in code. We classify this kind of bug, when code performs geometrically meaningless operations, as a *geometry error*. Gator provides a framework for declaring a type system that can define and catch geometry errors in programs.

The core concept in Gator is the introduction of *geometry types*. These types refine simple GLSL-like vector data types, such as `vec3` and `mat4`, with information about the

geometric object they represent. A geometry type consists of three components:

- The *reference frame* defines the position and orientation of the coordinate system. A reference frame is determined by its basis vectors and origin. Examples of reference frames are model, world, and projective space.
- The *coordinate scheme* describes a coordinate system by providing operation and object definitions, such as homogeneous and Cartesian coordinates. Coordinate schemes expresses how to represent an abstract value computationally, which identifies what the underlying GLSL-like type is.
- The *geometric object* describes which geometric construct the data represents, such as a point, vector, or transformation.

In Gator, the syntax for a geometry type is `scheme<frame>.object` This notation invokes both module members and parametric polymorphism. Coordinate schemes are parameterized by a reference frame, while geometric objects are member types of a parameterized scheme. For example, `cart3<world>.point` is the type of a point lying in world space represented in a 3D Cartesian coordinate scheme.

The three geometry type components suffice to rule out the errors described in Section 2.2. The rest of the section details each component.

2.3.1 Reference Frames

We can enhance the mathematical diffuse light computation above using geometry types:

```
float diffuseNaive(
  cart3<world>.point lightPos,
  cart3<model>.point fragPos,
  cart3<model>.direction fragNorm) {
    cart3<world>.direction lightDir =
```

```
      normalize(lightPos - fragPos);
    return max(dot(lightDir, normalize(fragNorm)), 0.0);
}
```

With these stronger types, the expression `lightPos - fragPos` in this function is an error, since `lightPos` and `fragPos` are in different frames. It is geometrically legal to subtract two positions to produce a vector; the only issue with this code is the difference of reference frames. We will further discuss how Gator determines subtraction is legal in Section 2.3.2.

Definition Reference frames in Gator are labels with an integer dimension. The dimension of a frame specifies the number of linearly independent basis vectors which make up the frame. Gator does not require explicit basis vectors for constructing frames; keeping basis vectors implicit helps minimize programmer requirements and helps avoid cluttering definitions with information we don't really need. We will discuss what keeps these basis vectors are implicit through transformations between reference frames in Section 2.4.

The Gator syntax to declare the three-dimensional model and world frames is:

```
frame model has dimension 3;
frame world has dimension 3;
```

2.3.2 Coordinate Schemes

To transform `fragPos` and `fragNormal` to the `world` reference frame, we need to provide an affine transformation matrix `uModel`.

```
float diffuse(
  cart3<world>.point lightPos,
  cart3<model>.point fragPos,
  cart3<model>.direction fragNorm,
    hom3<model>.transformation<world> uModel) {
```

```
  cart3<world>.direction lightDir =
    normalize(lightPos - (uModel * fragPos));
  return max(dot(lightDir,
    normalize(uModel * fragNorm)), 0.0);
```

For this example, we define matrix–vector multiplication $m \star v$ to update types akin to function application: it ensures that m is a transformation in the same frame as the vector and parameterized on the destination frame f, then produces an output direction in the frame f. With this definition, multiplying `uModel` by an object in the `model` reference frame will result in an object in the `world` frame.

Unfortunately, multiplying `uModel * fragPos` produces a Gator type error since `uModel` and `fragPos` are in different coordinate schemes. We will resolve this issue in the next subsection by converting between schemes.

Definition Coordinate schemes provide definitions of geometric objects and operations. Concretely, they consist of operation type declarations and concrete definitions for member objects and operations. Geometric operations defined in coordinate schemes are expected to provide geometrically correct code, and are generally intended (though not required) to operate between objects within the coordinate scheme. Recall that, instead of "baking in" a particular notion of geometry, Gator lets coordinate schemes provide types that define correctness for a given set of geometric operations.

```
with frame(3) r:
coordinate cart3 : geometry {
  object vector is float[3];
  ...
}
```

For example, we can define 3D vector addition in Cartesian coordinates, which consists of adding the components of two vectors together.

```
vector +(vector v1, vector v2) {
   return [v1[0] + v2[0], v1[1] + v2[1], v1[2] + v2[2]];
}
```

All coordinate schemes are required to be parameterized with reference frames, so `cart3<model>` and `cart3<world>` are different instantiations of the same scheme. Gator's `with` syntax provides parametric polymorphism in the usual sense; in this example, the 3-dimensional Cartesian coordinate scheme is polymorphic over all 3-dimensional reference frames.

2.3.3 Geometric Objects

To apply the `uModel` affine transformation to our position and normal, we first need to convert each to homogeneous coordinates. Recall from Section 2.2.3, however, that this coordinate system transformation *differs for points and directions*. To capture this distinction, we introduce the overloaded function `homify`:[2]

```
hom<model>.point homify(cart3<model>.point p) {
  return [p[0], p[1], p[2], 1.];
}
hom<model>.direction homify(cart3<model>.direction p) {
  return [p[0], p[1], p[2], 0.];
}
```

Unlike Cartesian coordinates, homogeneous coordinates have different representations for points and directions: the latter must have zero for its last coordinate, w.

To send `fragPos` and `fragNorm` to homogeneous coordinates, it suffices to call `homify` and let the Gator compiler select the correct overloaded variant:

```
homify(fragPos);  // Extends fragPos with w=1.
homify(fragNorm); // Extends fragNorm with w=0.
```

We repeat this process to define the function `reduce`, which maps homogeneous to Cartesian coordinates. Finally, we apply these functions to our model:

```
float diffuse(
```

[2]For simplicity, this example `homify` is written only for objects in the `model` frame. Gator supports function parameterization on reference frames, so we would normally write `homify` to work on any frame.

```
  cart3<world>.point lightPos,
  cart3<model>.point fragPos,
  cart3<model>.direction fragNorm,
    hom3<model>}.transformation<world> uModel) {
    cart3<world>.direction lightDir = normalize(lightPos -
      reduce(uModel * homify(fragPos)));
  return max(dot(lightDir,
    normalize(reduce(uModel * homify(fragNorm)))
    0.0));
}
```

Now, by using all three components of the geometry type, our code will compile and produce the correct Phong diffuse color shown in Figure 2.1.

Definition The object component of a geometry type describes the type's underlying datatype and provides information on permitted operations. Object type definitions can be parameterized on reference frames, such as writing affine transformations *to* a specific frame. For example, we can define some objects in homogeneous coordinates:

```
coordinate hom3 : geometry {
  object point is float[4];
  object direction is float[4];
  with frame(3) r:
  object transformation is float[4][4];
  ...
}
```

Object and type declarations in Gator extend existing types; for example, here `point` is defined as a subtype of `float[4]`. When an operation is applied to one or more objects, Gator requires that they have matching coordinate schemes and that the function being applied has a definition in this matching scheme. For example, by omitting a definition for addition between `point`s and their supertypes, we ensure that Gator will reject `fragPos + fragPos`.

2.4 Automatic Transformations

Gator's type system statically rules out bad coordinate system transformation code. In this section, we show how it can also help automatically generate transformation code that is correct by construction. The idea is to raise the level of abstraction for coordinate system transformations so programmers do not write concrete matrix–vector multiplication computations—instead, they declaratively express source and destination spaces and let the compiler find the right transformations. A declarative approach can obviate complex transformation code that obscures the underlying computation and can quickly become out of date, such as this shift from `model` to `world` space:

```
cartesian<world>.direction worldNorm =
  normalize(lightPos - reduce(uModel * homify(fragNorm)));
```

We extend Gator with an `in` expression that generates equivalent code automatically:

```
cartesian<world>.direction worldNorm =
  normalize(lightPos - fragNorm in world);
```

The new expression converts a vector into a given representation by generating the appropriate function calls and matrix–vector multiplication. Specifically, the expression *e* `in scheme<frame>` takes a typed vector expression *e* from its current geometry type `T.object` to the type `scheme<frame>.object` by finding a series of transformations that can be applied to *e*. With this notation, either the `scheme` or `frame` can be omitted without ambiguity, so writing `x in world` where `x` is in scheme `cart3` is the same as writing `x in cart3<world>`. Gator `in` expressions can only be used to change the coordinate scheme or parameterizing reference frame; that is, the geometric object of the target type must be the same as the original value type.

Implementation The Gator compiler implements `in` expressions by searching for transformations to complete the chain from one type to another. It uses a *transformation*

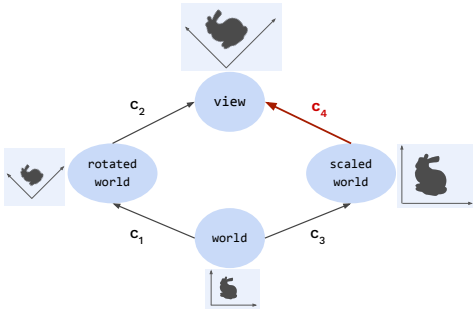

Figure 2.5: A *transformation graph* with provided transformations. The highlighted edge represents a newly added transformation function, which must be unique and agree with the existing paths on the graph.

$$
\begin{aligned}
c &\in \text{constants} \\
x &\in \text{variables} \\
f &\in \text{function names} \\
p &\in \text{primitives} \\
t &\in \text{types} \\
\tau &::= \text{unit} \mid \top_p \mid \bot_p \mid t \\
e &::= v \mid c \mid f(e_1, e_2) \mid x \text{ as! } \tau \mid x \text{ in } \tau \\
C &::= \tau\, x = e \mid e \\
P &= C; P \mid \epsilon
\end{aligned}
$$

Figure 2.6: Core Gator syntax.

graph where the vertices are types and the edges are transformation matrices or functions. Figure 2.5 gives a visual representation of a transformation graph.

2.4.1 Canonical Functions

The transformations that gator reasons about for automatic application are special: they must uniquely define a map from their domain to their range. Gator requires these functions to be labeled with the word `canon`. Gator defines three requirements on these transformations: (1) there can be only one canonical function between each pair of types in a given scope, (2) all canonical functions between reference frames must map between frames of the same dimension, and (3) a canonical function can only have one non-canonical argument.

To expand on condition (3); canonical functions may take in *canonical arguments*, which are variables labelled with the `canon` keyword. The most familiar example of this use is defining matrix—vector multiplication to be canonical; the matrix itself must be included and must be a canonical matrix:

```
with frame(3) target:
canon point *(canon transformation<target> t, point x) {
...
}
...
// Now declare the matrix as canonical
// for use with multiplication
canon hom<model>.transformation<world> uModel;
homPos in world; // --> uModel * homPos
```

It is legal to manually fill canonical arguments to functions with non-canonical variables; however, `in` expressions will never do so.

The intuition of canonical functions comes from affine transformations between frames and coordinate schemes. Since each frame has underlying basis vectors, transformations between frames of the same dimension which preserve these frames are necessarily unique; further, applying these bijective transformations does not cause data to "lose information." Similarly, coordinate schemes simply provide different ways to view the same information; there are often unique transformations between schemes that can be applied as needed to unify data representation.

This construction of canonical functions and automatic transformations is similar to constructions provided by C# and C++'s type coercion. The slightly different approach needed for `in` expressions will be discussed briefly in Section 2.8.

2.4.2 Correctness of Generated Transformations

With `in` expressions, Gator programmers sacrifice control for convenience: the compiler picks which transformation functions and matrices to use to get from one coordinate system to another. If all the individual transformations marked with `canon` are correct, then the composed "chain" generated for an `in` expression must also be correct. Functional verification of transformations, however, is not feasible in Gator's purely static setting: it would require not only the value of every transformation matrix, which typically varies dynamically over time, but also an intrinsic description of each coordinate system, such as the basis vectors for every reference frame, which is never available in real graphics code. We view heavyweight dynamic debugging aids for checking transformation correctness as important future work.

We can, however, state a simple consistency condition that is necessary but not sufficient for a system of canonical transformations to be correct. The transformation system should be *path independent:* for any two types τ_1 and τ_2, the behavior of any chain of transformations from τ_1 to τ_2 should be equivalent. In other words, every edge in the transformation graph corresponds to a function—so every path corresponds to a function composition, and every such path between the same two vertices should yield the same composed function. (This definition is equivalent to commutativity for diagrams [**?**].) Otherwise, the semantics of an expression e `in` τ would depend on the graph search algorithm that Gator uses to find routes in the transformation graph, which is clearly undesirable.

Because it is a purely static system, Gator does not enforce path independence. However, path independence motivates Gator's requirement that canonical transformations preserve dimensionality (see Section 2.4.1). Without this condition, we have found it is easy to accidentally violate path independence with non-invertible functions and result in

an ambiguous transformation graph for `in` expressions.

2.5 Formal Semantics

Gator provides a framework for defining geometry types as an "overlay" on top of computation-oriented programs in a base language without geometry types. In this section, we formalize a core of Gator to show that its constructs are sound with respect to such an underlying language. The goal is a theorem stating that well-typed Gator programs, when translated, result in well-typed programs in the target language. We focus on the generic, extensible Gator language rather than formalizing the rules for any specific geometric system—affine transformations on Cartesian coordinates, for example. Proving soundness with respect to a linear algebra domain would be interesting future work but is out of scope for this paper.

We define two languages: a high-level core semantics for Gator that includes its user-defined types, and a low-level abstract target language, Hatchling. Hatchling represents a sound imperative language with some set of primitive types and operators on those types. For example, an instance with fixed-size vector and matrix types can reflect a simple core of GLSL.

2.5.1 Syntax

Figure 2.6 lists the syntax of the formal core of Gator that we formalize in this section. The types in this core language consist of `unit` and a lattice over each primitive type p. The choice of primitives is kept abstract in this formalism to highlight that the Gator extend over arbitrary underlying datatypes. For example, in a GLSL core language, we

might have a primitive `float` or `vec3` – something like `vector` would be a custom type t and not a primitive.

A program in Gator is a series of commands; we simplify these to variable declaration, assignment, and expressions. Gator expressions are constructed around function applications, with `as` and `in` expressions to help manage types.. We assume functions always take two arguments for simplicity; extending this assumption for other argument counts is straightforward.

2.5.2 Typing Rules

We define a typing judgment for Gator programs, $\Gamma \vdash P : \tau$, that, for any program P and typing context Γ, produces a type τ. The complete semantics for this judgment can be seen in Figure 2.7. Note that Γ is kept constant throughout; declaring a variable requires looking up into the constant Γ to determine if the declared type matches the expected type. Keeping Γ constant will later help with translation; the type of any expression can be determined exactly from the constant global contexts Γ, X, and Φ along with the judgment P.

Gator requires a lattice for each primitive type; custom types on each lattice introduces new subtyping relations. We define a type ordering among types $\leq$ where $t_1 \leq t_2$ means that t_1 is a subtype of t_2. $\leq$ is expected to be reflexive and transitive. In a well formed program, $\leq$ must contain a rule for every user defined type, every type (except unit) must be a subtype of a primitive top type, and every bottom type $\bot_p$ must be a subtype of each subtype of the associated $\top_p$. In other words, $\leq$ must conform to a lattice structure for each primitive p. The complete summary of subtyping rules can be found in the attached supplementary materials.

$$\frac{\tau_1 \leq \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \qquad \frac{\mathrm{X}(c) = p}{\Gamma \vdash c : \bot_p} \qquad \frac{\Gamma(v) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = \tau}{\Gamma \vdash \tau\ x = e : \text{unit}}$$

$$\frac{\Gamma \vdash C : \tau_1 \qquad \Gamma \vdash P : \tau_2}{\Gamma \vdash C; P : \text{unit}} \qquad \frac{}{\Gamma \vdash \epsilon : \text{unit}} \qquad \frac{\Gamma \vdash e : \top_p \qquad \tau \leq \top_p}{\Gamma \vdash e \text{ as! } \tau : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \mathrm{P}(\tau_1, \tau_2) = f}{\Gamma \vdash e \text{ in } \tau_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Phi(f, \tau_1, \tau_2) = \tau_3}{\Gamma \vdash f(e_1, e_2) : \tau_3}$$

Figure 2.7: Typing Judgment

The typing information for functions is stored in a function typing context, Φ, which maps the tuple of function name and input types to the output type. The semantics of Φ are built to support overloaded functions.

Gator, as defined in these semantics, is parameterized over primitive types stored in primitive type context X, which maps a literal to its primitive type.

The map from in expressions to paths is managed by the judgment P. More precisely, P maps a given start and end type τ_1 and τ_2 to a function name that, when applied to an expression of type τ_1, produces an expression of type τ_2. We simplify the judgment of P here to only allow one step for notation clarity; in the real Gator implementation, the transformation may be a chain of functions. The details of this judgment P are omitted for simplicity, but amount to a simple lookup through the available functions for a function of the correct type.

2.5.3 Translation Soundness

To prove the translation soundness of Gator, we need to first define Hatchling and our translation from Gator to Hatchling. We will show that a well-typed Gator program must

$$
\begin{aligned}
[\![c]\!]_\Gamma &\triangleq c & [\![x]\!]_\Gamma &\triangleq x \\
[\![\tau\ x := e]\!]_\Gamma &\triangleq [\![\tau]\!]\ x := [\![e]\!]_\Gamma & [\![e \text{ as! } \tau]\!]_\Gamma &\triangleq [\![e]\!]_\Gamma \\
[\![e \text{ in } \tau_2]\!]_\Gamma &\triangleq [\![f(e)]\!]_\Gamma & &\text{where } \Gamma \vdash e : \tau_1 \text{ and } f = \mathrm{P}(e, \tau_1, \tau_2) \\
[\![f(e_1, e_2)]\!]_\Gamma &\triangleq f'(e_1, e_2) & &\text{where } \Gamma \vdash e : \tau_1, \Gamma \vdash e : \tau_2, \text{ and } f' = \Psi(f, e_1, e_2, \tau_1, \tau_2) \\
[\![\epsilon]\!]_\Gamma &\triangleq \epsilon & [\![C; P]\!]_\Gamma &\triangleq [\![C]\!]_\Gamma; [\![P]\!]_\Gamma \\
[\![t]\!] &\triangleq \top_p & &\text{where } t \leq \top_p \\
[\![\top_p]\!] &\triangleq \top_p & [\![\bot_p]\!] &\triangleq \top_p \\
[\![\text{unit}]\!] &\triangleq \text{unit}
\end{aligned}
$$

Figure 2.8: Translational semantics for expressions and types

translate to a well-typed Hatchling program.

Primitives in Gator can be translated to a type in the target language. For notation convenience we name primitives such that $\top_p$ in Gator translates to $\top_p$ in Hatchling.

We define the syntax of Hatchling to be identical to Gator syntax except for τ, which is instead written as $\tau ::= \mathsf{unit} \mid \top_p$, and without as! or in expressions. In other words, Hatchling is simply Gator with custom type labels and associated operations erased. In the formalism of Hatchling, abstraction over operation implementation is done using operation context Ξ that maps an operator name to its output type.

When Hatchling is parameterized to be a simple core of GLSL, some top types we might see are the `float` and `vec3` types. Translation from Gator would consist of erasing custom geometry types, such as `cart3<model>.point`, to their associated top type; in this case `vec3`.

To translate Gator's externally-defined functions (which may be overloaded on types not part of Hatchling), we invoke the context Ψ. Ψ maps the tuple of a function name, input expressions, and input types to an expression in the target language. For example, we might map Gator's definition of subtraction between points to be a GLSL subtraction between two `vec3`s. The resulting function names must each be unique and preserve

the translation of Gator's primitive types. A well formed function translation context Ψ would necessarily map functions to expressions of the correct return type, as constrained by Φ under translation.

We reuse the judgment P as a mechanism to resolve in expressions, applying the function result of evaluating the judgment to e. This must produce a result of the correct translated type for a well-formed judgment P.

The typing rule for operation expressions is:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Xi(o) = \tau_3}{\Gamma \vdash o(e_1, e_2) : \tau_3}$$

We emphasize this rule as being similar to Gator's rules for operations, but with a "translated" context using only Hatchling (i.e. primitive) types. We also note that Ξ does not take in types as arguments, thus Hatchling does not support overloaded functions.

We define translational semantics from type-annotated Gator to Hatchling in Figure 2.8. The typing contexts Γ and Φ are translated by replacing every τ in their range with $[\![\tau]\!]$.

Using structural induction over expressions in Gator, we are now able to show that

Theorem 1 (translational soundness)**.** *For all* Γ*,* e*, and* τ*, if* $\Gamma \vdash e : \tau$ *then* $[\![\Gamma]\!] \vdash [\![e]\!]_\Gamma : [\![\tau]\!]$.

That is to say, if Gator code type checks, then Hatchling code type checks. Since Hatchling is constrained to be sound, Gator must be sound. A sketch of this proof is included in the provided supplementary materials.

2.6 Implementation

We implemented Gator in a compiler that statically checks user-defined geometric type systems as described in Section 2.3 and automatically generates transformation code as described in Section 2.4. The compiler consists of 2,800 lines of OCaml. It can emit either GLSL or TypeScript source code, to target either GPU shaders or CPU-side setup code, respectively.

The rest of this section describes how the full Gator language implementation extends the core language features to enable real-world graphics programming. We demonstrate these features in detail in a series of case studies in Section 2.7.

2.6.1 Practical Features

Types While Gator is designed around geometry types, writing realistic code requires a more complete language design. Aside from the primitive types `bool`, `int`, `float`, and `string`, Gator supports fixed-length array types, such as `float[3]`, and type aliases.

New types may be declared as a *subtype* of an existing type. For instance, we can add support for the GLSL-style `vec3`:

```
type vec3 is float[3];
```

Through creating a custom type alias, we can, for example, provide support for a subtype of `float[3]`, the GLSL `vec3`. While the built-in `float[3]` type does not support vector addition, we will be able to write $x + y$ for `vec3`s x, y as in GLSL.

To allow literal values to interact intuitively with custom types, literals in Gator have special types. For example, the number 42 is of type `%int`. Gator introduces a typing

rule where each literal type $\%p$ is a subtype of every subtype of p. In other words, the literal type $\%p$ is the bottom type for the type hierarchy with top type p. We summarize these ideas in this example:

```
type vec3 is float[3];
vec3 s1 = [4.2, 4.2, 4.2];  // Legal
float[3] x = s1;            // Legal
vec3 s2 = x;                // ERROR: float[3] is not a vec3
```

This behavior of literal values allows us to capture the Gator-style intuition that a given vector can either be a geometric point or just a raw GLSL `vec3`, but this information is not known until the data is assigned to a variable.

Type Inference Gator supports local type inference using the `auto` keyword:

```
cart3<model>.point fragPos = ...;
// worldPos will have type cart3<world>.point
auto worldPos = fragPos in world;
```

External Functions Functions and variables defined externally in the Gator target can be written using the `declare` keyword.

```
declare vec3 normalize(vec3 v);
```

All arithmetic operations in Gator are functions which can be declared and overloaded. Gator has no built-in functions. Requiring this declaration allows us to include GLSL-style infix addition of vectors without violating coordinate systems restrictions:

```
declare vec3 +(vec3 v1, vec3 v2);
```

Addition is then valid for values of type `vec3`:

```
vec3 x = [0., 1., 2.];
vec3 result = x + x; // Legal
```

But emits an error when applied to two points, as desired, since they are not subtypes of vec3 and so there is no valid function overload:

```
cartesian<model>.point fragPos = [0., 1., 2.];
```

```
// ERROR: No addition defined for points
auto result = fragPos + fragPos;
```

Import System To support using custom Gator libraries in a readable way, we built a simple import system in Gator. Files can be imported with the keyword `using` followed by the name of the file:

```
using "../glsl_defs.lgl";
```

Unsafe Casting As an escape hatch from strict vector typing, Gator provides an unsound cast expression written with `as!`:

```
vec3 position = fragPos as! vec3;
```

Casts must preserve the primitive representation; we could not, for instance, cast a variable with type `float[2]` to `float[3]`. Unsafe casts syntactically resemble `in` expressions but are unsound and carry no run-time cost. These casts both allow for unsafe transformations for defining a function that is externally "known" to be safe, and for allowing the user to forgo Gator's type system and work directly with GLSL-like semantics, as seen in the example above.

2.6.2 Standard Library

Per Section 2.5, Gator does not include any built-in functions or operations. Our implementation does provide array indexing as a built-in function to help simplify definitions, but otherwise matches requires that operations such as `+` be explicitly declared.

We implement a standard library provides access to common GLSL operations. This

library consists of GLSL function declarations, scheme declarations for Cartesian and Homogeneous coordinates, and basic transformation functions such as `homify` and `reduce`. Relevant GLSL functions are declared to work on GLSL types, such as the addition operation operation in section 2.6:

```
declare vec3 +(vec3 x, vec3 y);
```

We build schemes in much the same way as introduced in Section 2.3.2, as with the sketch of the `cart3` scheme:

```
with frame(3) r:
coordinate cart3 : geometry {
  object vector is float[3];
  vector +(vector v1, vector v2) {
        return [
          v1[0] + v2[0],
          v1[1] + v2[1],
          v1[2] + v2[2]];
  }
}
```

Finally, we include `homify` and `reduce` transform between homogeneous and cartesian coordinates as discussed in Section 2.3.3:

```
hom<model>.point homify(cart3<model>.point p) {
  return [p[0], p[1], p[2], 1.];
}
cart3<model>.point reduce(hom<model>.point p) {
  return [p[0], p[1], p[2]];
}
```

We use this same library when implementing each shader for the case study.

2.7 Gator in Practice

This section explores how Gator can help programmers avoid geometry bugs using a series of case studies. We use the Gator compiler to implement OpenGL-based renderers that demonstrate a variety of common visual effects, and we compare against

Figure 2.9: Texture.

implementations in plain GLSL. We report qualitatively on how Gator's type system influences the expression of the rendering code (Section 2.7.1 and quantitatively on the performance impact of Gator's `in` expressions (Section 2.7.2).

Figure 2.10: Reflection.

2.7.1 Case Studies

To qualitatively study Gator's safety and expressiveness, we used it to implement 8 renderers based on the OpenGL API in its browser-based incarnation, WebGL [6]. To the best of our knowledge, there is no standard benchmark suite for evaluating the expressiveness and performance of graphics shader programs. Instead, we assemble implementations of a range of common rendering effects:

- *Phong:* The lighting model introduced in Section 2.2.
- *Reflection:* Use two-pass rendering to render an object that reflects its surroundings.
- *Shadow map:* Simulate shadows for moving objects by computing a projection.
- *Microfacet:* Texture model for simulating roughness on a surface.

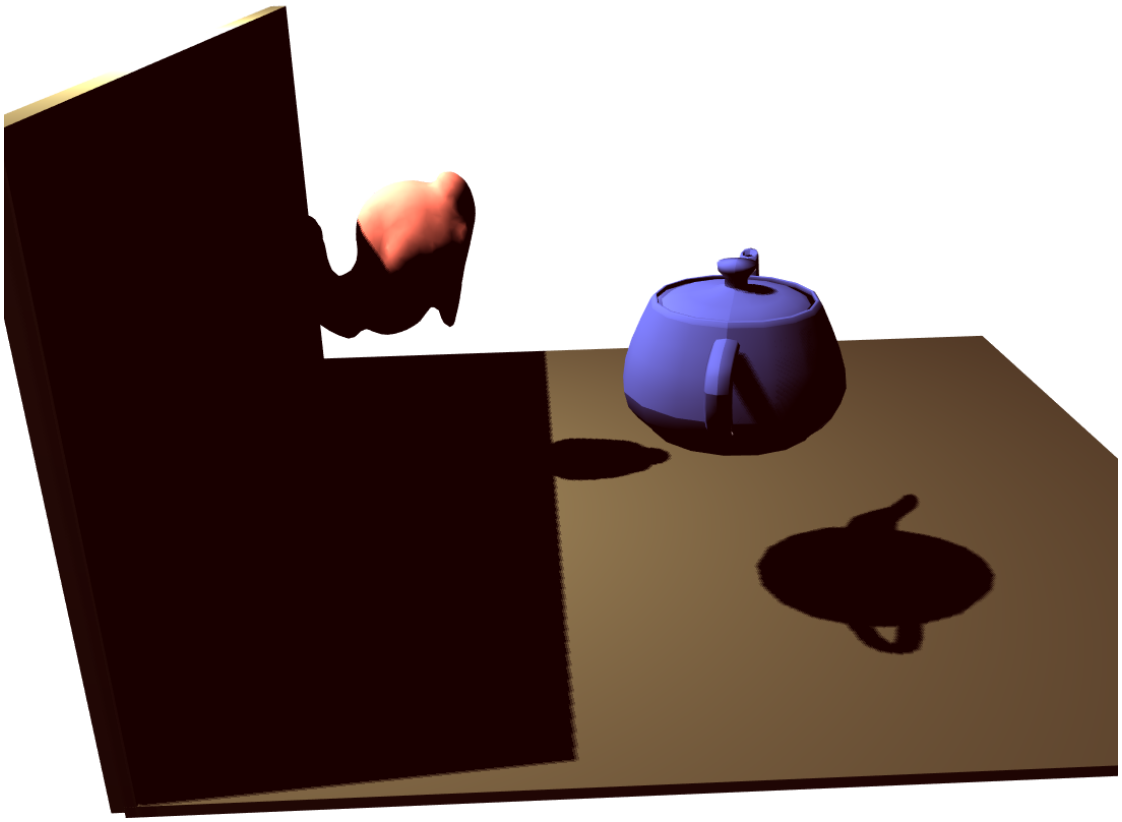

Figure 2.11: Shadow map.

- *Texture:* Use OpenGL's texture mapping facility to draw an image on the surface of an object.
- *Spotlight:* Phong lighting restricted to a spotlight circle.
- *Fog:* Lighting model with integration to simulate distortion from fog.
- *Bump map:* Texture model for simulating bumps on surfaces.

Each renderer consists of both CPU-side "host" code and several GPU-side shader programs. Figure **??** depicts the output of a selection of these renderers.

The rest of this section reports on salient findings from the case studies and compares them to standard implementations in GLSL and TypeScript. For the sake of space, we highlight the most distinct cases where Gator helped clarify geometric properties and

Figure 2.12: Microfacet.

prevent geometry bugs that would not be caught by plain GLSL. The complete code of both the Gator and reference GLSL implementations can be found online.[3]

Reflection Our reflection case study, shown in Figure 2.10, renders an object that reflects the dynamic scene around it, creating a "mirrored" appearance. The surrounding scene includes a static background texture, known as a *skybox*, and several non-reflective

[3]URL omitted for anonymous review

floating objects to demonstrate how the reflected scene changes dynamically.

Rendering a reflection effect requires several passes through the graphics pipeline. The idea is to first render the scene that the mirror-like object will reflect, and then render the scene again with that resulting image "painted" onto the surface of the object. There are three main phases: (1) Render the non-reflective objects from the perspective of the reflective object. This requires six passes, one for each direction in 3-space. (2) Render the reflection using the generated cube as a texture reference. (3) Finally, render all other objects from the perspective of the camera.

Reflection: Inverse Transformation For the second step, we refer to a cubemap—a special GLSL texture with six sides—to refer to the six directions of the scene. To calculate the angle of reflection, we need to reason about the interactions of the light rays in view space *as they map onto our model space*. Specifically, calculating the reflection amounts to the following operations, where `V` is the current vertex's position and `N` is the current normal vector, which must both be in the view frame:

```
uniform samplerCube<alphaColor> uSkybox;
...
void main() {
  ...
  cart3<view>.vector R = -reflect(V, N);
  auto gl_FragColor = textureCube(uSkybox, R in model);
}
```

The key feature to note here is the transformation `R in model`, which accomplishes our goal of returning the light calculation to the object's perspective (the model frame). This transformation requires that we map backwards through the world frame, a transformation which requires the inverse of the `model`→`world` matrix and the `world`→`view` matrix multiplied together. This interaction produces a unique feature in Gator's type system, where we need to both have a forward transformation and its inverse. The shader declares the matrices as follows, with the inversion being done preemptively on the CPU:

```
canon uniform hom<world>.
  transformation<view> uView;
canon uniform hom<model>.
  transformation<world> uModel;
canon uniform cart3<view>.
  transformation<model> uInverseViewTransform;
```

The inverse view transform uses a Cartesian (`cart3`) matrix because we intend only to use it for the vector `R`, which ignores the translation component of the affine transformation. The inverse transformation is what permits us to write `R in model`, while the forward transformations must be uniquely given to actually send our position and normal to the view frame (as noted before):

```
varying cart3<model>.point vPosition;
varying cart3<model>.normal vNormal;
void main()
auto N = normalize(vNormal in view);
auto V = -(vPosition in view);
...
}
```

Reflection: Normal Transformation Additionally, we need to reason about the correct transformation of the normal *with translation* (that is, when moving the object in space), which means that we need the inverse transpose matrix, which provides a distinct path between the model and view frames. The use of the inverse transpose of the model-view matrix is perhaps unexpected; it arises specifically for a geometry normal from a convenient algebraic result.

In GLSL, it is easy to mistakenly transform the normal as if it were an ordinary direction:

```
varying vec3 vNormal;
void main() {
  auto N = normalize(vec3(
    uView * uModel * vec4(vNormal, 0.)));
}
```

This code is wrong because `uModel * vec4(vNormal, 0.)` does not apply the

translation component of the `uModel` transformation. To prevent this kind of bug, the Gator standard library defines the `normal` type, which is a subtype of `vector`. A new `normalTransformation` type can only operate on normals. Using these types, a simple `in` transformation suffices:

```
canon uniform cart3<model>.
  normalTransformation<view> uNormalMatrix;
varying cart3<model>.normal vNormal;
void main() {
  // uNormalMatrix * vNormal
  auto N = normalize(vNormal in view);
}
```

The compiler uses the normal version of the transformation, correctly applying the translation component.

Shadow Map: Light Space Shadow mapping is a technique to simulate the shadows cast by 3D objects when illuminated by a point light source. Our case study, shown in Figure 2.11, renders several objects that cast shadows on each other and a single "floor" surface. The non-shadow coloring is simulated through Phong lighting as previously discussed.

As with the reflection renderer, to calculate shadows in a scene, we require several passes through the graphics pipeline. The first pass renders the scene from the perspective of the *light* and calculates the whether a given pixel is obscured by another. The second pass uses this information to draw shadows; a given pixel is lit only if it is not obscured from the light.

The first pass does all geometric operations in the vertex shader to render the scene from the light's perspective. This is easy to get wrong in GLSL by defaulting to the usual transformation chain:

```
void main() {
  // The usual transformation chain here is wrong!
  // We should instead be using
```

```
  //    uLightProjective and uLightView
  vec4 gl_Position = uProjective *
    uView * uModel * vec4(aPosition, 1.);
}
```

This incorrect transformation chain will lead to shadows in strange places and hard-to-debug effects.

In Gator, on the other hand, the work is done when typing the matrices themselves. From there, the transformation to light space is both documented and correct by construction:

```
attribute cart3<model>.point aPosition;
canon uniform hom<model>.
  transformation<world> uModel;
canon uniform hom<world>.
  transformation<light> uLightView;
canon uniform hom<light>.
  transformation<lightProjective> uLightProjection;

void main() {
auto gl_Position = aPosition in hom<lightProjective>;
// ...
}
```

We use the depth information in the final pass in the form of `uTexture`. To look up where the shadow should be placed, we must lookup the position of the current pixel in the light's projective space (which is where the position was represented in the previous rendering). In GLSL, we require the following hard-to-read code:

```
float texelSize = 1. / 1024.;
float texelDepth = texture2D(uTexture,
vec2(uLightProjective * uLightView *
  uModel * vec4(vPosition, 1.))) + texelSize));
```

Using the correct transformations is difficult and hard to be sure if the correct transformation chain was used once again. In Gator, on the other hand, this is straightforward:

```
float texelSize = 1. / 1024.;
float texelDepth = texture2D(uTexture,
  vec2(vPosition in lightProjective) + texelSize));
```

Microfacet: Custom Canonical Functions Anisotropic microfacet shading creates an illusion of roughness and bumpiness on a 3D modeled surface using information from the normal map of that surface. Modeling this correctly, however, requires an unusual technique: building a local reference frame from the perspective of the normal vector called the local normal frame.

Converting to the local normal frame of a given normal consists of a function call with the appropriate normal vector.

```
vec3 proj_normalframe(vec3 m, vec3 n) { ... }
vec3 geom_normal;
vec3 result = proj_normalframe(viewDir, geom_normal);
```

However, as with other conversions between spaces, writing this kind of code in GLSL can involve multiple nonobvious steps. If the normal and target direction are in different spaces, the GLSL code must look like this:

```
vec3 result = proj_normalframe(vec3(uView *
  uModel * vec4(modelDir, 1.)), geom_normal);
```

In Gator, we instead declare `proj_normalframe` with the appropriate types and a canonical tag, noting that the normal itself is a canonical part of the transformation:

```
frame normalframe has dimension 3;
canon cart3<normalframe>.direction proj_normalframe(
cart3<view>.direction m, canon cart3<view>.normal n) { ... }
```

We then declare the normal `geom_normal` with the appropriate type, and the transformation type becomes straightforward:

```
canon cart3<view>.normal geom_normal;
auto result = modelDir in normalframe;
```

Textures: Parameterized Types A *texture* is an image that a renderer maps onto the surface of a 3D object, creating the illusion that the object has a "textured" surface. Our texture case study renders a face mesh with a single texture (shown in Figure 2.9). While

this example does not provide any geometry insight, we highlight the study to show the broad utility of the types introduced by Gator for a graphics context. GLSL represents a texture using a `sampler2D` value, which acts as a pointer to the requested image, which is typically an input to a shader:

```
uniform sampler2D uTexture;
```

Textures are mapped to the image using the object's current texture coordinate:

```
varying vec2 vTexCoord;
```

Whereas textures themselves are typically constant (using the `uniform` keyword), a texture coordinate like `vTexCoord` differs for each vertex in a mesh (as the `varying` keyword indicates). To sample a color from a texture at a specific location, a fragment shader must use the GLSL `texture2D` function:

```
vec4 gl_FragColor = texture2D(uTexture, vTexCoord);
```

The result type of `texture2D` in GLSL is `vec4`: while textures typically contain colors (consisting of red, green, blue, and alpha channels), renderers can also use them to store other data such as shadow maps or even points in a coordinate system.

In Gator and its GLSL standard library, `sampler2D` is a polymorphic type that indicates the values it contains:

```
with float[4] T:
declare type sampler2D;
with float[4] T:
declare T texture(sampler2D<T> tex, vec2 uv);
```

For this renderer, the texture contains `alphaColor` values, which represent color values that can be used as `gl_FragColor`. The fragment shader is nearly identical to GLSL but with more specific types:

```
uniform sampler2D<alphaColor> uTexture;
varying vec2 vTexCoord;
void main() {
  alphaColor gl_FragColor = texture2D(uTexture, vTexCoord);
}
```

With this code, we guarantee that the texture represented by `uTexture` will produce a color which can be directly used by `gl_FragColor`. We therefore both provide documentation and prevent errors with trying to use the resulting vector as, say, a point for later calculations.

2.7.2 Performance

While Gator is chiefly an "overhead-free" wrapper that expresses the same semantics as an underlying language, there is one exception where Gator code can differ from plain GLSL: its automatic transformation insertion using `in` expressions (Section 2.4).

The Gator implementation compiles `in` expressions to a chain of transformation operations that may be slower than the equivalent in a hand-written GLSL shader. In particular, hand-written GLSL code can store and reuse transformation results or composed matrices, while the Gator compiler does not currently attempt to do so. The Gator compiler also generates function wrappers to enable its overloading. While both patterns should be amenable to cleanup by standard compiler optimizations, this section measures the performance impact by comparing Gator implementations of renderers from our case study to hand-optimized GLSL implementations.

Experimental Setup

We perform experiments on Windows 10 version 1903 with an Intel i7-8700K CPU, NVIDIA GeForce GTX 1070, 16 GB RAM, and Chrome 81.0.4044.138. We run 60 testing rounds, each of which executes the benchmarks in a randomly shuffled order. In each round of testing, we execute each program for 20 seconds while recording the time

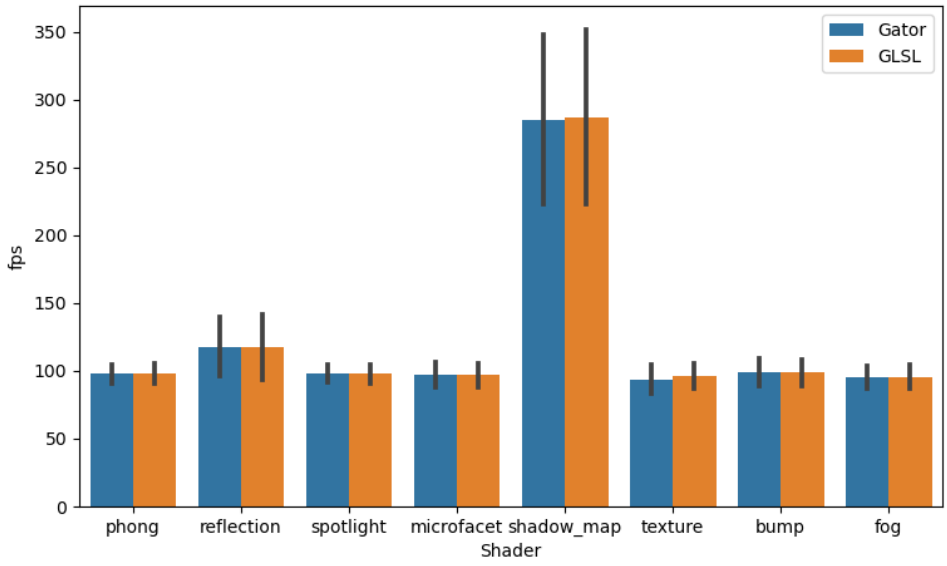

Figure 2.13: The mean frames per second (fps) for each shader for both the baseline (GLSL) and Gator code. Error bars show the standard deviation.

to render each frame. We report the mean and standard deviation of the frame rate across all rounds.

Performance Results

Figure 2.13 shows the average frames per second (fps) for the GLSL and Gator versions of each renderer, and Table 2.1 shows mean and standard deviation of each frame rate. The frame rates for the two versions are generally very similar—the means are all within one standard deviation. Several benchmarks have frame rates around 100 fps because they render the same number of objects and the bulk of the cost comes from scene setup. We used around 100 objects for all scenes except reflection and shadow to reduce natural variation and focus on measuring the cost of the shaders.

Table 2.1 shows the results of Wilcoxon signed-rank statistical tests that detect

	Gator		GLSL		p-value	
Shader	Mean	S.E.	Mean	S.E.	Wilcoxon	TOST
phong	97.84	0.22	97.67	0.21	0.187	0.003*
texture	95.82	0.27	93.75	0.31	<0.001*	0.996
reflect	117.8	0.72	117.7	0.65	0.638	0.188
shadow	287.0	1.91	285.1	1.85	0.365	0.636
bump	98.60	0.29	99.07	0.29	0.063	0.098
microfacet	96.71	0.27	96.91	0.28	0.640	0.020*
fog	95.74	0.26	95.41	0.25	0.119	0.033*
spotlight	97.83	0.21	98.07	0.20	0.299	0.005*

Table 2.1: Mean and standard error of the frame rate for the Gator and GLSL (baseline) implementation of each benchmark. We also give the p-value for a Wilcoxon sign rank test and two one-sided t-test (TOST) equivalence test that checks whether the means are within 1 fps, where * denotes statistical significance ($p < 0.05$).

differences in the mean frame rates. At an $\alpha = 0.05$ significance level, we find a statistically significant difference only for texture. However, a difference of means test cannot *confirm* that a difference does *not* exist. For that, we also use we use the two one-sided t-test (TOST) procedure [16], which yields statistical significance ($p < \alpha$) when the difference in means is within a threshold. We use a threshold of 1 fps. The test rejects the null hypothesis—concluding, with high confidence, that the means are similar—for the phong, microfacet, fog, and spotlight shaders.

The anomaly is `texture`, where our test concludes that a small (2 fps) performance difference does exist, although the differences are still within one standard deviation. Our best guess as to the reason is due to a result of the boilerplate functions inserted by Gator, some of which be optimized away with more work.

2.8 Related Work

SafeGI [12] introduces a type system as a C/C++ library for geometric objects parameterized on reference frame labels not unlike Gator's geometry types. The types introduced by SafeGI do not include information about the coordinate scheme, and so also requrie abstracting the notion of transformations to a map type which must be applied through a layer of abstraction. Additionally, SafeGI does not attempt to introduce automatic transformations like Gator's `in` expressions nor attempt to study the result of applying these types to real code.

The dominant mainstream graphics shader languages are OpenGL's GLSL [9] and Direct3D's HLSL [10]. Research on graphics-oriented languages for manipulating vectors dates at least to Hanrahan and Lawson's original *shading language* [4]. Recent research on improving these shading languages has focused on modularity and interactions between pipeline stages: Spark [1] encourages modular composition of shaders; Spire [5] facilitates rapid experimentation with implementation choices; and Braid [15] uses multi-stage programming to manage interactions between shaders. These languages do not address vector-space bugs. Gator's type system and transformation expressions are orthogonal and could apply to any of these underlying languages.

Scenic [3] introduces semantics to reason about relative object positions and λCAD [11] introduces a small functional language for writing affine transformations, although neither seem to have a type system for checking the coordinate systems they've defined. Practitioners have noticed that vector-space bugs are tricky to solve and have proposed using a naming convention to rule them out [18]. A 2017 enumeration of programming problems in graphics [14] identifies the problem with latent vector spaces and suggests that a novel type system may be a solution. Gator can be seen as a realization

of this proposal.

Gator's type system works as an overlay for a simpler, underlying type system that only enforces dimensional restrictions. This pattern resembles prior work on type qualifiers [2], dimension types [7], and type systems for tracking physical units [8]. Canonical transformations in Gator are similar in feel to Haskell's type class polymorphic functions, where Gator's `space` type can be defined as a type class and the `in` keyword behave similarly to Haskell lookup calls. Additionally, Gator's notion of automatic transformations is a specialized use type coercion, similar to structures introduces in the C# and C++ languages. What is particular about Gator's automatic type coercion is the notion of path independence discussed in Section 2.4, along with a definition of uniqueness and bijectivity of canonical transformations. Together, these requirements allow automation of coordinate system transformations that would not be allowed in other, similar systems.

2.9 Conclusion

Gator attacks a main impediment to graphics programming that makes it hard to learn and makes rendering software hard to maintain. Geometry bugs are extremely hard to catch dynamically, so Gator shows how to bake them into a type system and how a compiler can declaratively generate "correct by construction" geometric code. We see Gator as a foundation for future work that brings programming languages insights to graphics software, such as formalizing the semantics of geometric systems and providing abstractions over multi-stage GPU pipelines.

Geometry bugs are not just about graphics, however. Similar bugs arise in fields fields ranging from robotics to scientific computing. In Gator, users can write libraries to

encode domain-specific forms of geometry: affine, hyperbolic, or elliptic geometry, for example. We hope to expand Gator's standard library as we apply it to an expanding set of domains.

CHAPTER 3

ONLINE VERIFICATION OF COMMUTATIVITY

3.1 Introduction

Many systems use diagrams: graphs where nodes are domains and edges are transformation functions. A type system with coercions, for example, corresponds to a graph whose nodes are types and whose edges are coercions. Figure 3.2 illustrates an example in a simple language with units-of-measure types [**?**]. In such a system, an important correctness criterion is that the diagram *commutes*: when traversing the graph from any start node to any end node, applying every transformation along the path to any input value, the result is the same output value *independent of the path chosen between the two nodes*. With our coercion example, it is a problem if casting to a supposedly equivalent type as an intermediate step resulted in a different answer than a direct cast. Specifically, given a variable `x` of type `meters`, applying the cast `(wugs) x` can be done in two ways: either `(wugs)(feet) x` or `(wugs)(miles) x`. Which path is taken depends on the compiler; we would like the choice of paths to be semantically equivalent so the compiler is free to make a choice.

```
var x : meters = 1;
define foot:
    1 meter = 3.28 feet;
define miles:
    1 meter = 0.000621 miles;
define wugs:
    1 mile = 10000 wugs;
    1 foot = 10 wugs;
var y : wugs = (wugs) x;
```

Figure 3.1: A sample program with user defined type conversion.

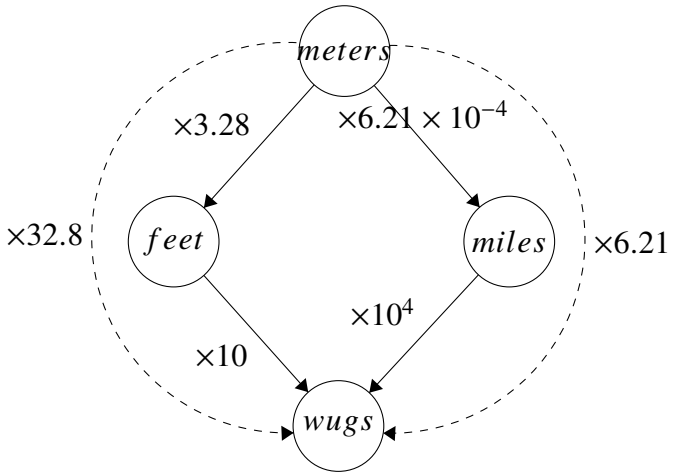

Figure 3.2: In this sample program, the user implicitly defines two ways to cast variable a from meters to the new unit wugs. The definitions are different, and a compiler performing implicit conversion would not know which to choose.

This paper is about efficiently checking commutativity in diagrams that arise in real systems. We assume a simple equivalence checker for individual transformation functions: in our type system example, for instance, it is possible to check transformation equivalence by comparing the conversion factors. Our aim is to analyze the graph of transformations and minimize the number of times we need to perform an equivalence check. Since diagrams may change over time in real systems, as new conversions are added, and verifying the entire system from scratch may be computationally expensive, we want an online method that only checks the impact of new edges. In Figure 3.2, for example, a run-time system can catch the point where the programmer adds a bad conversion definition by verifying each new conversion edge as it is created.

Efficient commutativity checking is not trivial. The presence of cycles implies a potentially infinite number of paths.. Further, naïvely checking if all path pairs that begin and end at the same node in a given diagram commute could require a number of function equality checks that grow as factorial in the number of nodes, because a path consists of an ordering of nodes. Previous work [**?**] has identified an $O(|E|^2|V|^4)$ algorithm to verify that a complete acyclic diagram commutes; however, it addresses neither online

addition nor cyclic diagrams.

For verifying commutativity over online addition, we identify two key insights. First, when a new edge is added, only one path per source and sink pair needs to be checked against the existing commutative diagram. Because the diagram commutes, all the paths between a given source and sink are equal and a representative to check against can arbitrarily be chosen. This leads to an $O(|V|^2(|E| + |V|))$ algorithm to verify a diagram remains commutative over the course of online addition, assuming an oracle to check the equality of functions. The algorithm makes an *asymptotically* optimal number of calls to the oracle.

Second, there is a single rule that places a partial, transitive ordering on paths indicating the amount of information they contain about other paths. This insight yields a greedy $O(|V|^4)$ optimization step that results in the number of oracle calls being exactly minimal. The optimization is critical when equality checking is expensive.

We evaluate our algorithms against random graphs and use them in two case studies. First, we use our algorithm in the domain specific geometry type language *Gator* [**?**] to ensure that user defined transformations between spaces stay consistent. Second, we use our algorithm to identify inefficiencies in a currency conversion graph. We empirically compare our solution to three baseline implementations: a naïve cycle-sensitive all-pairs check, a check for all path pairs that involve the new edge, and an algorithm suggested by previous work to solve the batch version of the problem for acyclic diagrams. Our proposed algorithms run orders of magnitude faster than the baseline implementations.

3.2 Formal Problem Setup and Terminology

We start by formalizing the notion of a diagram, drawing terminology from the previous acyclic work by Murota [**?**].

Notation. We start with a directed graph $G = (V, E)$, where V corresponds to sets of elements and edges (u, v) in E correspond to functions that maps elements of u to elements in v. These functions form a semigroup $\mathcal{F}$, where multiplication is function composition. A semigroup consists of a set and an associative binary operation, which we use to capture function composition. The correspondence between edges and functions is stored as a mapping $f : E \to F$, where f maps each edge to the function it represents.

A path is a sequence of edges. The edge-to-function mapping f can be naturally extended to paths: if path $p{=}e_1; \cdots ; e_n$ then $f(p) = f(e_1); f(e_2); \cdots ; f(e_n)$. We write $\partial(p)^+$ for p's start node, $\partial(p)^-$ for its end node, and $\partial(p)$ to denote the pair $(\partial(p)^+, \partial(p)^-)$.

A pair of paths p_1 and p_2 is said to *parallel* iff their terminal nodes are the same, i.e., $\partial(p_1) = \partial(p_2)$. ∂, ∂^+ and ∂^- are extended to apply to parallel pairs. For parallel pair ϕ = (p_1, p_2), $\partial(\phi) = \partial(p_1) = \partial(p_2) = (\partial(\phi)^+, \partial(\phi)^-)$.

Let $\mathcal{R}_{all}$ be the set of all parallel pairs of paths in a given diagram. The diagram commutes iff $\forall (p_1, p_2) \in \mathcal{R}_{all}$, $f(p_1) = f(p_2)$; that is, the composition of maps along any path connecting any pair u to v is independent of path choice.

Problems. The Online addition problem, given a commuting diagram and a new edge, returns whether the diagram commutes. Checking function equality is a domain specific, potentially hard problem, dependent on the nature of the graph. For example, in

our case study in graphics programming (see Section 3.5.1), edges are matrices and nodes are vector spaces, so function composition uses matrix multiplication and equivalence checking simply compares matrix values. We therefore assume some oracle for checking transformation function equivalence that will vary by domain. We therefore collapse the ONLINE ADDITION PROBLEM to the VERIFICATION SET PROBLEM; we solve the latter and assume an oracle with the results to produce the former. The latter, when given a diagram and a new edge, returns the set of parallel pairs of paths, such that if and only if the members in each pair have function equivalence, then the new graph must commute. The output to the ONLINE ADDITION PROBLEM can then be obtained as whether function equivalence checking for all pairs succeeds.

The algorithms in this paper assume that the function equivalence oracle is reflexive, symmetric, and transitive.

3.3 Baseline Algorithms

To examine the efficacy of our proposed solution to the VERIFICATION SET PROBLEM, we compare it to some potential alternatives. Specifically, we examine a naïve factorial algorithm, a slightly less naïve factorial algorithm which we identify to be a two-flip tolerant path search, and Murota's previous batch solution [**?**].

3.3.1 Naïve Baseline Algorithm

Our first goal is to develop a baseline (exponential) algorithm that can reason about cycles without producing an infinite set of paths. This algorithm will first pare the structure of the graph down to remove cycles, extract the pairs of paths in the graph, and finally

reason about each pair to check commutativity. This results in two components C and Q: the cycle verification pairs and acyclic parallel pairs, respectively.

We start with the set of all parallel pairs in the diagram. We pare it down to be finite by handling cycles: using a procedure like Johnson's algorithm [**?**], we find all simple cycles in the diagram. We create a cycle verification set, C, and verify for each cycle that a single traversal is equal to the identity function by adding $(v \to v, 1)$ for each node v in the cycle to C. Here, $v \to v$ is a simple cycle starting and ending at v, 1 is the identity function, and these must be verified to be equal to each other.

We then create a set $\mathcal{P}$ of all the paths in the diagram with no cycles, and filter the set $\mathcal{P} \times \mathcal{P}$, excluding pairs where the paths begin or end on different nodes, or are identical, to get the set of all cycle-free parallel pairs Q. After verifying C, it is sufficient to verify only Q (as opposed to all pairs) because cycles must now be the identity, so for any pair in the set of all parallel paths, any instance of a cycle can be removed to obtain an equivalent pair with shorter, cycle-free paths.

If the shorter pair has equal paths then the paths in the original pair must also be equal to each other. It is therefore safe to remove all pairs of paths with cycles, leaving only parallel pairs where neither path has a cycle. $\mathcal{P}$ is finite, bounded by $2^{|V|}$, as a path without cycles is an ordering on nodes, each node occurring at most once. $|\mathcal{P} \times \mathcal{P}|$, and consequently, $|Q|$, are also finite, bounded by $2^{2|V|}$. Thus the algorithm terminates and returns a finite (if large) set.

3.3.2 Baseline Incremental Algorithm

For an incremental algorithm, we explore how the addition of an edge can change the baseline to looking at a subset of the graph rather than every pair of paths. In achieving this, this second baseline essentially refines the results of the naïve; a similar structure, but with a substantially reduced set of paths to examine.

Like before, we start by creating a cycle verification set C', but includes only the simple cycles that pass through the new edge. Then, instead of Q, the set of all non-cyclic parallel pairs, the algorithm obtains its subset Q' consisting of all non-cyclic parallel pairs such that exactly one path in each pair passes through the new edge. To this end, the algorithm performs a *two-flip tolerant path search* whose output is passed into a *path extraction algorithm*; this search finds all parallel pairs for which only one path includes the new edge. The result of the path extraction algorithm to get the final output $Q' \cup C'$.

This narrowing can be done because the original diagram commutes. Pairs where both paths do not involve the new edge would remain equal (this would apply to cycles too; cycles that do not pass through the new edge must be the identity). Also, pairs where both paths involved the new edge would have to be equal. To see why this is true, each path could be thought of as consisting of the composition of three segments. For path pair p, and new edge from node S to node T, the first segment extends from $\partial(p)^+$ to S, the second, the new edge (S, T) itself, and the third, from T to $\partial(p)^-$. The new edge could only appear once because cycles have already been dealt with so only pairs where the path includes the new edge once need be checked. The first segment of both pairs would have to be equal because they existed as parallel pairs in the original diagram, and similarly the third segment would also have to be equal. The second segment, consisting of the same edge, would also have to be equal because the equivalence oracle is reflexive. A composition of these three equal components would be then be equal, since the oracle

would preserve transitivity of equality. We are left only with parallel pairs where exactly one of the paths passes through the new edge.

To resolve this algorithm fully, we will need to define the specifics of the two-flip tolerant path search and how to narrow down the results of this search into an actual set of paths to verify.

Two flip tolerant path search We use a "two-flip tolerant" path search from the source (S) to the sink (T) of the new edge to identify the pairs of paths where exactly one path includes the new edge.

In a normal directed graph path search, only forward edges, i.e., edges that go outward from the current node while executing the search are considered. A *two flip path* consists of up to three phases: in the first phase, only backward edges—pointing inward to the source of the search—are accepted. In the second phase, only forward edges are accepted, and in the third phase, again only backward edges are accepted. For a two flip tolerant path p, let $t_1(p)$ map to the first phase, $t_2(p)$, to the second, and $t_3(p)$, to the third. The node between the first two phases we refer to as the *first flipping point*, which has both edges pointing outward; similarly, we refer to the node between the latter two phases as the *second flipping point*, at which both edges point inwards.

We present the idea diagrammatically in Figure 3.3. Squiggly arrows represent path phases (these are the composition of zero or more edges, not a single edge). The new edge is represented with a dashed arrow. Here, $f_1; f_2; f_3$ is a two flip path, and $f_1; (S,T); f_3$ is a new path created because of the addition of (S,T) that forms a parallel pair with f_2.

The two flip tolerant path search returns the set of all paths between a given source and sink that have up to two flips (paths with zero or one flip are also accepted).

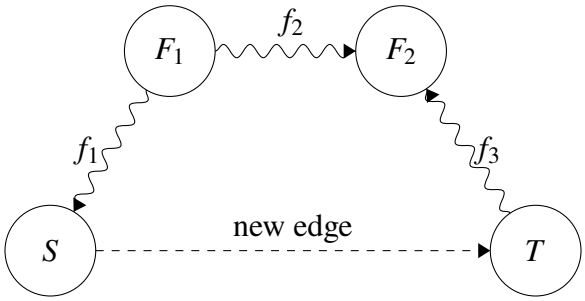

Figure 3.3: Two flip tolerant path.

Path extraction algorithm Next, the *path extraction algorithm* th en transforms the output of the two flip path search into the verification set, $Q' \cup C'$. Given a set of two flip tolerant paths from the new edge source to sink, the algorithm outputs a set of pairs to verify.

Let the new edge added to the diagram be (S, T) and the input set of paths, $\mathcal{P}$. The algorithm processes every two flip tolerant path p in $\mathcal{P}$ case-wise to obtain pairs to add to the output set.

- In the case where p has two flips, $t_1(p); (S, T); t_3(p)$ and $t_2(p)$ form a parallel pair.
- When p has only the first flip (which is to say, the third phase of the path is missing), the parallel paths are $t_1(p); (S, T)$ and $t_2(p)$.
- Similarly when only the second flipping point is present (so that there is no first phase), then the parallel pair is $(S, T); t_3(p)$ and $t_2(p)$.
- Finally when no flipping points are present, there are two possibilities: Either p is a path from S to T, in which case the parallel paths are simply the edge (S, T) and p, or p is a path from T to S. In this case, we have found a cycle, $p; (S, T)$, to be paired with the identity function. Like with the naïve algorithm, for every node v in the cycle, we add the pair $(v \to v, 1)$, where $v \to v$ is the cycle $p; (S, T)$ written to start and end at v.

Resolving the Incremental Algorithm We conclude our discussion of this incremental algorithm by proving that the result is the same as if we were running the naïve baseline algorithm. This in turn shows that we have found a more efficient algorithm to achieve the same result of providing a set of paths which can be used to check commutativity.

Theorem 2. *Perform the two-flip tolerant path search from the source to sink node of the edge that is to be added followed, and on the output, apply the path extraction algorithm. The result is the set $O = Q' \cup C'$ of new parallel pairs with exactly one path passing through the new edge and neither paths containing any cycles, and the set of simple cycles passing through the new edge.*

Proof. Every element in the output of the path extraction algorithm was by construction an element of O. Every cycle in C' can be expressed as $(S, T); p$, and corresponds to the input two flip tolerant path p.

It remains to show that every new parallel pair p in Q' corresponds to a two flip tolerant path. Let $\partial(p)^+ = F_1$ and $\partial(p)^- = F_2$. Only one path passes through (S, T). Let it be called p_1, and the other path, p_2. The two flip tolerant path from S to T can be constructed as follows: phase 1 is the segment of p_1 from F_1 to S, phase 2 is p_2, and phase 3 is the segment of p_1 from T to F_2. Effectively, F_1 corresponds to the first flipping point, and F_2, to the second. It is possible that some of F_1, F_2, S and T coincide (e.g., p starts at S, i.e., $F_1 = S$), in which case the corresponding segments between the coinciding nodes can be considered the identity; the resultant path simply has fewer than two flips. □

Analysis An upper bound on the number of pairs that this algorithm returns is $O(|V|^2 2^{|V|})$, since two flip tolerant paths are an ordering on nodes, each node appearing at most once, followed by a selection of the flip points. In practice, the algorithm

significantly outperforms the naïve batch baseline because it looks only at parallel pairs that involve the new edge, which is usually a small subset of all parallel pairs. Empirical results are presented in Section 3.6.

3.3.3 Optimal Batch Solution

Murota's main result [**?**] solves the batch version of VERIFICATION SET: given an acyclic diagram, it returns the minimal set of equality checks that succeed if and only if the diagram commutes. Murota describes an algorithm to find the ($|V|^2|E|$ bounded) minimal set of pairs that need be checked.

The approach in this algorithm, at a high level, is to define a function that takes in a subset of pairs and returns the subset of pairs whose equivalence is implied by the equivalence of the pairs in the input set. Then the algorithm greedily eliminates redundancies until a minimal set is reached.

A bilinking is defined to be a parallel pair that is disjoint but for their terminal nodes. The set of all bilinkings is $\mathcal{R}_0$. In an acyclic diagram, if all bilinkings are equal, all parallel pairs must also be equal since any given pair can be expressed as a composition of bilinkings.

Define $r_1 > r_2$ for bilinkings $r_1 = \{p_1, q_1\}$, $r_2 = \{p_2, q_2) \in \mathcal{R}_0$, if there exists a path p such that $\partial(p) = \partial(r_1)$ and p contains p_2. Define $\langle\rangle$ as: $\langle r\rangle = \{s \in \mathcal{R}_0 | r > s\}$.

For bilinking s, let $F(s)$ be the vector in $\mathrm{GF}(2)^{|E|}$ (where GF(2) is the Galois field, that is, finite field of two elements) representing the edges present in s (the n^{th} dimension of $F(s)$ is 1 if the corresponding edge is in s, and 0 otherwise). Let this function be extended to sets, so that for some set of bilinkings $\mathcal{S}$, $F(\mathcal{S}) = \{F(s) | s \in \mathcal{S}\}$. A notion of

Result: Find a spanning set Rs = [r_1, ... ,r_k].
Graph existingGraph
$R_s \leftarrow \{\}$
foreach *node v in V* **do**
    subgraph ← existingGraph.extractReachableSection(v)
    `/* Get the portion of the graph that can be reached starting from v. */`
    tree ← createMinimumSpanningTree(subgraph)
    excludedEdges = edges in subgraph - edges in tree
    foreach *edge e* ∈ *excludedEdges* **do**
        firstPath = tree.findPath(source: e.source, sink: e.sink)
        R_s.addElement((firstPath, e))
    end
end
return R_s

Algorithm 1: Finding a spanning set of path pairs, as in section 3.3.3.

linear independence in this vector field exists.

For a set of bilinkings $\mathcal{R}$, the closure function cl is defined as: $cl(\mathcal{R}) = \{s \in \mathcal{R}_0 | s$ is linearly dependent on $F(\mathcal{R})\}$. The closure function on $\mathcal{R}$ captures all the pairs that can be made by made by composing or "gluing together" the bilinkings in $\mathcal{R}$. Using these two functions, we define the function σ on a set of bilinkings $\mathcal{R}$ as $\sigma(\mathcal{R}) = \{s \in \mathcal{R}_0 | s \in cl(\mathcal{R} \cap \langle s \rangle)\}$. This is the function used to capture all the pairs whose equivalence is implied by the equivalence of pairs in $\mathcal{R}$. We use σ to iteratively check if a given pair is redundant. We eliminate Bilinkings until we reach a minimum "spanning" subset.

Roughly, the algorithm proceeds by first efficiently finding a *spanning* set of bilinkings (a subset whose verification implies the verification of all bilinkings in the graph). It does this, starting at every node, by finding the reachable subsection of the graph, and a spanning tree for the subsection. From each edges in the reachable section that is not a part of the tree, it generates a bilinking using the edge and a path in the tree that is parallel to the edge (Algorithm 1).

```
Result: Find a minimal spanning set of path pairs (bilinkings) R.
Function σ (input set S, spanning set Rs) :
    output ← {}
    for bilinking ∈ Rs do
        smallerPairs ← allShorterPieces(bilinking)
        /* Get fragments that could build up to the
           bilinking.  Corresponds to applying <>
           function.                                  */
        consideredPieces ← smallerPairs ∩ S
        /* Now see if bilinking can be built from these
           pieces.                                    */
        /* Linear independence is in GF(2) as in
           algorithm description.                     */
        if linearlyDependent(consideredPieces, bilinking) then
            output.add(bilinking)
        end
    end
    return output
R ← Rs
for i=1 to K do
    if r_i ∈ σ(R-r_i) then
        R ← R-r_i
    end
end
return R
```

Algorithm 2: Finding a minimal spanning set, as described in section 3.3.3.

With the spanning set thus initialized, it greedily tries to remove each pair from the spanning set if the set remains spanning even after removing the edge (Algorithm 2).

The proof of correctness can be found in Murota[**?**]. The number of checks returned by the algorithm is at worst $O(|V|^2|E|)$. The overall run time of an optimized implementation is $O(|V|^4|E|^2)$.

3.4 Solving the Online Addition Problem

We present a polynomial time solution to the VERIFICATION SET PROBLEM. As in the online baseline algorithm, we do not concern ourselves with parallel pairs where neither or both paths pass through the new edge.

The key observation allowing us to improve on the online baseline is a result of Theorem 3 (which we expand on later): for a given source and sink pair, only a single parallel pair needs to be verified. It is straightforward to see that, should our selected set of pairs and cycles passing through the new edge be verified commutative, the entire diagram must commute. Algorithm 3 uses this strategy of identifying a parallel pair with exactly one path through the edge for each (source, sink) pair.

The `try` block is executed at most $O(|V|^2)$ times, which is also the bound on the number of pairs verified. This bound is asymptotically tight, as can be seen in the case where the graph contains $2N$ nodes along S and T. Imagine dividing the nodes into two groups of N nodes each. Every node in group 1 has a forward edge to every node in group 2 and to S. T has a forward edge to every node in group 2. In this diagram, when adding edge (S, T), N^2 paths need to be verified which is polynomial in the total number of nodes, $2N + 2$.

If trying to optimize for path length (say, if composing functions is expensive) then "find any path" can be replaced with "find the shortest path."

An efficient implementation of the algorithm can run in $O(|V|^2(|V| + |E|))$ time, with space complexity not exceeding the asymptotic $O(|V|^2)$ bound on the output. In such an implementation, path finding from a given source node to all potential sink nodes could be done in a single $O(|V| + |E|)$ breadth first search.

```
Data: existing graph, new edge.
Result: Set of parallel pairs to verify.
Graph existingGraph; Edge newEdge;
parallelPairs ← {}
for src in existingGraph.Nodes do
    for snk in existingGraph do
        try:
            /* Use any standard path finding algorithm
               such as BFS to find a path in the existing
               graph from the specified source to sink.
               */
            Path pathWithNewEdge ← FindPath( sourceNode: src, sinkNode:
             newEdge.Source) +
            newEdge +
            FindPath( sourceNode: newEdge.Sink, sinkNode: snk)
            if src == snk then
                /* Assign the nullary path from src to snk.
                   */
                pathInOldGraph ← src
            end
            else
                Path pathInOldGraph ← FindPath( sourceNode: src, sinkNode:
                 snk)
            end
            parallelPairs.add((pathInOldGraph, pathWithNewEdge))
        catch PathFindingFailedException:
            /* No comparable pairs from node src to node
               snk that need to be checked              */
            continue
        end
    end
end
return parallelPairs
```

Algorithm 3: Online polynomial time algorithm to find parallel pair set.

3.4.1 Optimization Step

In the case where equality checks are very expensive, we begin by finding the minimal set of (source, sink) pairs such that checking for these pairs logically implies having checked the full diagram.

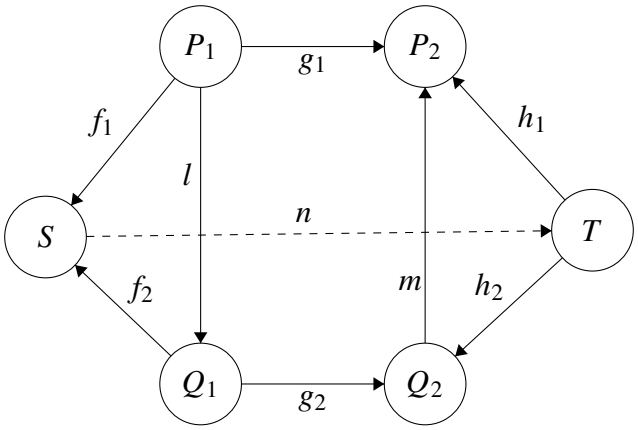

Figure 3.4: Reduction rule. Each arrow represents a path, where n is the new edge being added. While Algorithm 3 returns two pairs for verification, one from P_1 to P_2 and the other from Q_1 to Q_2, it actually suffices to just check a pair from Q_1 to Q_2 as demonstrated in theorem 3.

If Algorithm 3 were applied to the diagram shown in Figure 3.4 there would be redundancies in the output. It turns out that verifying $g_2 = f_2; n; h_2$ is sufficient to ensure the diagram still commutes on the addition of n.

Theorem 3. *If parallel paths $g_2 = f_2; n; h_2$ then it must be that $g_1 = f_1; n; h_1$.*

Proof. We use the fact that f_1=l; f_2 and h_1=h_2; m.

$$g_2 = f_2; n; h_2 \Rightarrow l; g_2 = l; f_2; n; h_2$$

$$\Rightarrow l; g_2; m = l; f_2; n; h_2; m \Rightarrow g_1 = f_1; n; h_1$$

The proof holds if any of the paths used are the identity, e.g., if f_1 is the identity so S and P_1 are the same node. □

We conclude that verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of all path pairs (Q_1, Q_2) such that Q_1 is a successor of P_1 and P_2 is a successor of Q_2. A successor S to node N is any node such that there exists a path

from N to S. Nodes are also their own successors and predecessors. The rule effectively places an ordering on the informativeness of path pairs based on their terminal nodes.

Given that a set of path pairs are equal, suppose we attempt to derive the proposition that a different parallel pair of paths is equal with a step-by-step application of inference rules. Under the assumption that edges are generic functions, and no other information is available, $\mathcal{F}$ is a semi-group. The only inference rules allowed are composition (given that $f_1 = f_2$, it must be that $g; f_1 = g; f_2$) and replacement of one path by a different, equal path (given $f_1 = f_2$ and $g; f_1 = h; f_1$, it must be true that $g; f_1 = h; f_2$). Any permutation of the repeated application of these two rules results in the "reduction rule" already described; it is therefore the only rule that can be used to reduce the set of path pairs to check.

That is to say, if verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of a pair with endpoints (Q_1, Q_2), then it must be that Q_1 is a successor of P_1 and P_2 is a successor of Q_2.

Using this information it is possible to choose a minimal subset of path pairs to verify, as in Algorithm 4. To summarize this algorithm conceptually, we start by constructing a graph with a node for each possible (source, sink) pair in the graph: each node then represents a possible choice for parallel pair endpoint pairs. Edges are drawn from node (P_1, P_2) to (Q_1, Q_2) if Q_1 is a successor of P_1 and P_2 is a successor of Q_2. We greedily search for the smallest set of nodes from which the entire graph would be reachable. The idea is to look for "roots" in the graph that have to be included in the ultimate verification set because they have no predecessor in the graph and cannot be verified "through" the verification of some other pair. Then all the successors whose verification is implied by the roots are eliminated.

```
Data: Existing graph, new edge.
Result: Set of parallel pairs to verify.
Graph existingGraph
Edge (S, T)
predecessors ← predecessors of S in existingGraph
successors ← successors of T in existingGraph
Graph terminalPairGraph ← empty
for q ∈ successors do
    for p ∈ predecessors do
        terminalPairGraph.addNode((q, p))
        for predecessor ∈ predecessors of q in existingGraph do
            for successor ∈ predecessors of p in existingGraph do
                terminalPairGraph.addEdge((predecessor, successor))
            end
        end
    end
end
verificationSet ← {}
while terminalPairGraph.nodes not empty do
    currentNode ← terminalPairGraph.node
     // an arbitrarily chosen node of terminalPairGraph
    while currentNode has predecessors do
        currentNode ← predecessorOfCurrentNode
         // an arbitrarily chosen predecessor of current
        Node
    end
    verificationSet.add(currentNode)
    terminalPairGraph.removeAllSuccessors(currentNode)
end
return verificationSet
```

Algorithm 4: Minimal set finding algorithm.

At the end of the greedy graph reduction we are left with the unique set of root nodes. The only way to reduce the set of parallel pairs is to apply the reduction rule of theorem 3, but all the ways in which the rule is applicable was already captured in the edges of the graph. The leftover set has no edges and no scope for further reduction.

Also, the verification of the parallel pairs returned in the algorithm implies that the output of the previous algorithm must commute and that the entire diagram must

commute.

The run time of the first step is $O(|V|^4)$, and that of the second step is $O(|V|)$, so that the overall bound is $O(|V|^4)$. Space complexity remains $O(|V|^2)$.

3.5 Case Studies

To demonstrate our algorithms applied to a real world situation, we search for inconsistencies in diagrams of geometry transformations, and in a diagram of the exchange rate between currencies. Each of these applications use commutative diagrams, and the commutative nature of each is necessary to reason about some form of correctness. We explore these examples with the intent of showing that the algorithms discussed apply to realistic settings and potentially identify real-world examples of incorrect behavior.

3.5.1 Gator

Gator is a domain specific language designed around geometry types, which are used to describe properties and transformations of geometric objects [**?**]. A key feature of Gator is `in` expressions, which insert code to automatically transform between two geometry types. For example, given a point `p` represented in 2-dimensional Cartesian coordinates (which has type `cart2`), we can transform this point into polar coordinates using the expression `p in polar`. These `in` expressions create a structure of commutative diagrams, allowing use as introduced in Section 3.1.

Specifically, Gator introduces transformations between *reference frames*, which are the geometry equivalent of transforming between linear algebra basis vectors. Each

edge on our transformation graph is thus a matrix, with composition of edges as matrix multiplication and an oracle checking matrix equality (up to a rounding error ϵ).

There are several examples of reasonably complicated transformation graphs that we can pick from. Gator includes graphics examples as part of its examples package, all of which are in the Gator paper; for this evaluation, we looked at the *phong*, *reflection*, and *shadow map* examples.

We implemented a system for interfacing between the optimal set path checker (Algorithm 4) in the open-source implementation of Gator. The system was tested with intentional bugs, of which it found them all, although no "real" bugs were found. The graphs used were of size 5 or less; for graphs of this size, the checker was able to run in real time with no noticeable loss of frames. Since the program is running at 60 frames per second, the checker was running at a rate faster than .01 seconds.

3.5.2 Currency Graph

We imagine a units-of-measure type system as being an interesting application of concurrency graphs; however, to make this more interesting and scale nicely to large graphs with existing data, we focus on the specific unit of currencies. Consider a diagram with nodes as currencies and a directed edge being the conversion rate from its source node's currency to its sink node's currency. Since the exchange rate of money from any given base currency to a target currency can be expected to be the same regardless of which intermediate currency transformations are used, this diagram should commute.

Using a web API[1] for currency data, we built the fully connected diagram of exchange rates between 32 currencies on a given day. To ensure that it indeed commuted, we

[1]https://exchangeratesapi.io

started with an empty diagram, and added in edges one by one. Before the addition of each edge, we used the algorithms (Algorithm 3 and Algorithm 4, the online polynomial and online minimal set algorithms, respectively) to ensure the addition of a new edge did not introduce inconsistencies in the existing diagram. If a new edge was problematic, the algorithms returned an example inconsistent pair that would arise from the addition of the edge. The pair would consist of two currency transformation sequences with the same source currency and ultimate destination currency, but with different effective exchange rates values, as computed by taking the product of all the exchange rates encountered through the chain.

We allowed an "error tolerance" so that differences reported would not be the trivial consequences of a floating point error. However, this relaxation of the equality oracle into imprecision meant that the mathematical reasoning that allow the algorithms to remove redundant path checks no longer applied. For instance, composing a new function with two approximately equal functions does not lead to equal results, so Theorem 3 fails with this approximate equality. When the algorithms reported no inconsistencies, it was still possible that the graph possessed inconsistencies above the given threshold and did not commute. Nonetheless, both algorithms were effective in catching inconsistencies. Algorithm 3 started finding inconsistencies at error tolerances to the order of 10^{-3}, and Algorithm 4, which makes more invalid redundant path check removals, at error tolerances to the order of 10^{-7}.

Averaging over evaluation for the first 30 days of 2020, building and verifying a diagram to completion (inclusive of the time required by network calls) took 243±19 seconds using Algorithm 4, and in 133±13 seconds with Algorithm 3. For this large of a graph and data set, these times are reasonable and show these algorithms can be used in a realistic setting. Finding actual inconsistencies further shows the value of using these

Table 3.1: Computation time for 9-node graph of density 0.4, averaged over ten runs.

Algorithm	Average seconds of computation
Naïve baseline	0.77
Two Flip tolerant	0.075
Batch algorithm	7.55
Algorithm 3	0.0038
Algorithm 4	0.00086

algorithms and commutative diagrams in the real world.

3.6 Evaluation

We compare performance of the following path checking algorithms: (1) the naïve baseline, (2) the less naïve two-flip baseline, (3) the batch baseline, (4) Algorithm 3, the non-minimal polynomial-time algorithm, and (5) Algorithm 4, the minimal set finding algorithm. The two metrics we evaluate are time for response and size of response set (smaller sets—tighter output results—would mean less calls to the oracle). We use randomly generated graphs of varying size: given a graph and a new edge, we time how long it takes for an algorithm to return the set of pairs that need to be verified. All computations were performed on a MacBook Pro 2015, 2.9 GHz dual-core Intel Core i5.

3.6.1 Comparison of Algorithm Time Cost

The average time taken by each algorithm over the course of 10 runs over randomly generated graphs with 9 nodes and 32 edges is listed in Table 3.1.

The naïve baseline performs poorly, taking well over a thousand seconds for even small graphs of 10 nodes. While the batch algorithm improves on this, it still does not

Table 3.2: Output size for 9 node graph of density 0.4, averaged over ten runs.

Algorithm	Average number of output pairs
Naïve baseline	39754.9
Two Flip tolerant	748.9
Batch algorithm	23
Algorithm 3	78.3
Algorithm 4	1

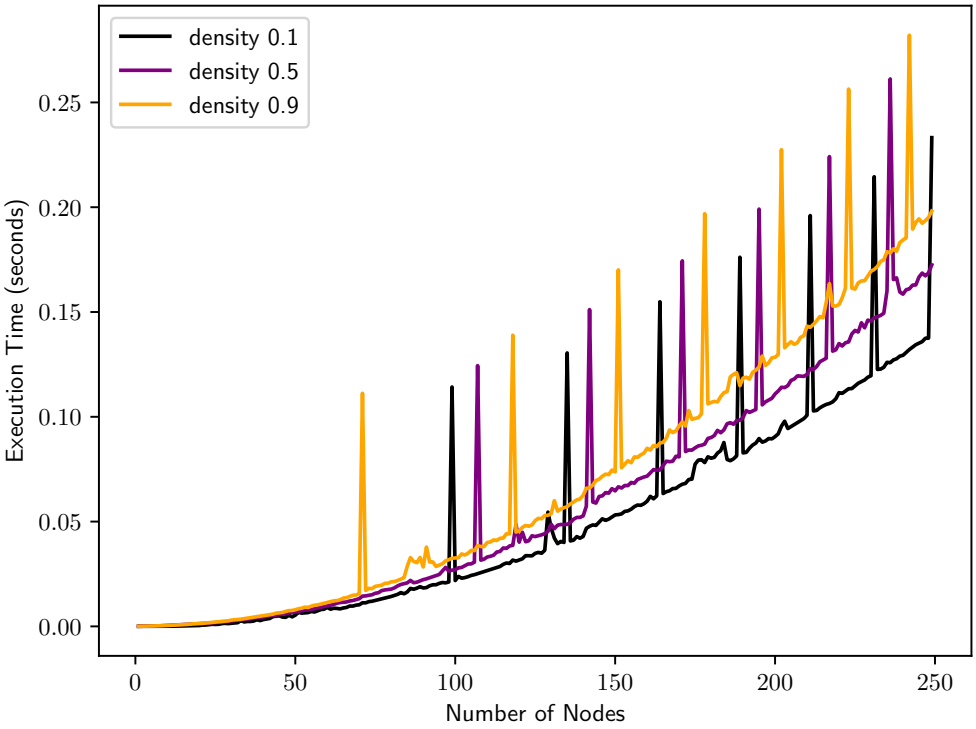

Figure 3.5: Algorithm 4.

scale very well, with computation for a graph with 14 nodes and 0.4 density taking hours. Our implementation does not memoize the construction of the vector and matrix representation of paths in GF(2); profiling indicates that this construction is a major factor in the high time cost for this algorithm. Algorithm 3 performs only slightly better than the batch algorithm. Surprisingly, the optimal set algorithm cuts time cost by several orders of magnitude, and runs in milliseconds for small graphs. All implementations are sensitive to density, performing better when density is low.

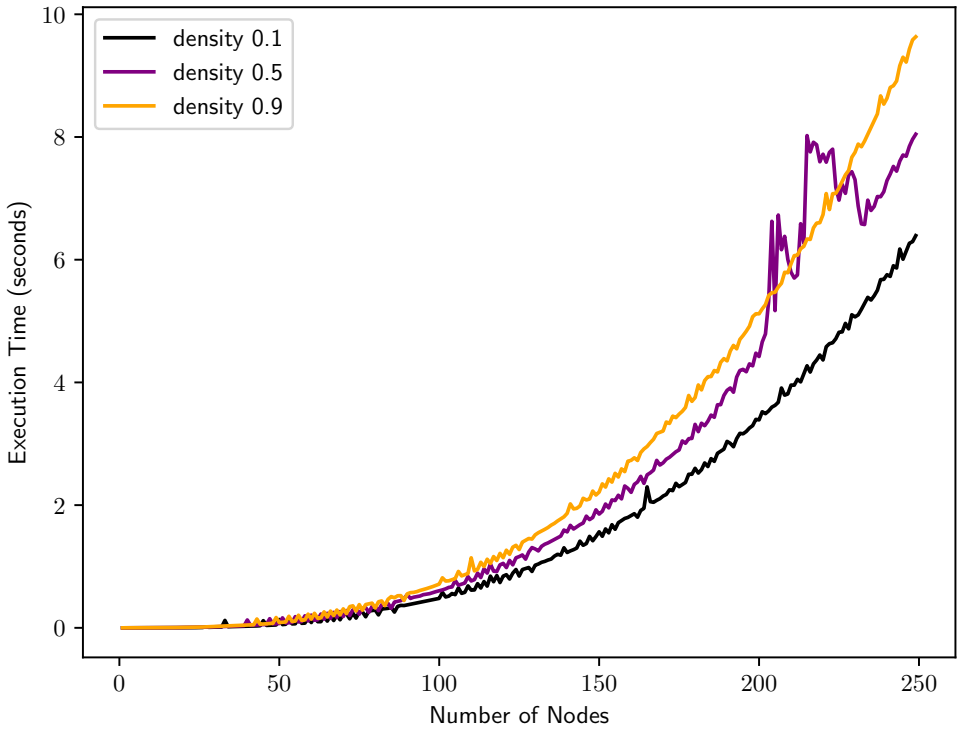

Figure 3.6: Algorithm 3.

3.6.2 Scaling of Time with Input Size

Figure **??** shows that the algorithms' time scales with size, as expected. [The plots xxx are weirdly tiny on my machine... any chance your plotting tool can output PDF images? —AS] Both Algorithm 4 and Algorithm 3 exhibit graphs that are polynomial in appearance. The naïve baseline as well as the two flip tolerant baseline display quick growth. The batch algorithm also grows fast, though not as much as the online checking baselines.

We define density to be the ratio of the number of edges in the graph to the total possible number of edges (which is $|V|^2$, where $|V|$ is the number of nodes). Run time relates to the density of edges in the input graph. The degree of the effect differs with the algorithms, as Figure **??** shows. Generally, denser graphs entail longer computation time.

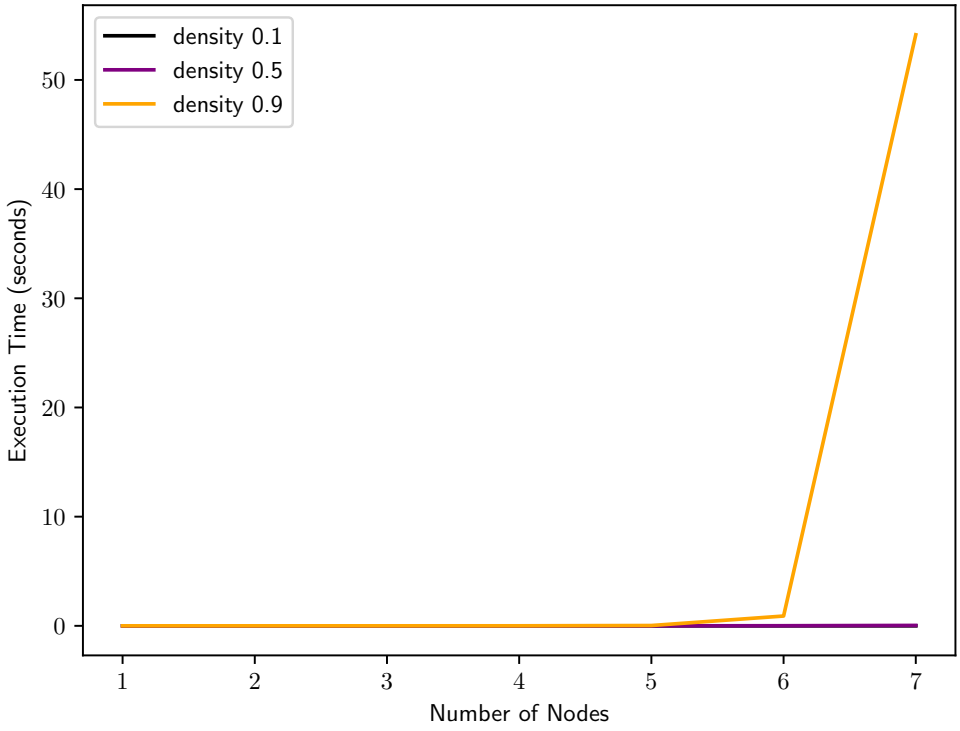

Figure 3.7: Naive baseline.

For the batch algorithm we use lower densities since the input graph must be acyclic. This puts an upper bound on density that approaches 0.5 in large graphs.

3.6.3 Variance

The periodic spikes in Figure 3.5 are striking. We plot the spread of results in Figure 3.10 to understand what is happening. Grey points are the results of evaluation on individual points, and error bars show standard deviation. The black curve traces the mean. We find Algorithm 4 has outliers about two standard deviation above the mean responsible for the spikes in the average. The outliers themselves follow a polynomial curve, appearing almost periodically. We have not yet identified the cause of the behavior. Figure **??** depicts the situation for Algorithm 3, where no such effect is observed.

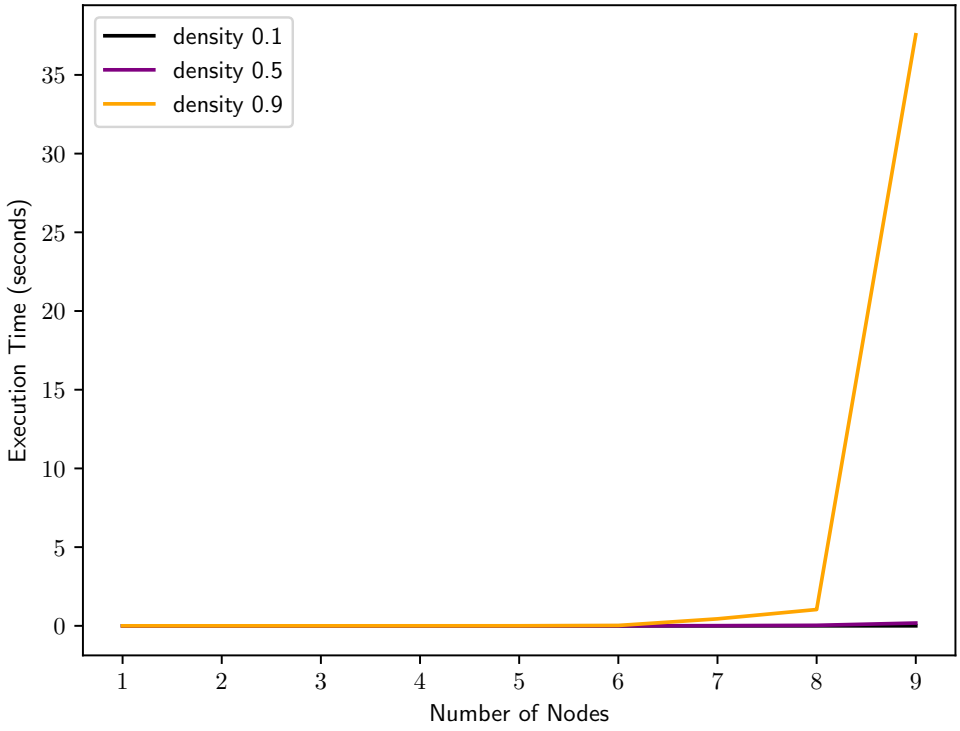

Figure 3.8: Two flip tolerant baseline.

3.6.4 Size of Output

Output size is a metric of interest, should the equality checking oracle be expensive. Table 3.2, summarizes the number of output pairs that the algorithms returned on average over 10 runs, for graphs with 9 nodes and 32 edges. These results are essentially as expected, although it is interesting to note that Algorithm 3 produces around triple the number of pairs compared to the batch algorithm. Also note that Algorithm 4 produces the minimal number of paths, showing why it is the minimal set algorithm.

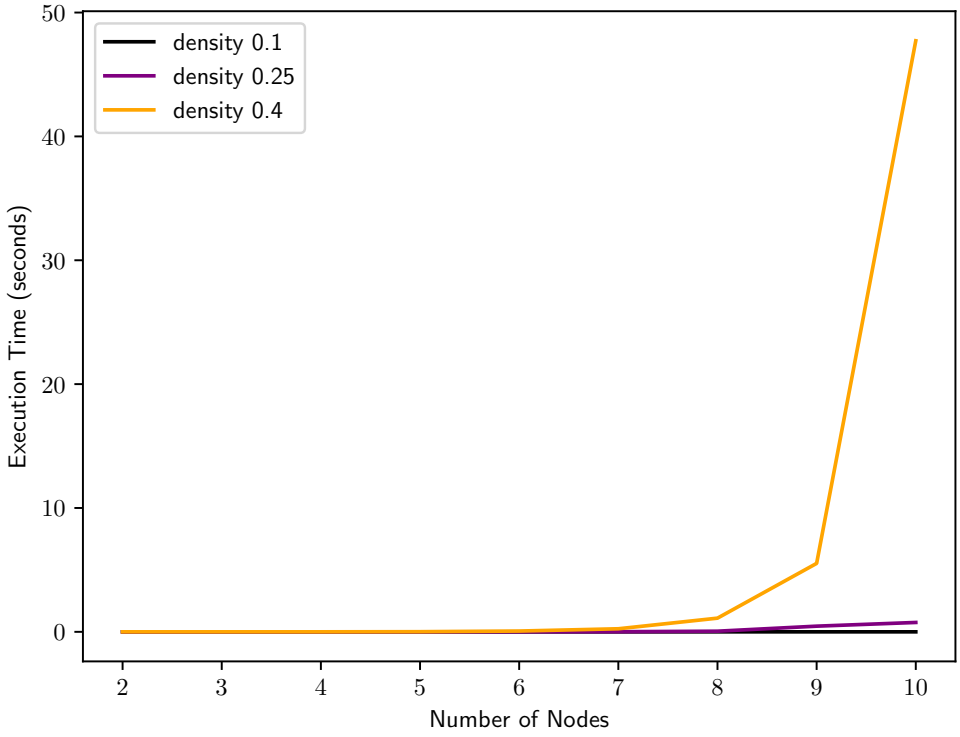

Figure 3.9: Batch algorithm baseline.

3.7 Related Work

Section 3.3.3 describes Murota's solution to efficiently finding the minimal set of path pairs that need to be compared to check if a given acyclic graph commutes [**?**]. We did not find any other work that solves the question of verifying that diagrams commute. However, the question of commuting does come up in programming languages with implicit type conversion. Gator [**?**], as described in Section 3.5.1, supports automatic type conversion between geometry types. The language implements some restrictions to eliminate obvious cases of non-commuting graphs, but does not verify that defined graphs commute, allowing scope for non-commuting graph definitions. Frink [**?**] is a language that supports automatic conversion between units and infinite precision floating point numbers. It does not appear to support the implicit definition of conversion between units [Need to spend some time confirming this —AK] but if extended to do so, would xxx

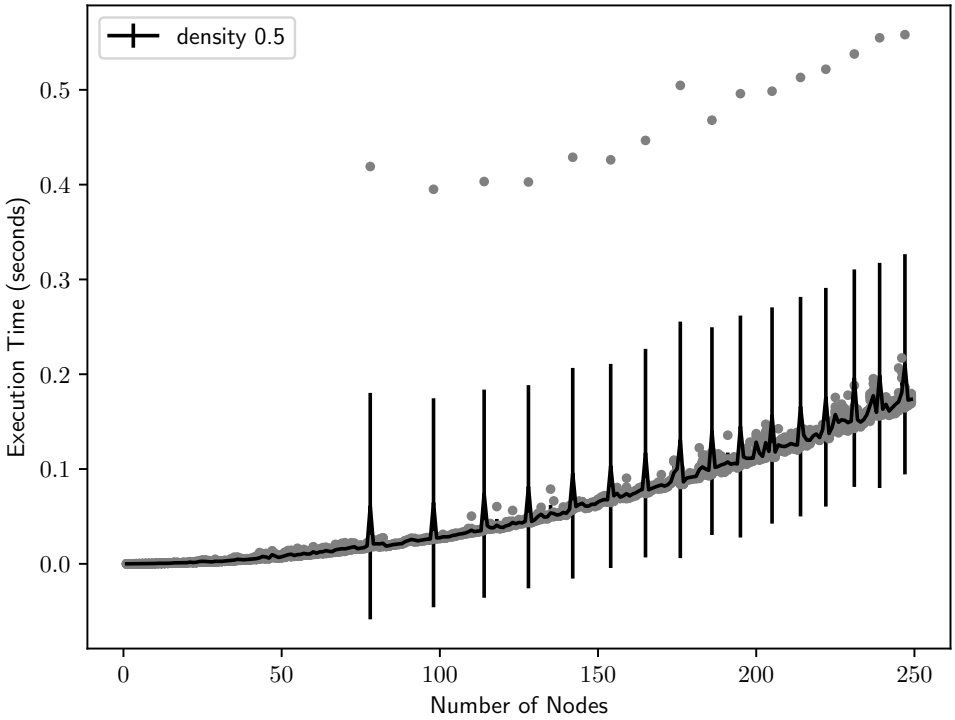

Figure 3.10: Algorithm 4.

Figure 3.11: Spreads of algorithm running times.

need to contend with the problem of commuting graphs. The same is true for F# which has support for units of measure [**?**] and Ada's GNAT compiler [**?**].

3.8 Conclusion

Being able to verify if diagrams commute allows a compiler to make deterministic automatic type conversions and can catch inconsistencies of definition in a program with user defined conversions. In this paper, we have presented verification algorithms that efficiently compute the set of paths that would equal to each other if and only if the diagram would still commute after addition of a new edge.

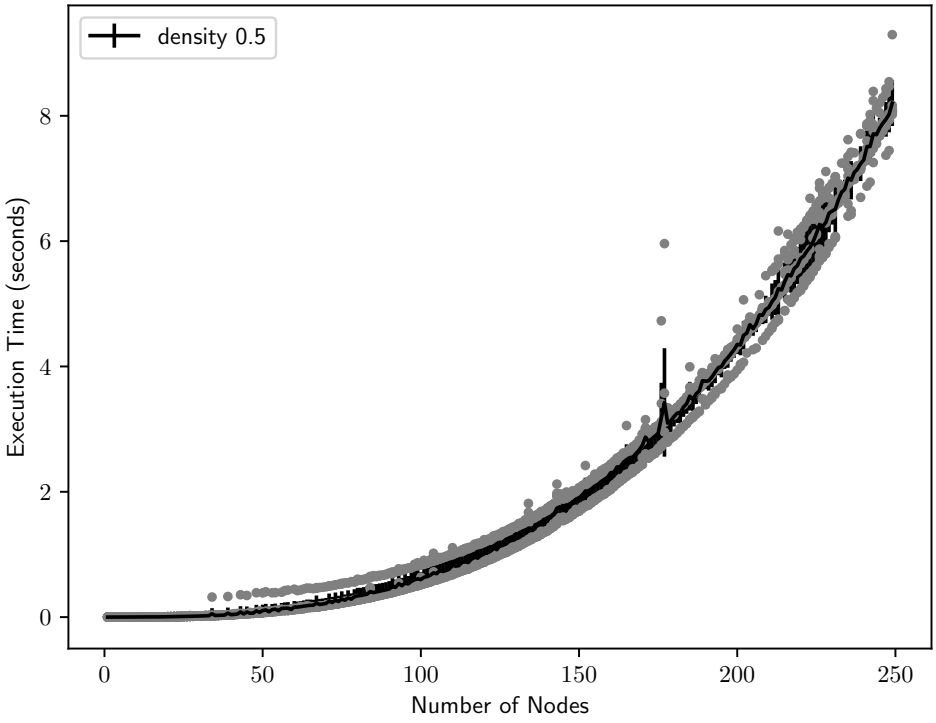

Figure 3.12: Algorithm 3.

Figure 3.13: Spreads of algorithm running times.

Integrating conversion consistency checks into widely used languages such as Scala could provide a lot of value to the program. Since Scala and several other languages provide automatic conversions between types, it seems important to ensure that the choice of which path to take (an apparently arbitrary choice) does not effect the behavior of the program. Having an algorithm to ensure the commutativity of the resulting diagrams can ensure that behavior is correct and help prevent semantic confusion or errors when using such features. More engineering work still remains to implement this feature in a language such as Scala, but this paper provides the algorithms necessary to explore a solution.

CHAPTER 4

CAIMAN: DSL FOR OPTIMIZING HETEROGENEOUS PROGRAM COMMUNICATION (CAIMAN)

4.1 Introduction

GPUs are hard to program I guess.

4.1.1 Separation of Concerns

Our running example throughout this paper will be used to illustrate the performance trade–offs inherent in conditional logic and device synchronization. Specifically, 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;
  }
}
```

In this fairly simple case, we likely expect the compiler to optimize these programs to be about the same. We may quickly become less confident when dealing with more complicated logic, however, often resulting in a need to fight the compiler to perform exactly the operations the programmer intends. This is especially true when we introduce another device into the equation, and the substantial increase in decisions an implementation makes about when to send information between devices and when to start asynchronous operations.

To examine this specific issue in our example, suppose `v2` and `v3` are large arrays. Rather than fiddling with the order of operations in `select_sum`, we might hope to improve performance by moving the `sum` operations on these large arrays to the GPU, with code we hope looks like the following:

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

4.1.2 Combinatoric Explosion

4.2 Background

4.2.1 Submitting to the GPU

4.2.2 Promises

CUDA, SYCL

Halide, Scheduling

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_cpu([i64; N]) -> i64
    extern(gpu) pure sum_gpu([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_cpu(v1) < 0 {
        sum_cpu(v2)
    }
    else {
        sum_cpu(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] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3)
  -> i64 @ node(val.result)
  impls select_sum,... {
    let s1 : i64 @ node(val.sum1) = sum_cpu(v1);
    let cond : bool @ node(val.condition) = s1 < 0;
    let res : @ node(val.result) = if cond {
        let s2 : i64 @ node(val.sum2) = sum_cpu(v2);
```

```
        s2
    }
    else {
        let s3 : i64 @ node(val.sum3) = sum_cpu(v3);
        s3
    };
    res
}
```

Our focus in this rewrite is to expose the (baseline) types used by a Caiman implementation. Each variable in an implementation has 2 components: an raw datatype and three specification types (we are only showing the value specification type for now for simplicity). `s1`, for example, has a raw datatype of `i64`, and a (value) specification type of `node(val.sum1)`.

The part of the type `node(val.sum1)` has three pieces, `node`, `val`, and `sum1`, with the following meanings:

- `node` states that the given type is a node in some specification function (as opposed to an `input` or an `output` to that specification)
- `val` states that the given type is a part of the value specification this function is implementing, `select_sum` in this case
- `sum1` states that we are working with the specific node named `sum1`.

Crucially, this is part of the type of the implementation variable as much as the datatype `i64`, though the specification type can be erased when compiling Caiman code. In other words, if we instead wrote the (incorrect) annotation:

```
let s1 : i64 @ node(val.sum2) = sum(v1);
```

Then our code would fail to compile. Interestingly, this line alone would be enough to fail to compile, as we defined `val.sum2=sum(v2)`, and since our implementation variable `v1` has type `input(val.v1)`, the types of `v2` and `v1` don't "match up".

An important observation is the type of `res`, which is derived from the `if-else` condition logic being applied. In essence, we consider control flow in Caiman's implementation language to have a return type, which can only be a `unit` if the associated specification type is a `unit`. We can show that this particular conditional logic works out, because the true case returns a result associated with the value type `v2`, while the false case returns a result associated with the value type `v3`. If we swapped the logic to instead produce `v3` in the true case and `v2` in the false case, we would have a type error.

With our types in hand, we can now rewrite our original code to reorder the sum operations to before the conditional logic, as we originally desired. The (type-inferred) code in Caiman can be simply written as follows:

```
fn select_sum_impl_2(v1, v2, v3)
  -> i64 impls select_sum,... {
    let s2 = sum_cpu(v2);
    let s3 = sum_cpu(v2);
    if sum_cpu(v1) < 0 {
        s2
    }
    else {
        s3
    }
}
```

A visibly typed version of this code can be found in the appendix. This implementation still typechecks and so maintains the value semantics of the `select_sum` specification – a proof of this claim will be given in 4.4. Importantly, by providing this function definition, we now have two implementations in the `select_sum` function equivalence class – these can be called elsewhere in Caiman by name with `select_sum_impl` or `select_sum_impl_2`.

We can now address the notion of actually using function equivalence in our implementation. Unfortunately, we are not able to yet write our call to `sum_gpu` (as alluded to in 4.3.1), since we will first need to specify more details about synchronizing to another

device (the GPU), and we have so far assumed that the CPU is our local host and so does not need synchronization.

To avoid introducing unnecessary complications to our straightforward equivalence relation, we will instead introduce another implementation of sum on the CPU, extending our equivalence class of `sum`:

```
feq sum {
    extern(cpu) pure sum_cpu_1([i64; N]) -> i64
    extern(cpu) pure sum_cpu_2([i64; N]) -> i64
    extern(gpu) pure sum_gpu([i64; N]) -> i64
}
```

This example is clearly synthetic and meant to be illustrative, but even in this case we could imagine a second definition of `sum_cpu`: specialized to be multi-threaded while our first definition is single-threaded.

We can freely use either definition within our `select_sum` function; for example, we could now write the following code, and the types of each value are identical to what we had before:

```
fn select_sum_impl_3(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    if sum_cpu_2(v1) < 0 {
        sum_cpu_1(v2)
    }
    else {
        sum_cpu_2(v3)
    }
}
```

This mechanism to freely interchange function definitions is core to Caiman's goal of separating performance decisions and specification, as discussed in Section 4.1. Note that here we have introduced yet another definition to the `select_sum` function equivalence class to explore swapping out specific definitions. In this way, we can now avoid the usual combinatorial explosion of logic being checked by relying on the value specification to

maintain static consistency.

It is important now to address the limitations of Caiman's function equivalence classes, since they are designed to be very simple (and we hope transparent). Equivalence classes only apply to exactly function calls, and must either be filled by external functions (which are not checked by Caiman), or by implementations in Caiman.

This means that basic properties like arithmetic or logical equivalence are not reasoned about by Caiman unless explicitly declared. Concretely, for example, we consider `1+1` to be a distinct value from the constant `2`. We will examine potential extensions to Caiman's minimal equivalence system when we discuss future work in Section 4.8.1.

4.3.3 Timeline and Spatial Specifications

We have thus far focused on a vertical slice through Caiman with the value specification language. We can now take the intuition and ideas from this explanation to work through Caiman's other two specification languages.

It is worth noting up–front that Caiman's particular choice of specification languages is not rooted in any sort of experimentation, but instead an intuition of what is needed for the examples we care about – you could extend the ideas presented here to write an entirely different set of specification languages. Detailing this more abstract notion of what a specification language is, however, is out of scope of this write–up, and will be left to another, more theoretical paper.

We capture the intent of the value specification language in Caiman as reasoning about the data produced and used by our computation. When we are working with another device (such as a GPU), however, it is also important to be precise about what threading

and memory resources we interact with. More specifically, we would like to be able to break apart synchronization and memory resources across control flow, in much the same way as the value language allows us to (safely) reuse data and decompose computation on that data into carefully designed pieces.

To achieve this goal, we also introduce the *timeline* and *spatial* specification languages to Caiman. These languages follow the syntax and declarative approach used by the value specification, but with operations intended to capture the intent of synchronization and memory primitives used when scheduling GPU operations.

A Caiman synchronization mirrors that of the GLSL and WGPU submission processes, described in Section 4.2.1. This process consists of exactly 3 events, described as a host managing the devices A and B, where device A is providing the data to run and device B is providing the computation and result:

1. The host requests an *encoding* location from device B, which is then given to device A.
2. After device A has finished encoding data, the host *submit*s that device B can begin computation.
3. Once device B has finished the computation, the host allows device A to *synchronize* and access the written data.

Caiman employs classic promise semantics for this model, described in 4.2.2. For the timeline specification, then, we introduce an `Event` type, which informally describes a point in time. Note that we implicitly also introduce subtypes for each step of this process to keep track of the logical flow, where each requirement is maintained structurally as being in dependency order. We also introduce the following three operations to match the stages described:

```
// Given an Event, starts an encoding on device B,
//   also produces a 'local' event associated with device A
//   along with a 'remote' event associated with device B
local, remote :- encode_event(e)
// Given a 'remote' event, begins a submission on device B
//   produces a 'submission' event
sub :- submission_event(remote)
// Given a 'submission' and 'local event,
//  produces a 'synchronization' event on device A
snc :- synchronization_event(local, sub)
```

We will explore a more detailed example of using the timeline language shortly in Section 4.3.4. First, however, we will summarize the (relatively simple) spatial specification language.

The spatial language is used primarily to specify memory behaviors to help with guarantees related to implemented loops or recursion. To this end, the spatial language provides a `BufferSpace` type, which defines a cluster of memory, along with exactly one operation, used to divide up this buffer space:

```
// splits up a given buffer space into n>=1 evenly sized pieces:
separate_buffer_space(buff, n);
// for example, we can split a buffer in half:
buff1, buff2 :- separate_buffer_space(buff, 2);
```

We will not being working through any examples that define a non-trivial spatial specification, but a recursive example can be found in Appendix [TODO]. xxx

Having defined our specification functions, we can now fully state that an implementation in Caiman must be associated with exactly one of each kind of specification. A given implementation need not implement the entire specification of each, so long as the input and output types of that implementation are correct as stated. Note that this means that a call into a particular Caiman implementation of a function equivalence class may result in a series of calls (some of which may be also used by other Caiman implementations of that specification).

Further, each specification could be implemented by any number of Caiman implementation functions. Additionally, each specification function has its own equivalence class, which refers to any implementation of that specification (it is rare to use a function equivalence class other than a value specification or for external functions).

Note that a specification can always be trivial (simply taking in a single input and returning it), and so we can syntactically elide a specification in both an implementation header and a variable type. In these cases, we assume that the type of the implementation variable associated with the missing specification function refers to the trivial specification.

4.3.4 Select Sum on the GPU

With our timeline and spatial specification functions more carefully defined, we can now construct an implementation of `select_sum` that calls into the GPU to calculate either `sum(v2)` or `sum(v3)` (there are many other variations we can write, but we will start with this approach). In Rust-like psuedocode, what we are looking to implement resembles the following logic:

```
if sum_cpu(v1) < 0 {
    sum_gpu(v2)
} else {
    sum_gpu(v2)
}
```

First, we reiterate our value specification for ease of readability:

```
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
}
```

Second, we provide a timeline specification for a *single* synchronization (since we only

call the gpu once in each branch, we need only to synchronize once). As a result, our specification will simply lay out our four stages in sequence:

```
tmln single_sync(e : Event) -> Event {
    local, remote :- encoding_event(e)
    sub :- submission_event(remote)
    sync :- synchronization_event(local)
    returns sync
}
```

Third, we provide a (trivial) spatial specification:

```
sptl trivial_spatial(b : BufferSpace) -> BufferSpace {
    returns b;
}
```

Now we can provide an exact implementation for this entire function. We will write this implementation without explicit value types, though an explicitly typed version can be found in Appendix [TODO]: xxx

```
fn select_sum_impl_gpu(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,single_sync,trivial_spatial {
    if sum_cpu(v1) < 0 {
        let enc = encode-begin @ node(tmln.(local, remote)) { v2 } gpu;
        // copy the data from v2 into a gpu variable `v2_gpu`
        encode enc.copy[v2_gpu <- v2];
        // schedule the gpu to write the result
        //   of sum_gpu(v2) to a variable named s2_gpu
        encode enc.call[s2_gpu <- sum_gpu(v2) @ val.sum2];

        // submit the calculation to the gpu
        let fence = submit @ tmln.sub enc

        // await the result
        let gpu_data = await @ tmln.sync

        // return s2_gpu from the bundled result
        gpu_data.s2_gpu
    }
    else {
        // we apply similar logic here
        let enc = encode-begin @ node(tmln.(local, remote)) { v3 } gpu;
        encode enc.copy[v3_gpu <- v3];

        encode enc.call[s3_gpu <- sum_gpu(v3) @ val.sum3];
        let fence = submit @ tmln.sub enc
        let gpu_data = await @ tmln.sync

        gpu_data.s3_gpu
        sum_cpu(v3)
    }
}
```

This implementation more-or-less mirrors the first approach we examined, namely calculating the sum of `v2` and `v3` inside of the condition. Now that we have the logic for using the GPU, this approach of first calculating both before the condition will (hopefully) be more immediately appealing.

Interestingly, in this case, we can apply both operations sequentially on the GPU with only a single synchronization, but this requires an unsatisfying black–box solution where we write a `sum_2_arrays` type function. Alternatively, we could write a new timeline specification which allows for two synchronizations, and if our goal were to interleave our encodings and synchronizations, then we would write a new specification.

However, if we are comfortable synchronizing twice, Caiman does provide a clean solution in the form of breaking up our implementation to use the timeline specification twice, without needing to modify any of our specifications:

```
fn select_sum_impl_gpu_2(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.result
  impls select_sum,single_sync,trivial_spatial {
    let enc = encode-begin @ node(tmln.(local, remote)) { v2 } gpu;
    encode enc.copy[v2_gpu <- v2];
    encode enc.call[s2_gpu <- sum_gpu(v2) @ val.sum2];
    let fence = submit @ tmln.sub enc
    let gpu_data = await @ tmln.sync
    let s2 = gpu_data.s2_gpu;

    select_sum_impl_gpu_2_inter()
}
fn select_sum_impl_gpu_2_inter(
  v1: [i64; N] @ val.v1,
  s2: i64 @ val.sum2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.result
  impls select_sum,single_sync,trivial_spatial {
    // similar logic to compute s3
    select_sum_impl_gpu_2_ret(v1, s2, s3)
}
fn select_sum_impl_gpu_2_ret(
  v1: [i64; N],
  s2: i64 @ val.sum2,
  s3: i64 @ val.sum2)
  -> i64 @ val.result
```

```
  impls select_sum,trivial_timeline,trivial_spatial {
    if sum_cpu(v1) < 0 {
        s2
    }
    else {
        s3
    }
}
```

This idea of breaking up a function to implement only part of a specification is crucial to using Caiman, as code written in this way can represent unfinished and can avoid duplicating computation. The reason this code is able to typecheck is precisely because of our explicit annotations on the arguments and return type of each function – otherwise, the program analysis needed often becomes intractable. Since Caiman can prove that the type boundaries of each function resolve, we need not worry about the potential complexity introduced between function calls.

Another important observation is that we can use a similar idea to break apart encoding and synchroniziation, allowing a programmer to first encode an operation, then continue doing work on the device, and finally synchronize. Note that Caiman's restrictions are such that we could not encode an operation and then never synchronize that operation (at least, within a Caiman program where we manipulate the timeline between pieces of control flow), since the types associated with the timeline and spatial specifications must match between each possible branch we could take.

Finally, it is worth acknowledging that this Caiman implementation has become quite complicated even for this simple operation. These implementations are, by design, rather detailed about each operation that is needed to synchronize information with the GPU (or just any other device). The intention behind this approach is to allow the user of Caiman to be as precise as possible with defining program implementation. That being said, having implementations be this dense seemingly defeats the purpose of the guarantees Caiman can make – if we require the programmer to both write careful specifications 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

implement those specifications to exacting detail, the effort to maintain this system can quickly get out of hand.

We will address this concern with a core piece of the Caiman design when we discuss in Section 4.5. First, however, we need to take a brief detour into examining a formal model for Caiman's type system, to provide formal backing and proof behind the core guarantee of Caiman's implementations actually being checked against the specification.

4.4 Formal Model

[I include alloc, copy, and ref here because it feels like I need to be clear about the imperative nature of Caiman's implementation, but I actually think these end up being pretty non–interesting. I feel like the rest of the formalism is enough in xxx

retrospect. —DG]

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 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 4. *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 4.3.1), 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.

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

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

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

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

Figure 4.3: Hatchling Specification Typing Judgment

- $\Psi : \mathbf{S} \mapsto \mathbf{T} \bigtimes (\mathbf{C} \bigtimes (\mathbf{S} \bigtimes \mathbf{S} \bigtimes \mathbf{S}))$, 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: If I cut the alloc and ref, then this is essentially just done I think —DG] xxx

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

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

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

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

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

Figure 4.4: Hatchling Implementation Typing Judgment

4.4.2 Operational Semantics

[I don't think I actually need operational semantics, since they sort of "fall out" xxx from the simplified model I decided to go with —DG]

4.5 Explication

4.5.1 Using Explication

4.5.2 Caiman IR

4.5.3 Core Algorithm

4.5.4 Controlling the Explicator

4.6 Caiman Engineering

4.6.1 WebGPU Target and Codegen

4.6.2 Stages of Compilation

4.7 Results

4.7.1 Timing

4.7.2 Explication

4.8 Conclusion

4.8.1 Future Work

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/`.