LANGUAGE DESIGNS 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 DESIGNS FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

Dietrich Geisler, Ph.D.

Cornell University 2024

There has been growing demand for graphical image rendering in the past several decades. This demand has arisen primarily in video games, but also from fields as broad as film, art, architecture, and scientific simulation. A major challenge with expanding use of rendering, however, is that graphics programming is difficult, requiring significant field expertise when abstractions break down.

In this dissertation, we will examine how we may be able to design programming languages to ameliorate some of these challenges. Our goal will be to examine two specific difficulties in graphics programming reasoning: geometric correctness and performance in heterogeneous device communication.

In the first chunk of this dissertation, we will examine Gator, a language which provides type-level reasoning for a class of bugs we describe as “geometry bugs", as well as a lightweight mechanism to reason about operations on geometry. In the second chunk of this dissertation, we will discuss Caiman, a language which typechecks heterogeneous implementations against a fixed specification. We will also examine how Caiman’s type-level restrictions allow for separating performance and correctness, as well as providing a mechanism for restricted synthesis of heterogeneous programs.

BIOGRAPHICAL SKETCH

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

For my siblings: Conrad, Emil, and Claire

ACKNOWLEDGEMENTS

This PhD would not have been possible without the support of mentors, colleagues, friends, and family: First and foremost, I would like to thank my advisor, Adrian Sampson. Without his mentorship, encouragement, and support, I would not be here today. Beyond research, Adrian has helped me find life direction, and how much his guidance means to me cannot be overstated.

The second chapter of this dissertation was written in collaboration with Irene Yoon, Aditi Kabra, Horace He, and Yinnon Sanders. Irene in particular provided a huge amount of work and produced some excellent design for building Linguine, the excellently–named predecessor to the Gator language.

Additionally, Evan Adler, Kimberly Baum, Ben Gillott, Henry Liu, and Palini Ramnarayan all provided important contributions to Gator systems and tooling, and worked on excellent projects that helped refine our understanding of Gator usability.

The third chapter of this dissertation was written in direct collaboration with Aditi Kabra, who also detailed much of the theory of Gator.

While I do not have specific writing to credit, thank you to Haoxuan Chen, Meredith Hu, Paul Joo, and David Siher for their discussions and contributions to this body of research. Their work was invaluable for helping refine our research direction and explore topics I wish we could have expanded further.

The fourth chapter of this dissertation, Caiman, could only have been written in collaboration with Oliver Daids, who both completely changed how we approached the problem presented by Caiman *and* worked out details for the complex theory and interlocking systems needed to realize the Caiman compiler. Stephen Verderame provided invaluable contributions in expanding Caiman with the frontend language presented in the paper, and the explanations provided here could not be given without his hard work.

Caiman could also not be what it is today without the work provided by Mia Daniels,

Mateo Guynn, and Patrick LaFontaine, all of whom contributed systems engineering and design to the sprawling Gator project.

I would like to thank Chris Batten, Nate Foster, Steve Marschner, José Martínez, and Walker White for their mentorship and support in producing this research and giving me both motivation and direction. This dissertation would not be the same without such diverse ideas and discussions.

I could not have finished this work without the support of my close friends: Gemma Clark, Jarem Kilby, Elliot Lee, Haobin Ni, Oliver Richardson, and Michael Roberts. I keep this paragraph short lest it take up the rest of what was supposed to be a dissertation, but I could not have done this work without your support. Thank you.

Thank you to my siblings, Conrad, Emil, and Claire for your love and support. Thank you to my Dad, who has stayed a fixture of my life despite geographic distance.

Finally, I must end by thanking my Mom, who has somehow raised and supported both me and my two brothers as a single parent for all these years. I truly could not have done this work without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer graphics has long been a core field of study within computer science. In the past decade alone, computer graphics has seen application in video games, animated film, scientific simulation, data visualization, and medical devices. The term computer graphics itself has become so broad as to be fuzzy; we refer here specifically to the study of modeling and rendering snapshots of some simulated space onto a screen or image.

Despite the number of applications and domains that make use of computer graphics, maintaining or using software systems for rendering (such as a *rendering engine*) can be extremely difficult and time–consuming. Manipulating a rendering engine can require specialized learning in topics as diverse as light physics, geometry, visual design, and material science. Additionally, the history of these specialized topics can often find themselves at odds with the practical realities of building a performant computer system, resulting in complex engineering constraints and implicit rules for manipulating code.

As a consequence of this complexity, computer graphics has many domain–specific challenges that have been solved through sheer engineering prowess on the tools available. For instance, game engines are frequently specialized to C++ engineering, relying on macros to control performance characteristics and providing highly specialized program behavior. Similarly GPUs come equipped with a rendering pipeline originally meant for the usual case of computer graphics, but as programmer specialization has outpaced hardware design, the GPU rendering pipeline has been taken apart and pieced back together to squeeze more performance or a specific behavior out of this hardware.

These challenges in graphics programming have real cost: large–scale rendering engines can be difficult to update for new technologies (such as Unreal Engine's slow push towards native support for raytracing), non–experts can be forced to rely on black–box implementations without any realistic mechanism to customize these implementations, and performance can be left on the table in critical applications.

The sheer breadth and complexity of these computer graphics and rendering systems, however, poses a unique opportunity for programming language designers. Improvements in languages for graphics specifically could provide mechanisms that keep up with graphics programmer needs faster than hardware design, and can expose domain–specific design challenges with interesting consequences for language research as a whole.

Despite there being both potential and real need for graphics–specific programming language design, this focus of study has remained largely untapped. Notable efforts in this direction include languages for static and dynamic reasoning about the rendering pipeline and specialized languages for automatic and symbolic differentiation in geometric spaces. There are, however, few other high–profile efforts, despite anecdotally there remaining many potentially interesting problems (which will be further discussed in Section **??**).

1.0.1 Summary of Work

In this dissertation, we present work that both identifies and provides language–level solutions for two specific challenges in graphics programming: geometric reasoning and performance exploration. In both cases, we identify properties of graphics programs which are either stated informally or otherwise known to the programmer, but are not communicated to the typechecker and compiler. Without this necessary context for the intended program semantics, the programmer loses the benefits of static typechecking and compiler optimizations, and may be forced to introduce the various C++ hacks described earlier.

Concretely, we describe three pieces of work:

- Gator, a language for providing semantics and typechecking for graphics–style

geometry

- A paper on commutative diagram verification needed to solve a technical challenge within Gator
- Caiman, a language for providing type–level support for heterogeneous performance exploration

Both Gator and the commutative diagram paper have been previously published.

Chapter 2 explores the relationship between the geometry of a scene being rendered and the code used to calculate properties of that scene through the lens of the Gator language. By identifying and naming three commonly–needed pieces of geometric information, Gator is able to provide geometry-aware types and semantics for several core graphics algorithms. We also show how introducing these types enables the Gator compiler to safely synthesize light–weight geometric transformations. The guarantees Gator aims to provide, however, resulted in needing a solution for online verification of commutative diagrams, a technical layer described further in Chapter 3.

Chapter 4 changes our focus to the performance concerns of graphics programmers, and specifically the narrow problem of inter–device communication, or heterogeneous programming. We introduce a language, Caiman, which provides a type system and compiler implementation for separating semantic and performance concerns in heterogeneous settings. We additionally develop and implement an algorithm for synthesizing these (otherwise complex) transformations in a type–directed and decomposable way, allowing a programmer to explore performance characteristics of a compiled program while maintaining control over the details of that program.

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 chapter 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. Figures 2.1 and 2.1 shows an example: a coordinate system handling bug yields incorrect visual output that would be difficult to catch with testing.

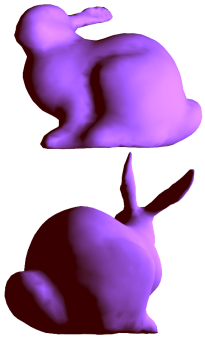


Figure 2.1: Correct implementation.

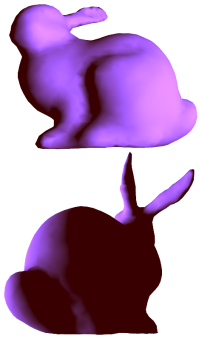


Figure 2.2: Incorrect implementation.

Figure 2.3: Objects rendered with an implementation of the diffuse component of Phong lighting [35], 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.

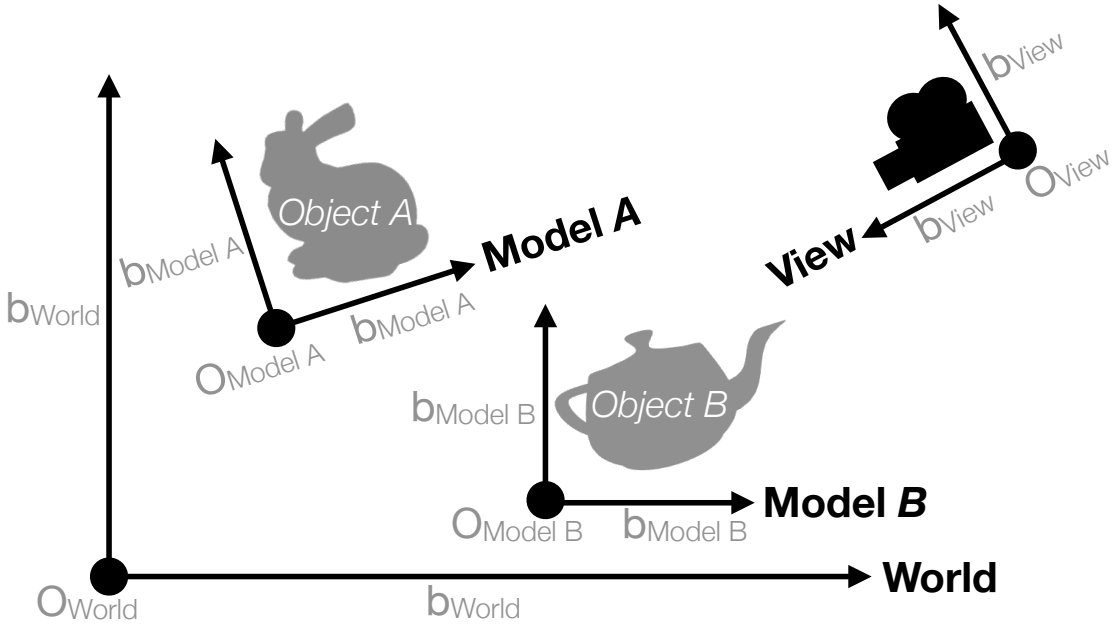


Figure 2.4: Coordinate systems in graphics code. **Model A**, **Model B**, **World**, and **View** are coordinate systems. A coordinate system is defined by its basis vectors b and origin O. The **View** represents the perspective of a simulated camera.

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 [41] 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 chapter 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 chapter 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 [33] 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 [23], 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 a (straightforward) property of producing a well–typed term. 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 chapter'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 [35].[1] We assume some basic linear algebra concepts but no background in graphics or rendering.

[1]Appendix A.1 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):

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

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*

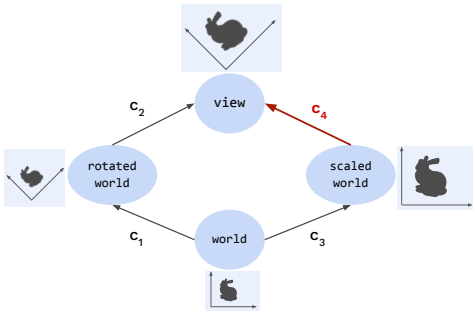


Figure 2.5: A *transformation graph* with provided transformations. The highlighted edge represents a newly added transformation function, which must be unique and agree with the existing paths on the graph.

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 labeled 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.

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 *coherent* (a term borrowed from language used for type coercions [5]): 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 coherence. However, coherence 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 coherence with non-invertible functions and result in an ambiguous transformation graph for `in` expressions.

Our construction of canonical functions and automatic transformations is similar to constructions provided by C# and C++'s type coercion, which are themselves examples of general type-theoretic coercion functions. Indeed, our intuition related to coherence can be summarized as what is (informally) needed to match the property needed for coherence given in [5]: "translations of any two derivations of the same typing judgement are equated in the target calculus". The slightly different practicalities used by our `in` expression implementation will be discussed briefly in Section 2.8.

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

Figure 2.6: Core Gator syntax.

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 provide details of typechecking, without proving type soundness. 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 of the Gator type system would be interesting future work but is out of scope for this chapter. Concretely, to show Gator type soundness, we would need to extend our formalism with a precise definition of a geometry bug and how our type system would formally rule out such a bug.

We define a high-level core semantics for Gator that includes its user-defined types.

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 partial order with $\top_p$ and $\bot_p$ 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.

Gator requires a partial order, along with a $\top_p$ and $\bot_p$, for each primitive type; custom types on each partial order introduce 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

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

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

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

Figure 2.7: Typing Judgment

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 $\top_p$.

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.

2.6 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

(a) Texture shader

(b) Reflection Shader

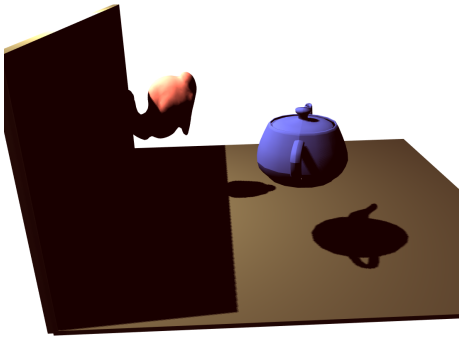
Figure 2.8: Example shaders implemented in Gator

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).

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 [16]. 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.



(a) Shadowmap shader

(b) Microfacet Shader

Figure 2.9: Example shaders implemented in Gator

- *Shadow map:* Simulate shadows for moving objects by computing a projection.
- *Microfacet:* Texture model for simulating roughness on a surface.
- *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. Figures 2.8 and 2.9 depict 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

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 2.8, 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 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);
}
```

[3]URL omitted for anonymous review

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 2.9, 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 2.8). 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.

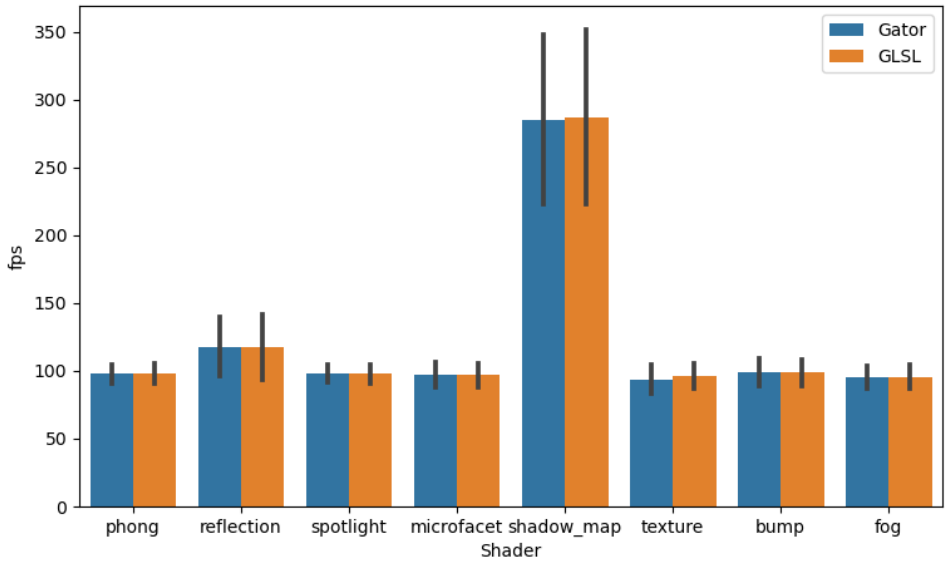


Figure 2.10: The mean frames per second (fps) for each shader for both the baseline (GLSL) and Gator code. Error bars show the standard deviation.

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

Performance Results

Figure 2.10 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

	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 ($p < 0.05$).

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 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 [40], 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 [33] 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 [23] and Direct3D's HLSL [28]. Research on graphics-oriented languages for manipulating vectors dates at least to Hanrahan and Lawson's original *shading language* [12]. Recent research on improving these shading languages has focused on modularity and interactions between pipeline stages: Spark [8] encourages modular composition of shaders; Spire [13] facilitates rapid experimentation with implementation choices; and Braid [38] 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 [10] introduces semantics to reason about relative object positions and λCAD [31] 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 [42]. A 2017 enumeration of

programming problems in graphics [37] 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 [9], dimension types [19], and type systems for tracking physical units [20]. 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 comes down primarily to problem domain specifics, focusing on geometric implications related to graphics programming. Tying Gator's type system approach to existing program implementations could provide an interesting (and useful!) direction for future work.

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 [18]. 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.

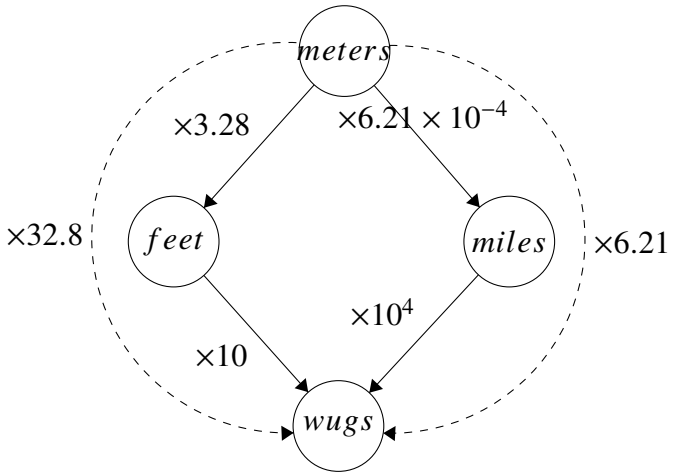


Figure 3.2: In this sample program, the user implicitly defines two ways to cast variable a from meters to the new unit wugs. The definitions are different, and a compiler performing implicit conversion would not know which to choose.

This chapter 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 [30] 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* [11] 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 [30].

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 chapter 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 [30].

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 [17], 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).

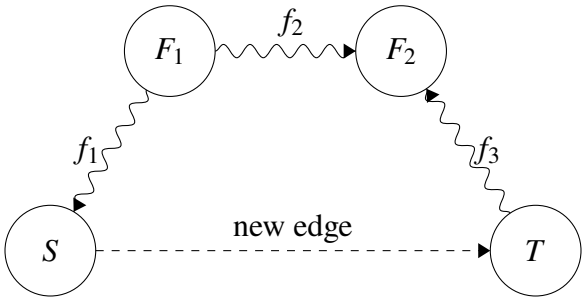


Figure 3.3: Two flip tolerant path.

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 1. *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 [30] 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 $r_i \in \sigma$*(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[30]. 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 2 (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.

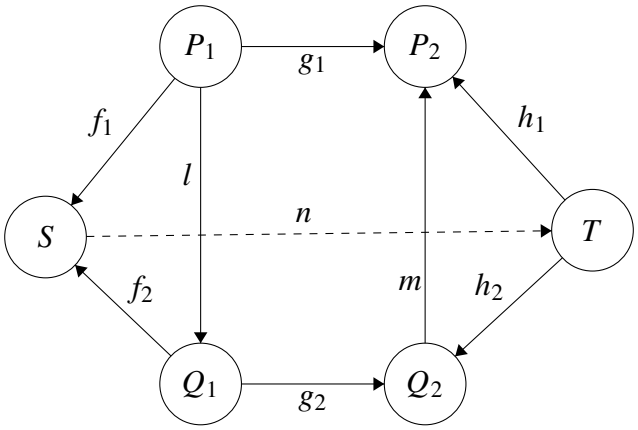


Figure 3.4: Reduction rule. Each arrow represents a path, where n is the new edge being added. While Algorithm 3 returns two pairs for verification, one from P_1 to P_2 and the other from Q_1 to Q_2, it actually suffices to just check a pair from Q_1 to Q_2 as demonstrated in theorem 2.

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 2. *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 2, 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 [11]. 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 2 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

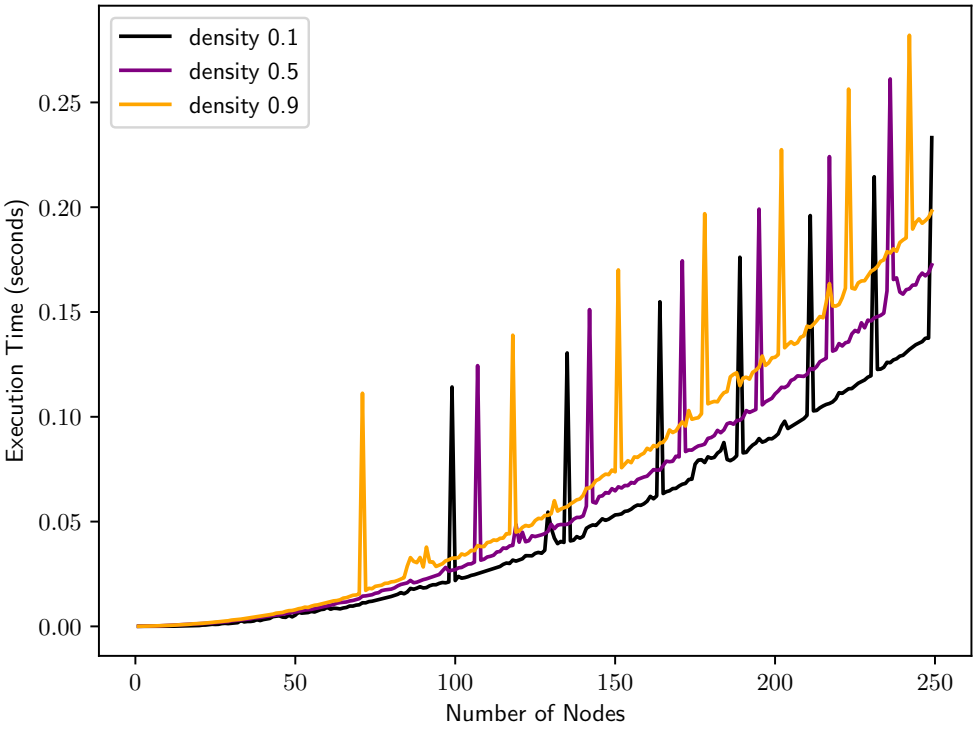


Figure 3.5: Algorithm 4.

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.

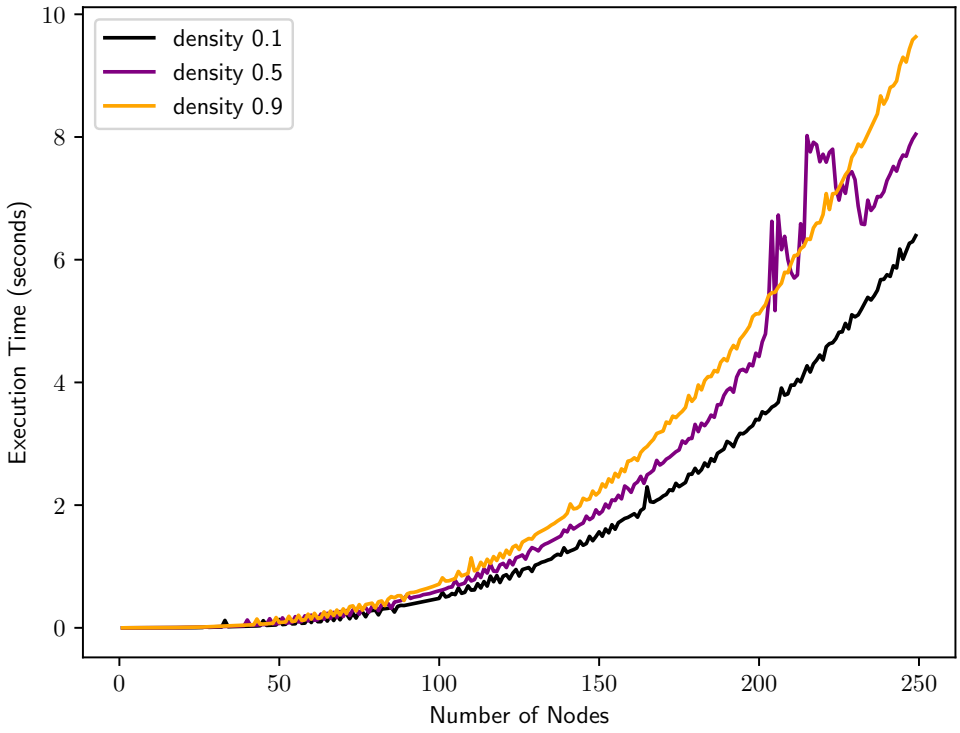


Figure 3.6: Algorithm 3.

3.6.2 Scaling of Time with Input Size

Figure **??** shows that the algorithms' time scales with size, as expected. 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. 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.

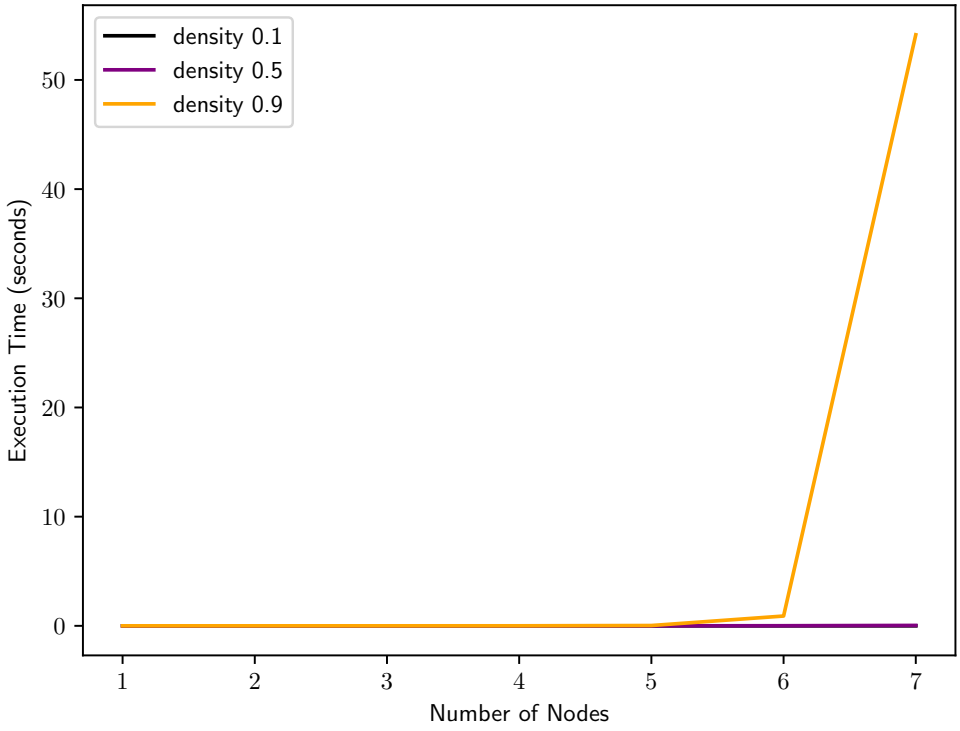


Figure 3.7: Naive baseline.

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.

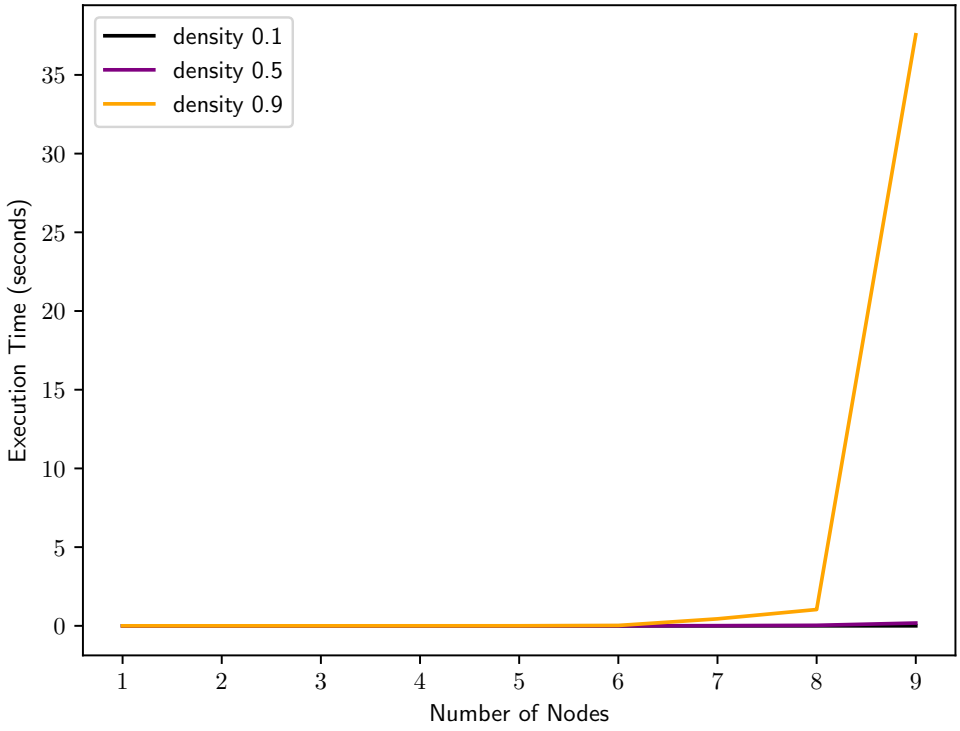


Figure 3.8: Two flip tolerant baseline.

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.

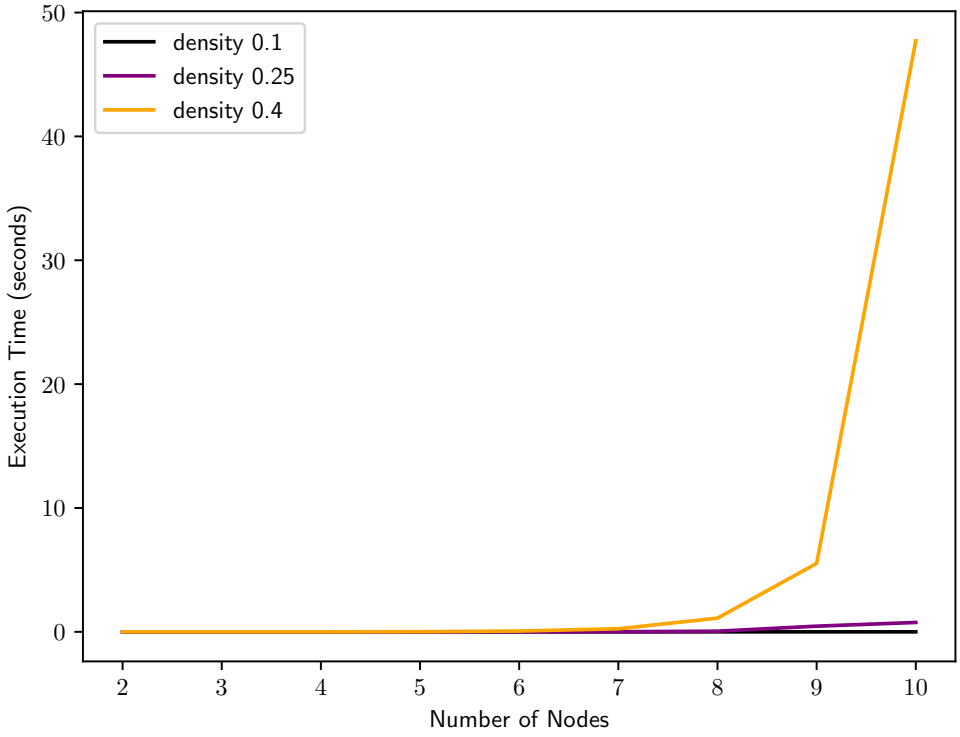


Figure 3.9: Batch algorithm baseline.

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 [30]. 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 [11], 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 [1] 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 but if extended to do so, would need to contend with the problem of commuting

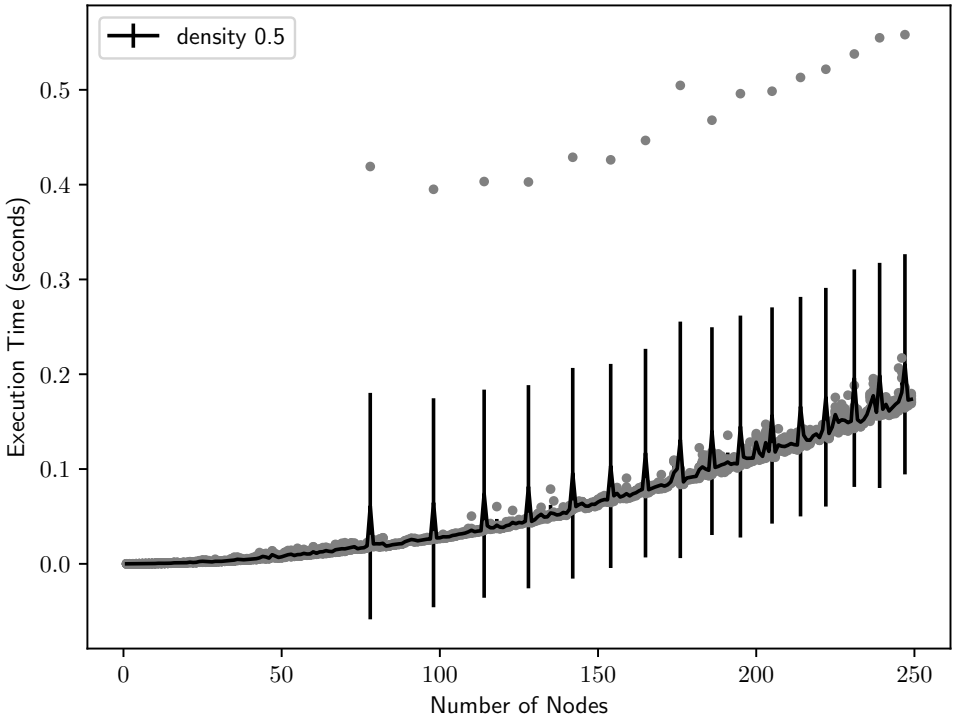


Figure 3.10: Algorithm 4.

Figure 3.11: Spreads of algorithm running times.

graphs. The same is true for F# which has support for units of measure [18] and Ada's GNAT compiler [39].

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 chapter, 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.

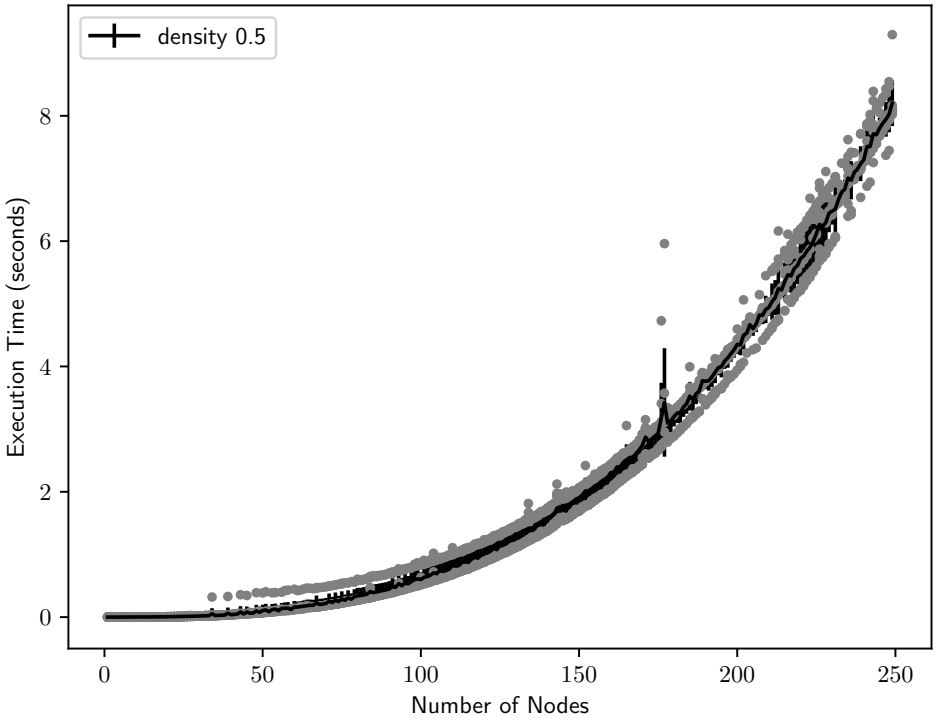


Figure 3.12: Algorithm 3.

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 chapter provides the algorithms necessary to explore a solution.

CHAPTER 4

CAIMAN: DSL FOR OPTIMIZING HETEROGENEOUS PROGRAM COMMUNICATION (CAIMAN)

4.1 Introduction

Accelerators have become of great interest to the computing community in the last few years [34] [7]. For specific applications or computations, specialized accelerators can provide significant performance improvements. Using an accelerator, however, can require significant engineering effort and programmer expertise.

A major reason for the complexity of accelerator programming comes from the significant performance concerns that often appear when designing accelerator code. Intuitively, when an engineer finds that an accelerator is needed for some task, performance is likely a consideration. Since accelerators are often multithreaded, with specific requirements on data layouts or algorithm design. These constraints and performance needs have led to years of research and industrial work to optimize accelerator *kernels* (accelerator–only units of computation). However, there is a significant cost in the communication between devices, and this cost can be specifically painful when communication occurs as blocking the logic of a program.

This cost has been explored in several works in the form of automatic compiler optimizations [43] and code generation [25], which rely on either a compiler or learning model to generate communications or kernels based on some heuristic. Despite these advances, anecdotally programmers are still optimizing performance–critical communication by hand. This is (at least partially) due to the need for experimentation and profiling in performance–critical code, where a given solution may not perform optimally in all

heterogeneous arrangements and even within a given codebase.

We argue, without any meaningful way to prove or refute this claim, that heterogeneous programming is diverse enough and performance–critical enough, that such hand–optimized solutions will continue to be important for kernel and shader engineers. As a result, we propose a solution that raises the level of abstraction for performance engineering, but engineered so that the layer of abstraction being constructed is both transparent and decomposable. We start by making more concrete the specific challenges we have observed while hand–optimizing kernels, specifically separating performance and correctness, and managing a combinatorial explosion of kernel interactions.

4.1.1 Performance Experimentation

Our running example throughout this chapter 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 : [i32; N1], v2 : [i32; N2], v3 : [i32; N3]) -> i32 {
  if sum(v1) < 0 {
    sum(v2)
  }
  else {
    sum(v3)
  }
}
```

Note that, in Rust, an expression without a `;` is equivalent to adding a `return`, so this code can be read as either returning the sum of `v2` or `v3`.

We might find that, for large arrays, this code is insufficiently performant in our setting. We might hope to be able to take an approach of the following Rust-like psuedocode, where we simply move our expensive `sum` operations onto the GPU:

```
select_sum_2(v1 : [i32; N1], v2 : [i32; N2], v3 : [i32; N3]) -> i32 {
  if (sum_gpu(v1) < 0) {
    sum_gpu(v2)
  }
  else {
    sum_gpu(v3)
  }
}
```

Upon measuring the performance of this solution, however, we may find that this code isn't "fast enough", even if we are using a highly–optimized GPU implementation of `sum`. If we want to improve the performance of this code further, we have several options. We could move this entire code onto the GPU (noting, crucially, that the GPU tends to perform poorly with conditions), but another reasonable approach would be to calculate each `sum` in advance, to avoid bottlenecking the next GPU calculation behind the CPU condition. Concretely, in psuedocode, this implementation may look something like:

```
select_sum_3(v1 : [i32; N1], v2 : [i32; N2], v3 : [i32; N3]) -> i32 {
  sum1 = sum_gpu(v1);
  sum2 = sum_gpu(v2);
  sum3 = sum_gpu(v3);
  if (sum1 < 0) {
    sum2
  }
  else {
    sum3
  }
}
```

To complicate this function even more, however, we may have a case where `v1` is a smaller array than `v2` or `v3`, and so sending `v1` to the GPU is introducing unnecessary overhead in our computation. For this special case, we may need to consider the following function variation (even within our same program):

```
// For N1 << N2,N3
select_sum_4(v1 : [i32; N1], v2 : [i32; N2], v3 : [i32; N3]) -> i32 {
  if (sum(v1) < 0) {
    sum_gpu(v2)
  }
```

```
  else {
    sum_gpu(v3)
  }
}
```

Note that, in this variation, we may find it optimal to revert to the original idea of having `sum_gpu` inside of the condition rather than before, since we are no longer blocking on communication between devices.

It is important to note that every one of these steps we described may depend on measuring performance in a specific setting, with specific array sizes, and even within specific functions. The particular functions we wrote may or may not be optimal – an analysis of our timing each variation of this function can be found in 4.8 (including blindly moving the entire `select_sum` function to the GPU), but we can only show these results for a particular setting. The takeaway, then, should be that a programmer may need to try each of these variations to find a performant solution, and may need to maintain multiple such functions (such as both `select_sum_3` and `select_sum_4`) in the same program, despite these functions doing the same computation!

To make matters worse, we are ignoring even more decisions that can be made between these black–box calls to `sum_gpu`. The decision space here will be expanded on in 4.3.1, which will also expand on why these functions may be much harder to write than what we have shown here. Before we continue, however, we need to explore another problem that arises from maintaining multiple definitions of `select_sum`.

4.1.2 Combinatoric Explosion

We have shown how to end up with several implementations of `select_sum`, where optimal performance may depend on setting-specific experimentation. Another key aspect

to optimizing heterogeneous performance, however, is optimizing for the context of surrounding functions. This is suggested by our reliance on the condition `N1 << N2,N3`, which implies that we ought to be careful about which "version" of `select_sum` to call based on information about the function calling `select_sum`. We can extend this logic the other direction, and find cases where we may prefer one implementation of `sum_gpu` over another for a particular choice of `select_sum`, which we can name `sum_gpu2`.

Concretely, let us define some `complicated_function`, where we have three calls (at various points) to `select_sum`. In–code, we can summarize this assumption as follows:

```
fn complicated_function(...) {
  select_sum(...);
  // ...
  select_sum(...);
  // ...
  select_sum(...);
}
```

We may find, however, that `complicated_function` is optimized for a particular arrangement of `select_sum` implementations; for instance, we might experiment with the following arrangement:

```
fn complicated_function(...) {
  select_sum_1(...);
  // ...
  select_sum_3(...);
  // ...
  select_sum_4(...);
}
```

This sort of experimentation can lead to an explosion of variations to try, where we may find that a given implementation of `sum_gpu` is preferable in a particular implementation of `select_sum` within this particular `complicated_function`, necessitating writing yet another variation of `select_sum`, which may require adjusting our other decisions, and so on and so forth. In this manner, we can quickly end up writing an exponential number of variations of functions, or, practically, write a complex system of macros,

comments, and/or implicit dependencies between code variants, with little in the way to help manage this design overhead.

4.1.3 Caiman Languages

To help address these problems with performance experimentation in heterogeneous programming, we introduce the Caiman language. The core goal of Caiman is to provide a separation between the definition and the implementation of a function, and to allow these function implementations to be used interchangeably in implementation (with some important type–level protections). In doing so, we make the following contributions in this work:

- We provide formal guarantees on multiple implementations for the same function *specification*
- We describe functions that are decomposable, in that a function implementation can be broken apart so that parts can be used by other implementations without duplicating the entire function, all without breaking Caiman's type–level guarantees of matching implementation to definition.
- We describe and implement a system within Caiman to harness type–directed program synthesis to help reduce the overhead of performance experimentation and to allow transparency into compiler optimizations.
- We provide a syntax and implementation for both human–writeable Caiman with the above guarantees, and a more precise Caiman IR for analysis and precise operational control.
- We demonstrate the performance tradeoffs for several synthetic WebGPU kernels, and show that a Caiman implementation of these kernels allows a similar

performance exploration.

4.2 Related Work

Scheduling Languages The philosophy of Caiman has overlap with scheduling languages such as Halide [36], Taco [24], and Exo [15]. The Caiman strategy of separating performance and implementation concerns is inspired by these sorts of scheduling languages, and indeed the direction for Caiman was initially written to be a scheduling language for heterogeneous interactions.

Despite this core inspiration, Caiman navigates the specific challenges of heterogeneous programming differently than the scheduling languages we are aware of. Most notably, Caiman emphasizes design for the combinatoric explosion of kernel design described in 4.1.2 and introduces type requirements for the multi–threading and condition logic present in heterogeneous decision making.

Heterogeneous Languages There have been a variety of heterogeneous programming languages in the last few years. We highlight three well–known languages/APIs for comparison, but acknowledge there are many more projects that we will not specifically address.

CUDA [6] is a computational language for the GPU that allows for compiling C++ code with a GPU program while providing and API “to use C++ as a high–level–language" [32]. A key feature of this approach, however, is to make the interface between CPU and GPU very thin (CUDA code strongly resembles a C++ function). This design makes writing code fairly straightforward, but can make extracting lower–level details for performance difficult.

OpenCL [21] is a widely used heterogeneous language solution, providing a language and API for a variety of heterogeneous settings. OpenCL provides a similar toolset to CUDA, but is designed to be more general–purpose and more controllable than GPU compute, at the expense of being somewhat difficult to program and require domain–specific expertise. SYCL [22] is an API built on top of (but separate to) OpenCL, and is meant to provide some of the higher–level benefits that can be seen with CUDA.

Vulkan [2] is a graphics–specific API for the GPU that is meant to give more control over exact GPU implementation and communication details. As with OpenCL, however, Vulkan can be difficult to use at scale due to the amount of domain–specific knowledge and optimization work needed. Despite this, the recent interest in Vulkan gives an indication of the value that exposing detailed interfaces can have for developing performant systems.

Caiman is designed as an alternative approach to these heterogeneous APIs, with an emphasis on transparency in the boundary between abstractions and performance–critical implementation details. A Caiman implementation could be compiled to any of these targets, or used as an alternative language/API.

4.3 Background

Before discussing the Caiman implementation, we first must examine some practical concerns when dealing with CPU–GPU communication. While Caiman could be used for many heterogeneous systems, our particular implementation is for CPU–GPU, and so we embed some of the terminology and concepts from the GPU in our Caiman discussion to make ideas concrete.

Our implementation of Caiman is specifically targeting WebGPU Shading Lan-

guage [29] (WGSL) through the Rust API WGPU [3]. Specific properties of our implementation are discussed in Section 4.7, but we will need to summarize WebGPU data movement to motivate details of the overall Caiman language design.

Our goal in this section is to provide an overview of how data is managed in WGPU as it relates to Caiman. We will be intentionally eliding many technical details of data movement in WGPU and more general graphics programming that exist in Caiman, but are unnecessary to understanding the Caiman typesystem and language design.

4.3.1 WGPU Data Submission

To move data from the CPU to the GPU, we distinguish four stages of work that need be done:

1. Setup: Request a memory location for writing to on the GPU, along with an `encoder` to manage data copying.
2. Encoding: Using the `encoder`, copy data from the CPU as needed, and set the program on the GPU that will actually be run.
3. Submission: submit that the data movement is finished and the GPU can begin processing work.
4. Await: Once the GPU is finished processing, the output data stored in memory can be safely read.

We name these steps as corresponding to the operations that Caiman will introduce, though similar naming schemes are used by WGPU and other GPU APIs. Each step must

be performed in sequence for a given operation, though data inputs during encoding and outputs after submission can be modified up until a signal is received for the next step.

A flag specifying when the next data transfer step or operation can occur is called a *fence*. Additionally, each of these operations are asynchronous between the CPU and the GPU, as the device waiting for data may be able to continue a previously defined operation until ready to read from a piece of memory. Such a location is called a *future*, indicating that the referred data can be read at some point in the future (if the computation terminates). It is common for futures to be implemented such that a device will *await* the future (hence our use of the Await step) until the data is written to the future and a flag is set, thus synchronizing data movement.

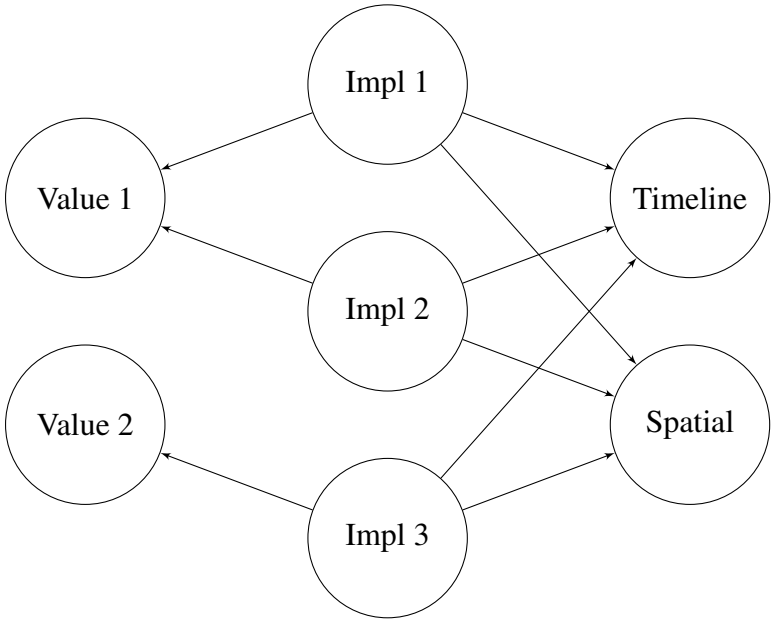
4.4 Practical Caiman

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 4.7.

A Caiman program consists of a series of functions written across several languages: the three *specification* languages and the one *implementation* language. Informally, a function written in the specification languages describe semantic behaviors 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 particular 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 Caiman implementation *must* implement one of each kind of specification, though specifications may be used by or broken up across multiple implementation functions. The relationship between implementation and specification functions is illustrated in Figure 4.4



To help parse this image, observe that every implementation function (`Impl`) has three outgoing arrows: one to each kind of specification language. Additionally, note that any number of implementations may share a single specification language, but that a given implementation may only implement one of each specification. A detail to highlight that is not easy to visually show, however, is that a given implementation need not implement every operation within a given specification; how this works practically we will see in more detail throughout this section.

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 4.5.

4.4.1 Value Specification

Caiman specifications are written as functions with a Rust-like header and bodies of pure expressions. We emphasize purity here to mean that 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 4.4.3.

Our value specification for `select_sum` is as follows:

```
val select_sum(v1: [i32; N], v2: [i32; N], v3: [i32; N]) -> out: i32 {
```

```
    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 pure (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 `i32`, and this function produces data of type `i32` (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 4.4.2, 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([i32; N]) -> i32
    extern(gpu) pure sum_gpu([i32; N]) -> i32
}
```

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.

4.4.2 Implementation Language

With our value specification and function equivalence classes in hand, we can now implement a (simplified) Caiman program for `select_sum`. This program is simplified in that we have not yet defined the timeline and spatial languages (as we will see, we can provide trivial definitions to practically elide them from the Caiman implementation). Also, 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.

We start with a useful Caiman detail, defining our function `sum` to support two CPU implementations and one GPU implementation, the details of which we will examine later in this section:

```
feq sum {
        extern(cpu) pure sum_cpu_1([i32; N]) -> i32
        extern(cpu) pure sum_cpu_2([i32; N]) -> i32
        extern(gpu) pure sum_gpu([i32; N]) -> i32
}
```

These declarations are written to resemble Rust header functions, with additional syntax to indicate that whether each declaration is on the cpu or gpu. We also indicate with `pure` that each function here will not modify the function arguments.

Our use of two definitions of `sum_cpu` are 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.

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: [i32; N], v2: [i32; N], v3: [i32; N])
  -> i32 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 can annotate lines of code as follows:

```
let cond : bool @ node(val.condition) = s1 < 0;
```

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 `i32`, and a (value) specification type of `node(val.sum1)`.

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

- `node`, which indicates that our specification variable is defined within the specification function rather than an argument or returned value of that function
- `val` states that the given type is a part of the value specification this function is implementing, `select_sum` in this case
- `condition` states that we are working with the specific node within our value specification named `sum1`.

We can also use this pattern to write the following header, equivalent to what we had written before:

```
fn select_sum_impl(
  v1: [i32; N] @ node(val.v1),
  v2: [i32; N] @ node(val.v2),
  v3: [i32; N] @ node(val.v3)
  -> i32 @ node(val.out) ... {
}
```

Our focus in this rewrite is to expose the (baseline) types used by a Caiman implementation. Crucially, this is part of the type of the implementation variable as much as the datatype

`i32`, though the specification type can be erased when compiling Caiman code. In other words, if we instead wrote the (incorrect) annotation:

```
let s1 : i32 @ 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 previously defined `val.sum2:-sum(v2)` in the specification. Specifically, since our implementation variable `v1` has type `input(val.v1)`, the type we provided of `val.sum2` and the declared return type of `sum(v1)` (namely `val.sum1`) are not the same.

An important observation is the type of `res`, which is derived from the `if-else` condition logic being applied. In essence, we consider `if-else` statements in Caiman's implementation language to always produce a return type. More precisely, the type of each branch of a condition in a Caiman implementation must exactly match the types of the associated specification variables.

For instance, we expect in this function to produce a value associated with the specification variable `node(val.sum2)` in the true case of the `select_sum` function, and a value associated with `node(val.sum3)` in the false case. If we were to change our code to instead calculate and return `sum(v3)` in the true case of this condition, we would produce a value associated with node(val.sum3). This mismatch is sufficient for Caiman to reject this program with 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 ... {
    let s2 = sum_cpu(v2);
    let s3 = sum_cpu(v3);
    if sum_cpu(v1) < 0 {
```

```
        s2
    }
    else {
        s3
    }
}
```

We visibly compare this with our earlier implementation:

```
fn select_sum_impl(v1: [i32; N], v2: [i32; N], v3: [i32; N])
-> i32 impls select_sum,... {
        if sum_cpu(v1) < 0 {
                sum_cpu(v2)
        }
        else {
                sum_cpu(v3)
        }
}
```

An explicitly typed version of both `select_sum_impl` and `select_sum_impl_2` can be found in Appendix B.1.1. This implementation still typechecks and so maintains the value semantics of the `select_sum` specification – a proof of this claim will be given in 4.5. 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 4.4.1), 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. As a reiteration, we have defined an equivalence class of `sum` as follows:

```
        feq sum {
                extern(cpu) pure sum_cpu_1([i32; N]) -> i32
                extern(cpu) pure sum_cpu_2([i32; N]) -> i32
                extern(gpu) pure sum_gpu([i32; N]) -> i32
        }
```

We can freely use either definition within our implementation of the `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 ... {
    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 2.1. 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 4.9.1.

4.4.3 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.

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 process mirrors that of the GLSL and WGPU submission processes, described in Section 4.3.1. 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 **??**. 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.

First, we build an event to represent requesting an encoding. Specifically, we take in some event and specify to start an encoding on device `B` with that event. We then return two events, the first corresponding to a local event (on device `A`), and the second corresponding to a remote event (on device `B`):

```
local, remote :- encode_event(e)
```

Second, we build an event to represent requesting an submission. Specifically, we take a remote event and return an event indicating that we have resolved this submission:

```
sub :- submission_event(remote)
```

Third, we build an event to synchronize with the device. Specifically, we take in a local event corresponding with some encoding, and a remote event corresponding with some submission, and produce an event indicating that we have resolved this synchronization:

```
snc :- synchronization_event(local, sub)
```

We will explore a more detailed example of using the timeline language shortly in Section 4.4.4. 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 `buff` into $n \geq 1$ 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. The trivial spatial specification we will use is written as follows:

```
sptl space(s: BufferSpace) -> BufferSpace {
        returns s
}
```

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). We will see a concrete example of this “partial specification" approach in Section 4.4.4.

4.4.4 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: [i32; N], v2: [i32; N], v3: [i32; N])
  -> out: i32 {
    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(in : Event) -> out : Event {
    local, remote :- encoding_event(in)
    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 most explicit value types, though a complete and explicitly typed version can be found in Appendix B.1.2. We start by providing a function indicating that

we are implementing `select_sum` along with `single_sync`:

```
fn select_sum_impl_gpu(v1: [i32; N], v2: [i32; N], v3: [i32; N])
  -> i32 impls
        select_sum,
        single_sync,
        trivial_spatial {
```

We first calculate our condition on the CPU, as before, noting that this operation has no connection to our timeline:

```
if sum_cpu(v1) < 0 {
```

Inside of this condition, we now request that the GPU create space and produce an encoding for our array copy `v2_gpu` and the write destination `v2_gpu_sum`. Once we have an encoding with sufficient space for `v2` to be written to, we can schedule a copy of that data locally onto the GPU, specifically into a separate variable named `v2_gpu`:

```
let encoder = encode-begin { v2_gpu, v2_gpu_sum } gpu;
encode encoder.copy[v2_gpu <- v2];
```

With our copy scheduled, we can now schedule a call to be made using the newly constructed `v2_gpu` variable:

```
encode encoder.call[v2_gpu_sum <- sum_gpu(v2_gpu)];
```

Now that we have worked out the operations needed on the GPU, we can submit that all operations are ready to begin. Note that we must provide an encoder that matches with our timeline specification:

```
let fence = submit @ tmln.sub encoder
```

Finally, we await the result of all data from the GPU, and return specifically the piece of that data containing our result, named as `s2_gpu`:

```
let gpu_data = await @ tmln.sync
gpu_data.s2_gpu
```

We would include a similar implementation of the above for the `else` case of our condition, using `v3` instead of `v2`. Observe that this means we have resolved our timeline requirements equivalently in either case, and so we have fully implemented our `single_sync` timeline specification.

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.

Expanding Caiman Specifications

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 this function entirely on the GPU and hide the details from Caiman. 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. For this approach, we need to define 2 functions, each of which implement the `single_sync` specification. The first of these is a simple function to calculate the sum of `v2`:

```
fn select_sum_impl_gpu_2(...) ...
  impls select_sum,single_sync,trivial_spatial {
    ... // usual invocation of the GPU with v2

    let s2 = gpu_result.v2_gpu_sum;
    select_sum_impl_gpu_2_ret(v1, s2, v3)
}
```

As part of this code, we call into another implementation `select_sum_impl_gpu_2_ret`. This function takes in similar arguments, but such that we have already calculated the sum of `v2`. Concretely, the implementation of this helper function may resemble the following:

```
// same arguments as before, but with s2 : i32
fn select_sum_impl_gpu_2_ret(...) ...
  impls select_sum,single_sync,trivial_spatial {
        ... // usual invocation of the GPU with v3
        let s3 = gpu_result.v3_gpu_sum;
        if sum(v3) < 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 specifications while avoiding 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 verify that the arguments and return types of each function are what the programmer declared, we need not worry about the potential complexity of deducing data flow interactions between functions.

Another important observation is that we can use a similar idea to break apart encoding and synchronization, allowing a programmer to first encode an operation, then continue doing work on the local 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.

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

Figure 4.1: Spawnling Specification Syntax

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 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 4.6. 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.

4.5 Formal Model

In this section, we will be describing a formal model of the Caiman language. We will use this to prove our assertion that the Caiman typechecker will ensure a given (typed) implementation has the same (observational) semantics as a specification it implements.

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

Figure 4.2: Spawnling Implementation Syntax

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 *Spawnling*. Spawnling 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, Spawnling has a specification and an implementation. Our primary goal will be to setup infrastructure for, but not prove, the following theorem:

Theorem 3. *Any well-typed Spawnling implementation program with types matching those of a well-typed Spawnling 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 4.4.1), 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 Spawnling 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 Spawnling 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 Spawnling 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 Spawnling 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.

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

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

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

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

Figure 4.3: Spawnling 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 Spawnling 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 Spawnling programs must specify an output type. For the Spawnling specification program, we denote this as τ_{out}, while for the Spawnling 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.

4.5.1 Typing Semantics

We describe the type semantics for a Spawnling specification in Figure 4.3. The first two rules, applied to $.$, indicate that Spawnling specification operations are unordered; we can either apply the typing judgment to the left or the right of a $.$ operation and use the

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

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

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

$$\frac{\Gamma, \Gamma' \vdash c, \Gamma'(x) = \psi_{\text{out}}}{\Gamma, \Gamma' \vdash c; \text{return}\, x}$$

Figure 4.4: Spawnling Implementation Typing Judgment

result in the other. Similarly, we terminate a specification program with `return`, and require that the expected type τ_{out}. Note also that we do not allow multiple definitions of a specification variable through requiring they are not already included in Ψ.

The most interesting (if dense) operations in this semantics are the types to the operations $\phi(\psi_1)$ and $\text{if}\, \psi_1\, \text{then}\, \psi_2\, \text{else}\, \psi_3$. In both of these cases, we require that types "match up", where we use _ to indicate that the remainder of our stored type is irrelevant. The type of what is added to the context Ψ is necessarily a bit messy, requiring that we retain information about the structure of the specification type for the implementation. Specifically, for function calls, we produce the returned type, the function being called, and the input to the function being called. Similarly, for conditions, we produce the (matching) type of each branch, no function being called, and the set of three variable used in the condition.

We then describe the type semantics for the Spawnling implementation in Figure 4.4. The semantics for these implementations match those of an imperative language in most cases, including allowing reassignment of variables, sequential operations, and

declarations of variables to fix a type. Since a Spawnling implementation must terminate with a `return`, a Spawnling implementation can only typecheck if we produce a result of type ψ_{out}.

A Spawnling implementation is not explicitly tied to a particular specification outside of the definition of Ψ being used. We note that the context Ψ being used is fixed throughout our typechecking of an implementation. We use the specification typechecking results stored in Ψ to validate that the argument x_1 to the function call f_ψ matches the type of our specification requirement, namely ψ_1. The rule for conditions is similar, instead requiring that each branch match with the associated branch of the specification.

We do not provide here the denotational semantics of either the Spawnling specification nor implementation languages, noting that they are straightforward to work out and necessary to resolve our correctness theorem.

4.6 Explication

As observed at the end of Section 4.4, hand–writing Caiman implementations are painful and can practically lead to significant barriers to experimentation. Specifically, the code itself is dense and can be hard to navigate, but we have found it to be anecdotally difficult to write even with compiler support. Indeed, it feels as though the work done when implementing a Caiman program is both essentially the same work as writing a Caiman specification and can obscure the performance characteristics being explored.

The entire design of Caiman, however, is such that this task of writing an implementation from a specification can be automated with annotation. More specifically, we can apply a form of type–directed program synthesis, as seen in several works .

We use explication to describe the task of synthesizing a Caiman implementation from a Caiman specification in the presence of programmer–written implementation requirements. To make this statement more concrete, we will start by working from example of how this looks in the Caiman frontend.

The author notes that at the time of this writing, the programs shown in this section cannot be written as–is in the Caiman frontend, due entirely to minor missing engineering work on AST transformations. Approximately equivalent programs can be written explicitly in the Caiman IR (described in 4.6.2, and the programs are included in Appendix B.1.4), but I believe it is important to note this limitation in our actual implementation. Consequently, more annotations may need to be included in the finished Caiman frontend (to help guide the explicator) than are shown here.

4.6.1 Using Explication

We will start by reiterating our usual value specification for `select_sum`:

```
val select_sum(v1: [i32; N], v2: [i32; N], v3: [i32; N])
  -> [out : i32] {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

Along with a trivial timeline and spatial specification, this is enough to write a fully explicated implementation of `select_sum`:

```
fn select_sum_expl(
  v1: [i32; N] @ val.v1,
  v2: [i32; N] @ val.v2,
  v3: [i32; N] @ val.v3
) -> i32 : val.out
  impls select_sum,trivial_time,trivial_space{
```

```
  ???
}
```

With explication, we still need to provide a header (arguments and return types) and the specifications to implement. When we write `???`, however, we are semantically saying that any type-safe code can be inserted to replace this *multi–line hole* in the program.

The Caiman *explicator* can use then types we have given this function to deduce some program that matches these types. The explicator will not, however, be able to use any other specifications than those given, meaning that it is often the case that no such program exists. For instance, if we had excluded `v1` from our arguments, the `select_sum` specification provides no way to construct `v1`, and so the explicator would fail to produce a solution.

Importantly, however, the explicator is somewhat possible to control outside of this strictly open `???` hole we just defined. More precisely, we can summarize our explicator requirements with three rules:

1. The synthesized code must typecheck according to our given inputs and outputs, and as defined by our specification.
2. For a fixed implementation and fixed specifications if the explicator produces a program, the explicator must always produce the same program. That is, the explicator must be deterministic for the resulting program.
3. Any operations provided by the programmer must be used in the order they were written. Additionally, new operations can only be added where we have a multiline hole (`???`)

The last of these rules, the ordering of operations, is the key piece of the Caiman explicator design that allows for performance manipulation without adjusting the specification (or

writing many implementation functions). We will describe precisely what this rule entails after exposing more of Caiman's implementation in Section 4.6.2. For now, as a concrete example, let us restrict the explicator for `select_sum` to ensure that the sum of `v2` and `v3` must be calculated before the condition:

```
fn select_sum_expl_2(
  v1: [i32; N] @ val.v1,
  v2: [i32; N] @ val.v2,
  v3: [i32; N] @ val.v3
) -> i32 : val.out
  impls select_sum,trivial_time,trivial_space{
  .. = sum(v2);
  .. = sum(v3);
  ???;
}
```

The key addition to this code is that we have stated that both `sum(v2)` and `sum(v3)` must be computed and stored (in the unnamed variable `..`) before any other computation can be done. With respect to our explication rules, the order here is important – by having the `???` after these computations, we've required that the explicator can't insert code that we don't expect before these operations.

Of course, the more important reason to introduce to explicator is to help with exploring the more detailed (and complicated) device communication space. Fortunately, we have introduced enough of Caiman design to use the explicator to help us with writing more complicated with the GPU, taking advantage of the bounds imposed by the timeline specification along with our current value specification. We first write the timeline specification as shown in Section 4.4.4 (restated here to help with readability):

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

And now we can write our implementation, providing some structure to ensure we achieve

our desired behavior of only encoding and syncing with the GPU once depending on our condition:

```
fn select_sum_expl_gpu(
  v1: [i32; N] @ val.v1,
  v2: [i32; N] @ [val.v2, tmln.in],
  v3: [i32; N] @ [val.v3, tmln.in])
  -> i32 @ [val.out, tlmn.out]
  impls select_sum,single_sync,trivial_spatial {
    if sum(v1) < 0 {
        ???
    }
    else {
        ???
    }
}
```

Since we have only allowed work within each branch of the condition, we can be confident that we will compute `v2` and `v3` respectively. Additionally, our restriction that our inputs are associated with `tmln.in` and our outputs are associated with `tmln.out` means that we must have synced with a device to produce these results. Strictly speaking, Caiman does not guarantee which device is used, as in our current explicator implementation we only allow synchronization with the GPU; this would, however, not be difficult to change.

A slightly more interesting observation is that eliding the condition here (`if sum(v1) < 0`) will result in the same program in this particular case, due to the way we have written our timeline specification. By both specifying we can only synchronize with the GPU once, only providing a definition for a single `sum` kernel, and requiring that we produce a result with `tmln.out`, we have restricted our implementation sufficiently that there is only one viable solution. This sort of constraint can be difficult to observe in practice, however, motivating the use of partial implementations such as the condition shown here in cases where the programmer explicitly desires this structure.

The other important consequence of observing the extent by which Caiman specifications restrict implementation is how many specification and partial implementations

arrangements result in an overconstrained and unsolvable system. This can be seen as advantageous, in that attempting an explication can save a programmer significant effort after reaching an impossible–to–proceed state, but the explicator may instead time out without resolution. Consequently, transparent messaging is important when working with the explicator, both information about what the explicator was able to produce and where it may have become stuck. The specific engineering around the explicator and user messaging will be explored in Section 4.7.1.

For the remainder of this section, we will examine the algorithm used by the Caiman explicator. Before we can do so, however, we need to expose a lower–level representation of Caiman code, the Caiman IR.

4.6.2 Caiman IR

So far, we have focused on the human–usable Caiman frontend, used to illustrate the intention of Caiman design and how this design can be realistically used by a programmer. When dealing with explication and precise performance characterization, however, it is helpful to expose the precise operations evaluated by Caiman in the form of an exact (dense) intermediate representation. It is worth noting that the Caiman IR is implemented to be interfaced with directly as needed, though ideally a programmer is able to avoid writing such code explicitly.

Caiman IR is written to be syntactically distinct from the frontend to avoid confusion, and has the following properties meant for analysis:

- A Caiman IR program consists of “funclets" (similar to basic blocks with named inputs and outputs) to contain operations. There is no control flow in a funclet, and

calls between funclets use a style similar to CPS, or more precisely, to compiling without continuations [27].

- Caiman IR uses Static Single Assignment, so each name is unique. For ease of use, we allow `\%_` to be used to represent an anonymous variable (which may still be used by the Caiman explicator).
- Each operation consists of exactly instruction, and produces either nothing or exactly one result.
- There are minimal operations in Caiman IR (around 30 with our current implementation), with much of the work being done on external operations.

These properties are why we need to expose Caiman IR for explication. Specifically, we need to have a sense of Caiman's internal representation to describe explication concretely.

For the sake of examples, we will expose 7 operations in a Caiman IR implementation function. Note that there are distinct operations used in the Caiman IR specifications; these are similar enough to the specifications we have already seen that we will not need to show them here (complete Caiman IR examples can be found in Appendix B.1.3). For these operations we are exposing, we are ignoring some implementation cruft in the Caiman IR that is unnecessary for understanding the programs, but will appear in the examples provided. These operations are described as follows:

- `alloc-temporary`, which takes in a place (local, cpu, or gpu) and an exposed datatype (such as `i32`), and produces a memory location that can be written to. Caiman IR does not distinguish between stack and heap allocations. We expose `alloc-temporary` as the only allocation mechanism for simplicity, in cases

where we need to return the reference rather than just data, we would use another operation instead.

- `read-ref`, which takes in a reference and returns the data referred to by this reference.
- `begin-encoding`, which takes in a place to encode to (cpu or gpu), an associated timeline specification node, and a list of memory locations to request for encoding, then returns a reference to an encoder.
- `local-do/encode-do`, which takes in an external function (such a `sum`), a specification node to match with, arguments to the function, and, in the case of `encode-copy`, an encoder to operate with. With this, this operation writes the result of this function to call to a given allocation slot. Returns nothing.
- `local-copy/encode-copy`, which takes in a source reference and a destination reference (and an encoder in the case of encode–copy), and copies the data referred to by the source to the location referred to by the destination. Returns nothing.
- `submit`, which takes in an encoder and a timeline specification node to associate with, and submits the encoded jobs to the associated device. Returns a reference to the resulting fence to later synchronize on.
- `sync-fence`, which takes in a fence and a timeline specification node, and requires synchronizing on that fence before continuing. Returns nothing.

Additionally, the Caiman IR defines several possible tail edges for funclet implementations to provide type information and to manage continuations when interacting with control flow. We will expose 2 of these tail edges, though to avoid excessive detail, we will defer discussion of the continuation logic to Section 4.7:

- `return` takes a single argument as the value to be returned from this funclet. Note that, to pass typechecking, the given data must match all of the specifications required by the return type of this funclet.
- `schedule-select` takes in a `i32` or `i32` value to use to select a function, an ordered array of funclets (with matching arguments and return types) to select, the specification node(s) this selection is associated with, the arguments to the selected function, and a function to join with per continuation–passing style. The return type of each selected funclet must match arguments of the joined funclet, and the return type of the join funclet must match the return type of this funclet.

Worked Example

With our definitions in mind, we can now show an example assembly implementation for `select_sum`. We will be showing the hand-translated equivalent to the implementation first shown in 4.4:

```
fn select_sum_impl(v1: [i32; N], v2: [i32; N], v3: [i32; N])
  -> i32 impls select_sum,... {
    if sum_cpu(v1) < 0 {
        sum_cpu(v2)
    }
    else {
        sum_cpu(v3)
    }
}
```

Since we will need to refer to specification nodes for our assembly implementation substantially, we will also rewrite our usual value specification to make these names concrete:

```
val select_sum(v1: [i32; N], v2: [i32; N], v3: [i32; N])
  -> [out : i32] {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
```

```
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

Since Caiman IR funclets do not have control flow, we must start by defining a function to compute the condition of the `if` statement, and then calling funclets that represent the left and right branches with `schedule-select`. Note that we will be using `...` to indicate elided syntax cruft that is irrelevant for this example (as opposed to the `???` or `?` Caiman syntax for an explication hole):

```
schedule[...] %select_sum_main<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    // We are eliding irrelevent allocation details
    // Note we allocate locally rather than on the CPU
    //   in doing so, we can avoid synchronizing on the CPU
    //   which may, in general, be another device
    %ref = alloc-temporary local [...] i32;

    // write the result of sum(v1) to ref
    local-do-external %sum val.%sum1(%v1) -> %ref;

    // read the result into a local variable
    %sum1 = read-ref i32 %ref;

    // We will not show how to get the constant 0
    %zero = ...

    // write the result of sum1 < 0 to ref
    local-do-external %lt val.%condition(%sum1, %zero) -> %ref;

    // read the result into a local variable
    %cond = read-ref i32 %ref;

    // we will not show how we are computing join
    %join = ...

    // we will be writing the left and right branches shortly
    // note that we must pass %v2 and %v3 to both
    //   since their input types must "match up"
    // note also that we specify %result rather than %out
    //   since the if/then/else in the specification gives %result
    schedule-select %cond [%select-sum-left, %select-sum-right]
      [val.%result, ...] (%v2, %v3) %join;
}
```

Syntactically, it is worth noting that we use % to indicate a Caiman IR defined

variable to help with distinguishing from the frontend and to visually distinguish between operations and variables. We have been careful to note each detail we are eliding, which are either irrelevant or out of scope for this writing – most notably the calculation of the join operation, the complete type syntax Caiman IR uses, and additional qualifiers needed for allocation.

This code provides a precise representation of the line-by-line operations required for Caiman's analysis and compilation. Fortunately, the left and right branches of this condition are simpler than this selection program, at least when we run locally:

```
schedule[...] %select_sum_left<...>
(%v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%sum2 ...] {
    %ref = alloc-temporary local [...] i32;
    local-do-external %sum val.%sum2(%v2) -> %ref;
    %res = read-ref i32 %ref;
    return %res;
}

schedule[...] %select_sum_right<...>
(%v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%sum3 ...] {
    %ref = alloc-temporary local [...] i32;
    local-do-external %sum val.%sum3(%v3) -> %ref;
    %res = read-ref i32 %ref;
    return %res;
}
```

These funclets are nearly identical, except for their return type and computation. We observe that these functions can be used in selection because we have annotated our selection with `val.%result`, which is sufficient for the typechecker to agree so long as each branch return type "lines up" with its respective branch in the specification. Finally, then, we can write the (trivial) funclet to join the original selection:

```
schedule[...] %select_sum_ret<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    return %res;
```

```
}
```

Morally, this function mirrors the return that we have in the specification, and explicitly provides the connection between `val.%result` and `val.%out`.

Extending to the GPU

With our overview of Caiman IR in hand, we will briefly examine how to interface with the GPU. This code will be quite similar to our high–level Caiman frontend example shown in 4.4.4, but is useful for demonstrating our introduced Caiman IR GPU operations.

Specifically, we can modify our funclet `select_sum_left`, (note that similarly we could modify `select_sum_right`). Concretely, we will be implementing the usual single–synchronization timeline function:

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

Concretely, our Caiman IR implementation is as follows:

```
schedule[...] %select_sum_left<...>
(%v2 : val.%v2 time.%in ...,
 %v3 : val.%v3 time.%in ...)
  -> [%out : val.%sum2 time.%out ...] {
    // Create a reference to the gpu memory
    //   we will be able to write to
    // Note that this memory is not available
    //   until we have an encoding
    %v2_gpu = alloc-temporary gpu [...] [i32; N];
    %res_gpu = alloc-temporary gpu [...] i32;
    %res_local = alloc-temporary local [...] i32;

    // Begin an encoding with references
    //   for our input (v2_gpu) and output (res_gpu)
    %enc = begin-encoding gpu time.%enc [%v2_gpu, %res_gpu] [];

    // We can now write to this memory location
```

```
    // Note that this copy does not have
    //   an associated specification operation
    // In other words, if we had some
    //   other mechanism to have a value
    //   of type v2 in the right place at the
    //   right time, we could use that instead
    encode-copy %enc %v2 -> %v2_gpu;

    // Encode running the `sum` operation after our copy
    // Writes the result to our earlier encoded memory location
    encode-do %enc %sum_gpu val.%sum2(%v2_gpu) -> %res_gpu;

    // Submite the operations and recover a fence
    %fnc = submit %enc time.%sub;

    // Synchronize locally against the GPU result
    sync-fence %fnc time.%snc;

    // Copy the result back to the local allocation
    // We can do this because we have synchronized
    local-copy %res_gpu -> %res_local;
    %res = read-ref i32 %res_local;
    return %res;
}
```

In this GPU implementation, we hint at type rules to manage memory allocations when encoding, submitting, and synchronizing. We will include engineering for these interactions in our explicator implementation, but the type–safety guarantees we would intuitively aim for are left to a future work.

4.6.3 Explication of Caiman IR

Before diving into this session, we note that, at the time of this writing, our actual explicator implementation for this example at this time of writing is missing a couple of implementation details that are described here. Specifically `???` is not carefully tested, and `begin_encoding` is not fully implemented.

With our more-exact representation of Caiman IR in hand, we can provide a precise definition of Caiman's explication syntax. Caiman allows exactly two synctatic "holes",

defined as follows:

- `?`, which indicates a specific component of a specific operation can be explicated
- `???`, which indicates that the explicator can introduce any number of operations (including introducing new funclets for control flow).

As before, `???` also means that the explicator can only introduce operations within the block outlined by operations surrounding this multi–line hole. We will summarize these requirements through a partial explication for `select_sum_main`:

```
schedule[...] %select_sum_main<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    // do any setup needed to work out the condition
    ???;

    // require that we use exactly %lt,
    // and that we must work out val.%condition in this funclet
    // note that we explicitly state
    //  a requirement of two arguments here
    //  though we do not say what these arguments are
    local-do-external %lt val.%condition(?, ?) -> ?;

    // do any setup neeeded for the schedule-select
    ???;

    // We must branch into the funclets
    //   select-sum-left and select-sum-right
    // We also must have this select be associated with val.%result
    // otherwise, the explicator may do as it would please
    schedule-select ? [%select-sum-left, %select-sum-right]
      [val.%result, ...] (?, ?) ?;
}
```

There are many ways we could write this partial implementation such that no solution exists, including providing the incorrect number of required arguments to `%lt`. Given how large the space of impossibilities are, the explicator makes no attempt to prove that no program solution exists, but instead will fail to terminate. Additionally, the explicator does not fully typecheck existing code, and so only ensures that the code produced

is type–safe assuming a correctly typed partial implementation. Details of how our engineered explicator with the Caiman typechecker are expanded more in 4.7.1.

The Caiman explicator also works with the operations needed to encode and synchronize on another device. For example, the following funclet header is sufficient to produce the implementation described in 4.6.2:

```
schedule[...] %select_sum_left<...>
(%v2 : val.%v2 time.%in ...,
 %v3 : val.%v3 time.%in ...)
  -> [%out : val.%sum2 time.%out ...] {
    ???;
    encode-do ? %sum_gpu val.%sum2(?) -> ?;
    ???;
}
```

Note that we must include this specific `encode-do` to be sure that we are actually running the operation on the GPU. Otherwise, the explicator could very reasonably choose to run this operation locally, and simply apply a no–op to the GPU to progress to `time.%out`. In other words, the particular operation we expect to apply to compute `sum2` would otherwise be left as an implementation detail of the explicator.

4.6.4 Core Algorithm

We now have the context to define an explication algorithm. Generally speaking, our approach will be to make arbitrary (type–safe) decisions for the first node of the funclet, recurse to see if we produce a valid explication result, and then try the next decision we could make. Conceptually this algorithm resembles a depth first search through the decisions we could make, and if we find a type–safe result, we simply return this result.

The algorithm used to explicate a node of a Caiman IR funclet is more precisely described in Algorithm 5. We start this recursion with an empty list of solved nodes, all of our funclet inputs in the set of unused nodes, and our funclet definition. Note that the

funclet definition we provide is intended as a static reference, and is not modified as we recurse.

If *NodeExplication* returns **NONE**, then no solution was found. If this function returns a pair that includes a list of any requests for multiline explication, then we continue the recursion with the next solution for the first node. Otherwise, we accept the valid explication solution as the list of explicated nodes (along with an explicated tailedge, which is elided in the result for readability).

unusedNodes deserves special mention, as this set holds information about all programmer–given nodes (starting with the funclet inputs). Per our requirements on explication, we must use all programmer–given nodes at some point in the function, so we use this set to track if our current explication attempt fulfills this constraint. Multi–line explication nodes are exempt, so we need not add them to the set as we recurse.

We now need to specify several of the functions used in this algorithmic approach:

- *TailEdgeExplication* works out if, for the given funclet types and current explicated nodes, we are able to both produce a result of the correct type from the funclet and use all remaining unused nodes. If we can fulfill both requirements, this function returns the resulting explicated tail edge and any multi–line explication requests. Otherwise, this function returns **NONE**.
- *ValidSolutions* takes in our current node selections and our current node index (along with our usual funclet information), and determines all the ways this node could be explicated given what we already have, called solutions. Note that, abstractly, this function is exhaustive – for example, if we have an allocation, this function will iterate through every possible type that could be allocated. To maintain algorithmic determinism, the order we iterate through solutions must be

deterministic. In practice, we provide heuristics on this order of solutions to make it more likely we will find an explication result quickly.

- *GetUsedNodes* takes in our current solution, and returns the set of nodes it uses.
- *GetMultilineRequests* takes in our current solution and list of nodes we have already found, and returns a list of nodes that must be added to a multiline explication. Note that, since we must use every programmer node to complete explication, these newly generated nodes will not “conflict" with existing user choices. We will expand what requests are needed for a given node shortly.

A key piece of the Caiman explication algorithm is writing down what a given explication solution will need to request for multiline explication. This detail is why Caiman’s type system is setup such that the specification(s) provided are sufficient to narrow down such a request. Indeed, in practice, there is often only one operation (or chain of operations) that will satisfy a given operation under our specification type constraints.

To make this decision making more concrete, we will specify the exact operations required for a few of our defined Caiman IR operations. Note that the precision of Caiman IR is crucial here, allowing us to have a precise (finite) number of holes we can fill for a given operation. Note also that we use the term *selects* as a stand in for iterating through every possible fill choice and fixing that decision for the remaining requirements:

- `alloc-temporary` has no requirements, and, as presented, iterates through every place and every supported datatype.
- `local-do-external` first selects a value specification node (say `val.\%sum1`) and an implementation in that class. This operation then requires all arguments to that specification value be already calculated, and that there is an allocation for our return type as a write target.

- `read-ref` selects a datatype, and then requires that there be an allocation of that datatype.
- `submit` selects a timeline specification node, and requires that we have an encoder at a time where we can submit.

While we have not enumerated every implementation operation we have presented, it should be straightforward to derive the remaining operations. One operation that bears special mention, however, is `begin-encoding`, which allows us to encode any number of operations as an argument. The algorithmic solution is to consider the powerset of all allocations and iterate through this set. In practice, we treat `begin-encoding` as a special case of multiline explication, where instead of allowing arbitrary operations, we can request that an allocation be included in the encoding.

While Caiman does currently use the algorithm we have described for explication, the critical takeaway is that there *exists* an algorithm that satisfies our requirements from earlier. Practically, we expect that pushing parts of this explication algorithm to a solver or synthesis tool may help with explicator performance and scalability. This being said, a qualitative analysis of using Caiman's explication approach will be discussed in Section 4.8.1

4.7 Caiman Engineering

We have built an implementation of the Caiman language and Caiman IR as a toolchain we will refer to as the Caiman compiler. For our implementation[1], we take in Caiman or Caiman IR code, apply explication and semantic analysis, and emit Rust code to connect Rust (on the CPU) and WebGPU (on the GPU) through the WGPU interface.

[1]https://github.com/cucapra/caiman

Our choice to emit Rust and WGPU is somewhat arbitrary – indeed, practically we would recommend that future work on a Caiman compiler instead target an IR rather than Rust directly, and a lower–level interface like Vulkan. Nevertheless, our implementation of Caiman shows how we can construct graphics–focused and dynamic CPU/GPU code, more concretely described in Section 4.8.

4.7.1 Compiler Infrastructure

The stages of work done by the Caiman compiler can be seen in Figure **??**. For the remainder of this section, we will examine some of the technical details for each stage of compilation, working from the top of the compiler (parsing) to the bottom (code generation).

Caiman to Caiman IR

Caiman has two distinct representations in the frontend and the IR, resulting in two distinct Abstract Syntax Trees (ASTs). The transformation from the Caiman frontend AST to the Caiman IR AST is straightforward (if non–trivial to build), needing to transform the specification and implementation logic to the single-operation and SSA style of the Caiman IR. Specifications (value, timeline, and spatial) in Caiman have no control flow, so translating to the similar representation in Caiman IR requires only (simply) reducing expressions.

Translating a Caiman implementation to Caiman IR is somewhat more involved, however, as we need to account for the control flow of the function (with respect to the specification) as well as provide information about the allocation of resources in

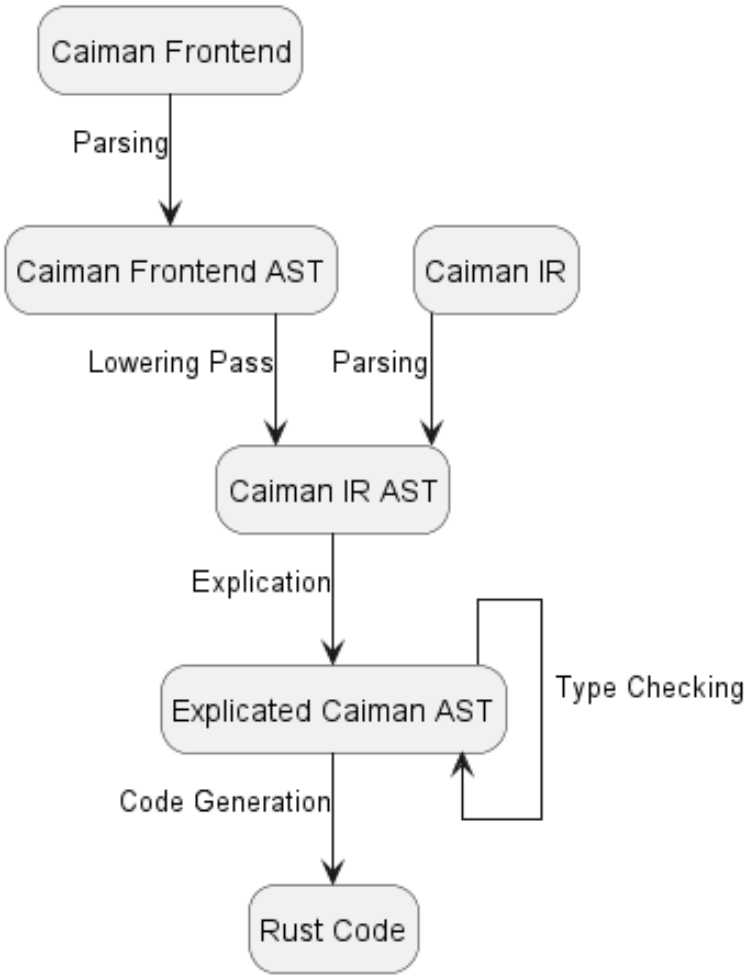


Figure 4.5: Structure of the Caiman Compiler

computation. For conditions, function calls, and recursion, we translated directly for the call site based on the types of the values being used. We do not currently attempt to implement loops in Caiman. Allocations are calculated based on value use – in principle, it would improve explicator flexibility to move allocation logic to the explication search, but we have not practically engineered this approach.

Additionally, both specifications and implementations include standard Hindley-Milner type inference [14]. For the implementation language, this type inference is primarily useful for control flow headers (which cannot be explicated), and values not inferred are instead pushed to the explicator for solving.

Caiman IR explication and typechecking

To lower the Caiman IR for typechecking, we erase variable and function names and validate program structural assumptions. We then explicate the program as described in Section 4.6. To help make debugging explicated Caiman code manageable, the explicator maintains names throughout, requiring the explicator to retain metadata about the transformed program. Additionally, at this stage, the explicator performs minimal typechecking to perform explication. Importantly, however, the explicator makes no guarantee that existing code typechecks, and only intends to produce explicated code that will typecheck if existing code will typecheck. We do not validate this claim, and instead leave formalizing this connection to future work.

After explication, we guarantee that the explicator either failed to terminate (due to a type error or due to timing out), or every hole in the AST representation of the Caiman IR is filled. Practically, we represent this state with a duplicate AST with no hole variants. The typechecker is then able to walk through each function and validate our requirement that each implementation function does the work promised by the implemented specifications, as stated in Section 4.5. Additionally, the typechecking pass here validates properties of the allocations and data between funclets, a significant detail of Caiman engineering that is out of scope of this writing.

Code Generation

Finally, the resulting Caiman IR program is emitted as Rust code, which can then be linked to, compiled, and run. This Rust code notably includes calls to externally-defined Rust functions and setting up WebGPU API allocations and function calls. At this stage, code generation makes no attempt to optimize the resulting code (such as inlining code),

leaving these additional optimizations to the underlying compiler.

Caiman code generation currently does not support directly emitting WebGPU code, and as a result treats each WebGPU function as a black–box call. It is possible to fuse these calls, or even to emit WebGPU directly in Caiman, but these additions are not yet implemented. This detail can lead to significant performance loss, and certainly would require an implementation to use Caiman in a practical setting.

4.8 Results

At the time of this writing, quantitative results of Caiman are minimal and not worth reporting. We have, however, implemented around 100 synthetic examples of Caiman programs, which can give qualitative insight into the current implementation of Caiman described. Intended future performance measurements will be summarized in Section 4.9.1.

The programs implemented in Caiman are intended primarily for testing, but provide some idea of the flexibility of our produced implementation. Notable examples of Caiman programs include the following, the full code of which is included in Appendix B.1.5:

- Nested conditional logic
- Recursive functions
- WebGPU function calls, including conditional logic and recursive calls
- Fixed-sized arrays
- `select_sum`, as described in Section 4.4

In developing these examples, we have found that Caiman's type system is remarkably flexible, though it takes time to become comfortable with implementing Caiman code.

Specifically, implementing a Caiman specification requires some amount of care when breaking up the specification and managing types of each operation in cases where type inference is insufficient.

This being said, the Caiman type system has caught dozens of errors we have made while implementing these functions, including mistakes in managing control flow when calling into the GPU. We have also found that, practically, Caiman programs can be written essentially as we might expect despite the specificity of Caiman's type requirements.

The main issue we have had implementing programs is that some implementation rewrites are restricted under Caiman's type system without changing the specification, which is precisely what we would like to avoid. A notable example of this weakness is the technical detail described in Section 4.7.1, where the Caiman semantics is insufficient for merging two calls that we know to be equivalent to a specific single function call.

We have not yet implemented functions that significantly and scientifically test Caiman's resulting runtime performance. Qualitatively, all Caiman programs we have written run nearly instantly, but without comparison to directly written WebGPU programs, this result is, bluntly, insufficient to draw any real conclusions.

4.8.1 Explication

Explication is the most likely stage of Caiman compilation to produce interesting timing results, both in terms of the time of compilation and the runtime of these programs. We have implemented synthetic examples of explication, with at most 30 lines of code for a given explication implementation. Notably, however, at the time of this writing, we have not fully implemented the `???` statement described in Section 4.6.

At the time of this writing, qualitatively, the explication examples we have written compile and run nearly instantly. We expect this to be similarly instant for an implementation written entirely with a `???` statement. Where we expect to see the pathologically slow examples are in functions that use a mix of `???` statements and a small number of restricted expressions, where the greedy search done by the explicator becomes less likely to quickly find a solution. We expect these cases to scale exponentially with the number of lines in a given funclet.

4.9 Conclusion

We have argued that writing performant heterogeneous operations often requires experimentation with multiple function implementations of the same underlying operation. We have seen how this gap between intent and implementation can lead to programmer mistakes and maintaining a combinatorial explosion of function implementations.

With Caiman, we have shown a mechanism by which we can separate the specification and implementation of such heterogeneous operations. We have shown how we can represent control flow and recursive operations in a specification and how to verify that an implementation associated with a specification will calculate the data as we expect.

We have also examined how we can use explication to automate generation of these implementations, and specifically in such a way that this automation can be broken down. More concretely, we have examined how automating the generation of programs can be controlled through providing both a specification and precise line–by–line generation requirements.

Finally, we have described our specific implementation for CPU/GPU code to generate

Rust code to interface with WGPU. We have also shown the structure of the Caiman compiler, and have hinted how code generation can be replaced to work in a variety of heterogeneous settings.

4.9.1 Future Work

Most immediately, timing information about explication and generated Caiman code will be gathered and reported. Such data will allow for more concrete and narrow arguments as to the efficacy of our Caiman implementation and an examination of the potential scalability of explication.

An important observation about the explication results we have observed is that the explicator does not attempt to reason about the interactions between multiple funclets. Introducing general control flow to the explicator would *likely* reduce performance (and expand engineering needs) substantially. This being said, implementing explication in such a way as to generate functions to fill in control flow seems relatively achievable, and would help substantially in implementation details that currently need to be written by hand.

The remainder of this section will discuss potential directions for Caiman that have been discussed, but not implemented.

Caiman's implementation in Rust with WebGPU is fairly narrow and can be difficult to manage. Implementing Caiman to directly generate code for an IR (such as Cranelift [4] or LLVM [26]) would improve Caiman's usability and transparency tremendously. Additionally, targeting Vulkan would make more practical sense in exposing more details in the Caiman communication model.

It would be very interesting to explore an implementation of Caiman targeting either compute (with CUDA or OpenCL), or another device entirely. Caiman seems directly useful for multiple–device code architectures, or for using another non–CPU host (such as GPU–driven logic). Caiman was also designed with FPGAs and network chips in mind, though without an implementation, how effective this design transfers is completely untested.

Rewriting kernels entirely in Caiman seems impractical in many cases, but it also seems conceptually straightforward to write a translator from a subset of C or CUDA to a Caiman specification. Such tooling could have clear practical advantages, allowing a user to harness the type–level power of Caiman without needing to rewrite every specification by hand. The untested downside of this approach could be that Caiman's type system would be too rigid to allow such a straightforward translation, and this would require practical experimentation.

Finally, Caiman explication code can be difficult to debug, particularly in cases where the explicator is directed incorrectly and generates a bunch of strangely–named nonsense that exposes some mistake in the specification. The explication step we have described and implemented is transparent to the compiler, however, and there is an interesting potential HCI challenge in exploring an approach to improve Caiman error messages or provide some sort of visual or IDE tooling to examine explicated Caiman code.

```
Data: A list of solved Caiman IR Nodes, a set of nodes we have yet to use, and
      the current Caiman IR funclet.
Result: Either (a list of explicated nodes, a list of requests for multi–line
        explication to be done) or NONE if no solution was found
[Node] solvedNodes
{Node} unusedNodes
Funclet funclet
begin
    index ← Length(filledNodes)
    nodes ← Nodes(funclet)
                          // base case (we assume 0-indexing)
    if index = Length(nodes) then
        return TailEdgeExplication(filledNodes, unusedNodes, funclet)
    end
    currentNode ← nodes[index]
    if IsMultilineHole(currentNode) then
        result ← NodeExplication(filledNodes + currentNode, unusedNodes,
         funclet)
        if IsNone(result) then
            return NONE
        end
        else
            (explicated, requests) ← result
            return (requests + explicated, [])
        end
    end
    for solution ∈ ValidSolutions(nodes, index, funclet) do
        usedNodes ← GetUsedNodes(solution)
        solutionRequests ← GetMultilineRequests(solution, solvedNodes)
                    // copy nodes, with our solution added
        newNodes ← nodes + solution
        newUnusedNodes ← (unusedNodes ∪ currentNode) \ usedNodes
        result ← NodeExplication(newNodes, unusedNodes, funclet)
        if IsNone(result) then
            (explicated, requests) ← result
                        // prepend our solution and requests
            return (solution + explicated, solutionRequests + requests)
        end
    end
    return NONE
end
```

Algorithm 5: NodeExplication

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

We have examined how domain–specific language design for graphics programming can help programmers with managing correctness and performance considerations. With Gator, we have shown how we can define a specialized type system for reasoning about geometric types, and how these type definitions can provide information for multi–operation transformations. Additionally, we have shown how Gator exposed a surprising challenge in reasoning about equivalence of these transformations through the work on commutative diagrams.

With the Caiman language, on the other hand, we examine how definitions of correctness and equality with respect to a specification can help explore performant solutions. Additionally, we have shown how we can provide decomposable program implementations for heterogeneous code from these fixed specifications through explication.

Perhaps the most valuable work that should be done is validating the use of these approaches in a more practical setting. While we have provided evidence of usability for Gator and Caiman, it can be difficult to rely on intuition for the practicalities of these designs, particularly in larger–scale industrial projects. How do users interact with Gator types and Caiman language features? Can beginners realistically learn and apply these systems? How well does Caiman's approach scale once compilation speed is a realistic concern? These are all questions that may require an academic user study or larger-scale tooling.

To emphasize this second direction more, both Gator and Caiman would have significantly more impact and potential use with better tooling and integration into existing applications. Rewriting a project with hundreds of thousands of lines of code

in either of these settings is clearly unrealistic, and while Gator and Caiman both are designed to “slot in" to an existing codebase, both type systems are much more useful when given more information about the structure of a system.

Concretely, this sort of effort simply requires more engineering. There are, however, several interesting research and design questions within this engineering. Could we meaningfully build a (partial) C-to-Caiman compiler to help with optimizing existing programs, while preserving the Caiman semantic representation? Could we partially annotate a large–scale graphics program with Gator–style geometry types and have inference be “good enough" to be useful? Caiman’s transparency seems to lend itself to some interesting usability cases – could we build a specialized extension for Caiman to “view" the code that the explicator produces?

APPENDIX A

GATOR APPENDIX

A.1 GLSL Phong Source Code

This section lists the full code for the Phong lighting model in plain GLSL [23].

```
precision mediump float;

// External Function Declarations
uniform mat4 uModel;
uniform mat4 uView;
varying vec3 vNormal;
uniform vec3 uLight;
varying vec3 vPosition;

void main() {
vec3 ambient = vec3(.1, 0., 0.);
vec3 lightColor = vec3(0.4, 0.3, 0.9);
vec3 specColor = vec3(1., 1., 1.);

vec4 homWorldPos = uModel*vec4(vPosition, 1.0);
vec3 camPos = normalize(vec3(uView*homWorldPos));
vec3 worldNorm =
normalize(vec3(uModel*vec4(vNormal, 0.0)));
```

```
vec3 lightDir =
normalize(uLight - vec3(homWorldPos));
vec3 reflectDir = reflect(lightDir, worldNorm);

vec3 diffuse =
max(lightWorldDot, 0.0) * lightColor;

float spec = pow(max(-dot(
camPos, reflectDir), 0.), 32.);
vec3 specular = spec * specColor;

gl_FragColor = vec4(ambient+diffuse+specular, 1.0);
}
```

A.2 Gator Phong Source Code

This section lists equivalent code in Gator.

```
#"precision mediump float;";
using "../glsl_defs.lgl";

// Reference Frame Declarations

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

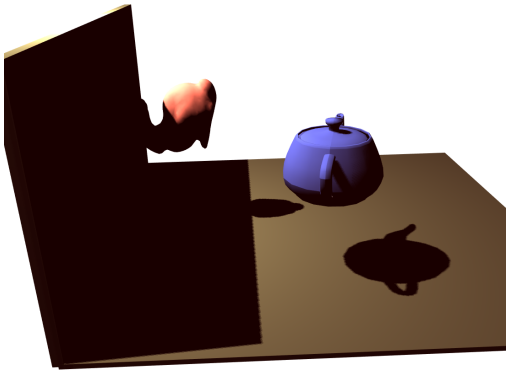
```
frame light has dimension 3;

// Global Variables

varying cart3<model>.point vPosition;
canon uniform hom<model>.transformation<world> uModel;
canon uniform hom<world>.transformation<camera> uView;
varying cart3<model>.vector vNormal;
uniform cart3<light>.point uLight;
canon uniform hom<light>.transformation<world> uLightTrans;

// Shader Code

void main() {
color ambient = [.1, 0., 0.];
color diffColor = [0.4, 0.3, 0.9];
color specColor = [1.0, 1.0, 1.0];

auto worldPos = vPosition in world;
auto camPos = worldPos in camera;
auto worldNorm = normalize(vNormal in world);

auto lightDir = normalize((uLight in world) - worldPos);
auto lightWorldDot = dot(lightDir, worldNorm);
scalar diffuse = max(lightWorldDot, 0.0);
```

```
auto reflectDir = normalize(reflect(-lightDir, worldNorm) in camera);

scalar specular = pow(max(dot(normalize(-camPos), reflectDir), 0.), 3

vec4 gl_FragColor =
vec4(ambient + diffuse * diffColor + specular * specColor, 1.0);
}
```

A.3 Case Study Images

(a) Texture.

(b) Reflection.

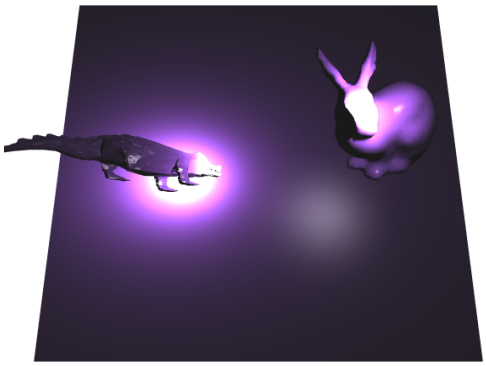


(c) Shadow map.

(d) Microfacet.

Figure A.1: Example outputs from first four renderers used in our case studies.

(a) Phong



(b) Fog

(c) Bump Map

(d) Spotlight

Figure A.2: Example outputs from last four renderers used in our case studies.

APPENDIX B

CAIMAN APPENDIX

B.1 Caiman Examples

B.1.1 Typed Select Sum

Versions 1 and 2

```
#version 0.1.0

tmln time(in: Event) -> Event { returns in }

sptl space(s: BufferSpace) -> BufferSpace { returns s }

extern(cpu) pure sum([i64; 4]) -> i64

val select_sum(v1: [i64; 4], v2: [i64; 4], v3: [i64; 4]) -> out: i64 {
        sum1 :- sum(v1)
        sum2 :- sum(v2)
        sum3 :- sum(v3)
        condition :- sum1 < 0
        result :- sum2 if condition else sum3
        returns result
}

fn select_sum_impl(
v1: [i64; 4] @ [node(val.v1)],
v2: [i64; 4] @ [node(val.v2)],
v3: [i64; 4] @ [node(val.v3)]
)
-> i64 @ [node(val.out)] impls select_sum,time,space {
        let sum1 : i64 @ node(val.sum1) = sum(v1);
        let condition : bool @ node(val.condition) = sum1 < 0;
        if @ [node(val.result)] condition {
                let sum2 : i64 @ [node(val.sum2)] = sum(v2);
                sum2
        }
        else {
                let sum3 : i64 @ [node(val.sum3)] = sum(v3);
                sum3
        }
}

fn select_sum_impl_2(
v1: [i64; 4] @ [node(val.v1)],
v2: [i64; 4] @ [node(val.v2)],
v3: [i64; 4] @ [node(val.v3)]
)
```

```
-> i64 @ [node(val.out)] impls select_sum,time,space {
        let sum2 : i64 @ [node(val.sum2)] = sum(v2);
        let sum3 : i64 @ [node(val.sum3)] = sum(v3);

        let sum1 : i64 @ node(val.sum1) = sum(v1);
        let condition : bool @ node(val.condition) = sum1 < 0;
        if @ [node(val.result)] condition {
                sum2
        }
        else {
                sum3
        }
}

pipeline main { select_sum_impl }
```

B.1.2 Typed Select Sum GPU

```
#version 0.1.0

sptl space(s: BufferSpace) -> BufferSpace { returns s }

feq sum {
        extern(cpu) pure sum_cpu(x : [i32; 4]) -> out: i32
        extern(gpu) sum_gpu(x : [i32; 4]) -> out: i32
        {
                path : "gpu_sum.comp",
                entry : "main",
                dimensions : 3,
                resource {
                        group : 0,
                        binding : 0,
                        input : x
                },
                resource {
                        group : 0,
                        binding : 1,
                        output : out
                }
        }
}

val select_sum(v1: [i32; 4], v2: [i32; 4], v3: [i32; 4]) -> out: i32 {
        sum1 :- sum(v1)
        sum2 :- sum(v2)
        sum3 :- sum(v3)
        condition :- sum1 < 0
        result :- sum2 if condition else sum3
        returns result
}

tmln single_sync(in: Event) -> out: Event {
        local, remote :- encode_event(in)
```

```
        sub :- submit_event(remote)
        sync :- sync_event(local, sub)
        returns sync
}

fn select_sum_impl(
v1: [i32; 4] @ [node(val.v1)],
v2: [i32; 4] @ [node(val.v2), node(tmln.in)],
v3: [i32; 4] @ [node(val.v3), node(tmln.in)]
)
-> i32 impls select_sum,single_sync,space {
        let sum1 @ node(val.sum1) = sum_cpu(v1);
        let condition = sum1 < 0;
        if condition {
                let encoder = encode-begin @ node(tmln.(local, remote)) { v2
                encode encoder.copy[v2_gpu @ [node(val.v2), node(tmln.remote
                encode encoder.call[v2_gpu_sum <- sum_gpu'<1, 1, 1>(v2_gpu)]
                let fence = submit @ node(tmln.sub) encoder;
                let result = await @ node(tmln.sync) fence;

                let v2_sum = result.v2_gpu_sum;
                v2_sum
        }
        else {
                let encoder = encode-begin @ node(tmln.(local, remote)) { v3
                encode encoder.copy[v3_gpu @ [node(val.v3), node(tmln.remote
                encode encoder.call[v3_gpu_sum <- sum_gpu'<1, 1, 1>(v3_gpu)]
                let fence = submit @ node(tmln.sub) encoder;
                let result = await @ node(tmln.sync) fence;

                let v3_sum = result.v3_gpu_sum;
                @out { input: node(tmln.out), output: node(tmln.out) };
                v3_sum
        }
}

pipeline main { select_sum_impl }
```

B.1.3 Caiman IR Examples

First, we show an example of straightforward series of external operations:

```
version 0.0.2

// implements:
// fn foo() -> i64 {
//     x = 3;
//     y = 2;
//     n1 = x + y;
//     n2 = x + n1;
//     return n1 + n2;
// }
```

```
ffi i64;
event %event0;
buffer_space %buffspace;
native_value %i64 : i64;

function @add(%i64, %i64) -> %i64;
function @main() -> %i64;

external-cpu-pure[impl @add] %add(i64, i64) -> i64;

value[impl default @main] %foo() -> [%out: %i64] {
    %res_t = call @add(%n1, %n2); // 8 + 5 = 13
    %res = extract %res_t 0;
    %n2_t = call @add(%x, %n1); // 3 + 5 = 8
    %n2 = extract %n2_t 0;
    %n1_t = call @add(%x, %y); // 3 + 2 = 5
    %n1 = extract %n1_t 0;
    %x = constant %i64 3;
    %y = constant %i64 2;
    return %res;
}

timeline %time(%e : %event0) -> %event0 {
    return %e;
}

spatial %space(%bs : %buffspace) -> %buffspace {
    return %bs;
}

schedule[value val = %foo, timeline time = %time, spatial space = %space]
%foo_main<time-usable, time-usable>() ->
[%out : val.%out-usable %i64] {
    %x_loc = alloc-temporary local [] i64;
    %y_loc = alloc-temporary local [] i64;
    %n1_loc = alloc-temporary local [] i64;
    %n2_loc = alloc-temporary local [] i64;
    %res_loc = alloc-temporary local [] i64;

    local-do-builtin val.%x() -> %x_loc;
    local-do-builtin val.%y() -> %y_loc;

    %x = read-ref i64 %x_loc;
    %y = read-ref i64 %y_loc;
    local-do-external %add val.%n1_t(%x, %y) -> %n1_loc;
    %n1 = read-ref i64 %n1_loc;
    local-do-external %add val.%n2_t(%x, %n1) -> %n2_loc;
    %n2 = read-ref i64 %n2_loc;
    local-do-external %add val.%res_t(%n1, %n2) -> %res_loc;

    %res_val = read-ref i64 %res_loc;
    return %res_val;
}

pipeline "main" = %foo_main;
```

Second, we show an example involving a call to the GPU:

```
version 0.0.2

// Performs a single computation on the GPU,
// encoding, submitting, and waiting all in one funclet.

ffi i32;
native_value %i32 : i32;
ref %i32l : i32-local<flags=[map_read, map_write,
  copy_src, copy_dst, storage]>;
ref %i32g : i32-gpu<flags=[map_read, map_write,
  copy_src, copy_dst, storage]>;
event %event0;
buffer %buffer_gpu : gpu<flags = [map_read, map_write,
  copy_src, copy_dst, storage], alignment_bits = 0, byte_size = 1024>;
buffer_space %buff_space;

function @simple(%i32) -> %i32;
function @foo(%i32) -> %i32;

external-gpu[impl @simple] %simple(%x : i32) -> [%out : i32]
{
    path : "gpu_external.comp",
    entry : "main",
    dimensionality : 3,
    resource {
        group : 0,
        binding : 0,
        input : %x
    },
    resource {
        group : 0,
        binding : 1,
        output : %out
    }
}

value[impl @foo] %foo(%x : %i32) -> %i32 {
    %c = constant %i32 1;
    %y_t = call @simple(%c, %c, %c, %x);
    %y = extract %y_t 0;
    return %y;
}

timeline %foo_time(%e : %event0) -> [%out: %event0] {
    %enc = encoding-event %e [];
    %enc1 = extract %enc 0;
    %enc2 = extract %enc 1;
    %sub = submission-event %enc2;
    %snc = synchronization-event %enc1 %sub;
    return %snc;
}

spatial %foo_space(%bs : %buff_space) -> %buff_space {
    return %bs;
}

schedule[value val = %foo,
```

```
  timeline time = %foo_time, spatial space = %foo_space]
%foo_main<time.%e-usable, time.%out-usable>
(%x_loc : val.%x-usable %i32l)
-> [%out : val.%y-usable %i32] {
    %c_loc = alloc-temporary local [storage] i32;
    %x_gpu = alloc-temporary gpu [storage, copy_dst] i32;
    %y_gpu = alloc-temporary gpu [storage, map_read] i32;
    %y_loc = alloc-temporary local [map_write] i32;

    local-do-builtin val.%c() -> %c_loc;
    %enc = begin-encoding gpu time.%enc [%x_gpu, %y_gpu] [];
    encode-copy %enc %x_loc -> %x_gpu;
    %c = read-ref i32 %c_loc;
    encode-do %enc %simple val.%y_t(%c, %c, %c, %x_gpu) -> %y_gpu;

    %fnc = submit %enc time.%sub;
    sync-fence %fnc time.%snc;

    local-copy %y_gpu -> %y_loc;
    %result = read-ref i32 %y_loc;
    return %result;
}

pipeline "main" = %foo_main;
```

which includes associated WGSL code:

```
#version 450

layout(set = 0, binding = 0) readonly buffer Input_0 {
    int field_0;
} input_0;

layout(set = 0, binding = 1) buffer Output_0 {
    int field_0;
} output_0;

layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
void main()
{
    output_0.field_0 = input_0.field_0 + 1;
}
```

B.1.4 Caiman IR Explicated Implementation

Partially example in the current working version of Caiman. Note that this example is not "maximally" explicated, but will hopefully illustrate what can be written as–is.

```
version 0.0.2
```

```
ffi i64;
ffi array<i64, 4>;
ref %i64l : i64-local<flags=[]>;
event %event0;
buffer_space %buffspace;
native_value %array4 : array<i64, 4>;
native_value %i64 : i64;

function @sum(%array4) -> %i64;
function @is_negative(%i64) -> %i64;
function @select_sum(%array4) -> %i64;

external-cpu-pure[impl @sum] %sum(array<i64, 4>) -> i64;
external-cpu-pure[impl @is_negative] %is_negative(i64) -> i64;

value[impl default @select_sum] %main(
  %v1 : %array4, %v2 : %array4, %v3 : %array4) -> [%out : %i64] {
    %res = select %sel %left %right;
    return %res;

    %s_t = call @sum(%v1);
    %s = extract %s_t 0;
    %sel_t = call @is_negative(%s);
    %sel = extract %sel_t 0;

    %left_t = call @sum(%v2);
    %left = extract %left_t 0;
    %right_t = call @sum(%v3);
    %right = extract %right_t 0;
}

timeline %time(%e : %event0) -> %event0 {
    return %e;
}

spatial %space(%bs : %buffspace) -> %buffspace {
    return %bs;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_head<time-usable, time-usable>(
    %v1 : val.%v1-usable time-usable space-usable %array4,
    %v2 : val.%v2-usable time-usable space-usable %array4,
    %v3 : val.%v3-usable time-usable space-usable %array4
) ->
    val.%out-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum ? ? -> ?;
    %_ = read-ref i64 ?;

    local-do-external %is_negative ? ? -> ?;
    %sel = read-ref i64 ?;

    %djoin = default-join;
    %join = inline-join %select_sum_join [] %djoin;
```

```
    schedule-select %sel
        [%select_sum_left, %select_sum_right]
        [val.%res, time, space]
        (%v2, %v3)
        %join;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_left<time-usable, time-usable>(
    %v2 : phi-val.%v2-usable time-usable space-usable %array4,
    %v3 : phi-val.%v3-usable time-usable space-usable %array4
) ->
    val.%left-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum ? ? -> ?;
    %_ = read-ref ? ?;
    return ?;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_right<time-usable, time-usable>(
    %v2 : phi-val.%v2-usable time-usable space-usable %array4,
    %v3 : phi-val.%v3-usable time-usable space-usable %array4
) ->
    val.%right-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum val.%right_t ? -> ?;
    %_ = read-ref ? ?;
    return ?;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_join<time-usable, time-usable>(
    %res : val.%res-usable time-usable space-usable %i64
) ->
    val.%out-usable time-usable space-usable %i64
{
    return ?;
}

pipeline "main" = %select_sum_head;
```

B.1.5 Caiman Frontend Examples

The following examples all compile and run as expected in the current build of Caiman.

Nested conditional logic:

```
#version 0.1.0

tmln time(e: Event) -> Event { returns e }
sptl space(bs: BufferSpace) -> BufferSpace { returns bs }


val main() -> i64 {
    b :- true
    c :- false
    d :- false
    one :- 1
    two :- 2
    three :- 3
    four :- 4
    left :- one if b else two
    right :- three if c else four
    z :- left if d else right
    returns z
}

fn foo() -> i64
    impls main, time, space
{
    let d = false;
    var v;
    if d {
        let b = true;
        let two = 2;
        v = two;
        if b {
            let one = 1;
            v = one;
        }
    } else {
        let c = false;
        let four = 4;
        v = four;
        if c {
            let three = 3;
            v = three;
        }
    }
    v
}

pipeline main { foo }
```

Recursion:

```
#version 0.1.0

tmln time(e: Event) -> Event { returns e }
sptl space(s: BufferSpace) -> BufferSpace { returns s }

val gcd(a: i64, b: i64) -> i64 {
    returns a if b == 0
            else gcd(b, a % b)
}

fn gcd_impl(a: i64, b: i64) -> i64 impls gcd, time, space {
    if b == 0 {
        a
    } else {
        gcd_impl(b, a % b)
    }
}

pipeline main { gcd_impl }
```

WGSL Function Calls:

```
#version 0.1.0

extern(cpu) pure baz() -> i32
extern(cpu) pure bar() -> i32
extern(gpu) gpu_merge(x : i32, y: i32) -> out: i32
{
    path : "gpu_merge.comp",
    entry : "main",
    dimensions : 3,
    resource {
        group : 0,
        binding : 0,
        input : x
    },
    resource {
        group : 0,
        binding : 1,
        input : y
    },
    resource {
        group : 0,
        binding : 2,
        output : out
    }
}

val foo(c: bool) -> out: i32 {
    a :- baz()
    b :- bar()

    snd :- a if c else b

    r :- gpu_merge'<1, 1, 1>(a, snd)
    returns r
}
```

```
tmln foo_time(e: Event) -> out: Event {
    loc, rem :- encode_event(e)
    sub :- submit_event(rem)
    snc :- sync_event(loc, sub)
    returns snc
}

sptl foo_space(bs: BufferSpace) -> BufferSpace {
    returns bs
}

fn foo_impl(c: bool) -> i32 impls foo, foo_time, foo_space {
    @in {input: input(tmln.e), output: node(tmln.out) };
    let a = baz();
    let b = bar();
    let e = encode-begin @ node(tmln.(loc, rem))
      { a_gpu, b_gpu, y_gpu } gpu;
    encode e.copy[a_gpu <- a];
    if c {
        @in { input: node(tmln.loc), output: node(tmln.loc),
            a: node(tmln.loc), b: node(tmln.loc), e: node(tmln.rem),
            a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
              y_gpu: node(tmln.rem) };
        encode e.copy[b_gpu <- a];
    } else {
        @in { input: node(tmln.loc), output: node(tmln.loc),
            a: node(tmln.loc), b: node(tmln.loc), e: node(tmln.rem),
            a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
              y_gpu: node(tmln.rem) };
        encode e.copy[b_gpu <- b];
    }
    @in { input: node(tmln.loc), output: node(tmln.out),
        e: node(tmln.rem),
        a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
          y_gpu: node(tmln.rem) };
    encode e.call[y_gpu <- gpu_merge'<1, 1, 1>(a_gpu, b_gpu)];
    let f = submit @ node(tmln.sub) e;
    let r = await @ node(tmln.snc) f;
    @out { input: node(tmln.out), output: node(tmln.out) };
    r.y_gpu
}

pipeline main { foo_impl }
```

Fixed–size arrays for select sum:

```
#version 0.1.0

feq sum {
    extern(cpu) pure sum1([i64; 4]) -> i64
    extern(cpu) pure sum2([i64; 4]) -> i64
}

val select_sum(v1: [i64; 4], v2: [i64; 4], v3: [i64; 4]) -> out: i64 {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
```

```
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- sum2 if condition else sum3
    returns result
}

sptl space(s: BufferSpace) -> BufferSpace { returns s }

tmln time(e: Event) -> Event { returns e }

fn select_sum_impl(
    v1: [i64; 4],
    v2: [i64; 4],
    v3: [i64; 4]
)
-> i64 impls select_sum,time,space {
    if sum1(v1) < 0 {
        sum2(v2)
    }
    else {
        sum2(v3)
    }
}

pipeline main { select_sum_impl }
```

BIBLIOGRAPHY

[1] Frink. http://frinklang.org/#Features. Accessed: 2020-08-10.

[2] Khronos Vulkan registry. https://www.khronos.org/registry/vulkan/.

[3] WGPU, 2019. https://github.com/gfx-rs/wgpu.

[4] Bytecode Alliance. Cranelift, 2016. https://cranelift.dev/.

[5] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[6] Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[7] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[8] Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. 2011.

[9] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. 1999.

[10] 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. 2019.

[11] Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. Geometry types for graphics programming. In *OOPSLA*, 2020.

[12] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. 1990.

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

[14] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[15] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 703–718, New York, NY, USA, 2022. Association for Computing Machinery.

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

[17] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4:77–84, 1975.

[18] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, CEFP'09, page 268–305, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] Andrew J. Kennedy. Dimension types. 1994.

[20] Andrew J. Kennedy. Relational parametricity and units of measure. 1997.

[21] Khronos. Opencl, 2009. `https://www.khronos.org/opencl/`.

[22] Khronos. Sycl, 2014. `https://www.khronos.org/sycl/`.

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

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[25] W. B. Langdon and M. Harman. Evolving a cuda kernel from an nvidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on*

Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[27] Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*, pages 482–494. ACM, June 2017.

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

[29] Mozilla. WebGL, 2011. `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`.

[30] Kazuo Murota. Homotopy base of an acyclic graph—a combinatorial analysis of commutative diagrams by means of preordered matroid. *Discrete Appl. Math.*, 17(1–2):135–155, May 1987.

[31] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. 2018.

[32] NVIDIA. Cuda programming guide, 2009. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[33] Jiawei Ou and Fabio Pellacini. SafeGI: Type checking to improve correctness in rendering system implementation. 2010.

[34] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, 2022.

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

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[37] Adrian Sampson. Let's Fix OpenGL. 2017.

[38] Adrian Sampson, Kathryn S McKinley, and Todd Mytkowicz. Static stages for heterogeneous programming. 2017.

[39] Edmond Schonberg and Vincent Pucci. Implementation of a simple dimensionality checking system in ada 2012. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, HILT '12, page 35–42, New York, NY, USA, 2012. Association for Computing Machinery.

[40] 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.

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

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

[43] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 86–97, New York, NY, USA, 2010. Association for Computing Machinery.