LANGUAGE DESIGN FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Dietrich Geisler

August 2024

LANGUAGE DESIGN FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

Dietrich Geisler, Ph.D.

Cornell University 2024

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

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

BIOGRAPHICAL SKETCH

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

This document is dedicated to all Cornell graduate students.

ACKNOWLEDGEMENTS

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

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi
List of Tables . . . viii
List of Figures . . . ix

1 Introduction 1

2 Gator: Geometry Types for Graphics Programming 2

3 Online Verification of Commutativity 3

4 Caiman: DSL for Optimizing Heterogeneous Program Communication (Caiman) 4

4.1 Introduction . . . 4
4.1.1 Separation of Concerns . . . 4
4.1.2 Combinatoric Explosion . . . 6
4.2 Background . . . 6
4.2.1 Submitting to the GPU . . . 6
4.2.2 Promises . . . 6
4.3 Practical Caiman . . . 6
4.3.1 Value Specification . . . 7
4.3.2 Implementation Language . . . 11
4.3.3 Timeline and Spatial Specifications . . . 15
4.3.4 Select Sum on the GPU . . . 18
4.4 Formal Model . . . 22
4.4.1 Typing Semantics . . . 25
4.4.2 Operational Semantics . . . 26
4.5 Explication . . . 27
4.5.1 Using Explication . . . 27
4.5.2 Caiman IR . . . 27
4.5.3 Core Algorithm . . . 27
4.5.4 Controlling the Explicator . . . 27
4.6 Caiman Engineering . . . 27
4.6.1 WebGPU Target and Codegen . . . 27
4.6.2 Stages of Compilation . . . 27
4.7 Results . . . 27
4.7.1 Timing . . . 27
4.7.2 Explication . . . 27
4.8 Conclusion . . . 27
4.8.1 Future Work . . . 27

A Chapter 1 of appendix **28**

Bibliography **29**

LIST OF TABLES

LIST OF FIGURES

4.1 Hatchling Specification Syntax 22
4.2 Hatchling Implementation Syntax 22
4.3 Hatchling Specification Typing Judgment 25
4.4 Hatchling Implementation Typing Judgment 26

CHAPTER 1

INTRODUCTION

CHAPTER 2

GATOR: GEOMETRY TYPES FOR GRAPHICS PROGRAMMING

CHAPTER 3

ONLINE VERIFICATION OF COMMUTATIVITY

CHAPTER 4

CAIMAN: DSL FOR OPTIMIZING HETEROGENEOUS PROGRAM COMMUNICATION (CAIMAN)

4.1 Introduction

GPUs are hard to program I guess.

4.1.1 Separation of Concerns

Our running example throughout this paper will be used to illustrate the performance trade–offs inherent in conditional logic and device synchronization. Specifically, we will be building the `select_sum` function, which takes in three arrays ‘v1‘, ‘v2‘, and ‘v3‘, and returns the sum of either ‘v2‘ or ‘v3‘ depending on the sign of ‘v1‘.

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

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

This code is fairly straightforward to understand, and has the significant upside of having its behavior exactly dependent on the abstraction of using an existing (working) function. This approach, however, is not necessarily performant.

While we can often rely on an optimizing compiler to work out a reasonably fast arrangement of this code, the compiler cannot always assume the *intent* of the programmer, restricting potential optimizations. As a result, the programmer may need to rewrite this code to change performance, but without changing the semantic meaning of this code:

```
select_sum(v1 : [i64; N], v2 : [i64; N], v3 : [i64; N]) -> i64 {
  int sum2 = sum(v2);
  int sum3 = sum(v3);
  if (sum(v1) < 0) {
    return sum2;
  }
  else {
    return sum3;
  }
}
```

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

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

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

4.1.2 Combinatoric Explosion

4.2 Background

4.2.1 Submitting to the GPU

4.2.2 Promises

CUDA, SYCL

Halide, Scheduling

4.3 Practical Caiman

To explain how Caiman works, we will work through the example shown in the introduction, namely `select_sum`. For this illustration, we will be using code written the user-facing Caiman frontend, as opposed to the Caiman IR described in more detail in 4.6.

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

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

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

4.3.1 Value Specification

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

```
// function header, any number of arguments
spec_kind name(arg : type) -> return_type {
  var :- expression
  returns var
}
```

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

```
// function call
fn_name(expr)
// the usual notion of conditions
```

```
if cond then expr1 else expr2
```

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

Our value specification for `select_sum` is as follows:

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

We have written this specification to be more verbose than needed for the sake of providing a more detailed examination of the semantics being used here. Many of these lines can be condensed or combined (we can just write `if sum(v1) < 0 ...`, for example).

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

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

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

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

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

Function Equivalence

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

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

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

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

Note that select_sum must also be in such a function equivalence class – syntactically we name this equivalence class to be the same as our defined value function, in this case just select_sum. In this way, Caiman specification functions can call other specification functions without introducing assumptions about the implementations of those functions.

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

4.3.2 Implementation Language

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

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

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

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

```
fn select_sum_impl(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3)
  -> i64 @ node(val.result)
  impls select_sum,... {
    let s1 : i64 @ node(val.sum1) = sum_cpu(v1);
    let cond : bool @ node(val.condition) = s1 < 0;
    let res : @ node(val.result) = if cond {
        let s2 : i64 @ node(val.sum2) = sum_cpu(v2);
        s2
    }
    else {
        let s3 : i64 @ node(val.sum3) = sum_cpu(v3);
        s3
    };
    res
}
```

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

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

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

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

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

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

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

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

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

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

We can now address the notion of actually using function equivalence in our implementation. Unfortunately, we are not able to yet write our call to `sum_gpu` (as alluded to in 4.3.1), since we will first need to specify more details about synchronizing to another device (the GPU), and we have so far assumed that the CPU is our local host and so does not need synchronization.

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

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

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

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

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

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

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

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

4.3.3 Timeline and Spatial Specifications

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

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

We capture the intent of the value specification language in Caiman as reasoning about the data produced and used by our computation. When we are working with another device (such as a GPU), however, it is also important to be precise about what threading and memory resources we interact with. More specifically, we would like to be able to break apart synchronization and memory resources across control flow, in much the same way as the value language allows us to (safely) reuse data and decompose computation on that data into carefully designed pieces.

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

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

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

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

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

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

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

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

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

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

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

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

4.3.4 Select Sum on the GPU

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

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

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

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

Second, we provide a timeline specification for a *single* synchronization (since we only call the gpu once in each branch, we need only to synchronize once). As a result, our specification will simply lay out our four stages in sequence:

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

Third, we provide a (trivial) spatial specification:

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

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

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

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

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

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

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

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

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

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

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

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

    select_sum_impl_gpu_2_inter()
}
fn select_sum_impl_gpu_2_inter(
  v1: [i64; N] @ val.v1,
  s2: i64 @ val.sum2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.result
  impls select_sum,single_sync,trivial_spatial {
    // similar logic to compute s3
    select_sum_impl_gpu_2_ret(v1, s2, s3)
}
fn select_sum_impl_gpu_2_ret(
  v1: [i64; N],
  s2: i64 @ val.sum2,
  s3: i64 @ val.sum2)
  -> i64 @ val.result
  impls select_sum,trivial_timeline,trivial_spatial {
    if sum_cpu(v1) < 0 {
        s2
    }
    else {
        s3
    }
}
```

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

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

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

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

Figure 4.1: Hatchling Specification Syntax

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

Figure 4.2: Hatchling Implementation Syntax

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

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

4.4 Formal Model

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

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

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

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

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

As an observation about this proof, the term *equivalent outputs* relies on a precise notion of equivalence and values in this context. Specifically, equivalence is defined exactly as any function that is a member of its function class (as described in subsection 4.3.1), while value refers to the series of operations that define some result. In this sense, we observe that the operation $1 + 1$ is not considered to be the same value as the constant 2, unless an equivalence is explicitly stated.

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

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

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

A Hatchling implementation program, on the other hand, similarly requires only 1 dynamic context, but requires 3 constant global contexts (where Δ and Ψ are built to be derived from the specification associated with the current implementation). These contexts are as follows:

- $\Gamma : \mathbf{V} \mapsto \mathbf{T} \bigtimes (\mathbf{S} \cup \top)$, the sole dynamic context, which maps from an implementation pointer to the type of the data being referred to by that pointer. This context also includes the associated value if one has been written to the location pointed at by this variable.

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

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

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

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

Figure 4.3: Hatchling Specification Typing Judgment

- $\Psi : \mathbf{S} \mapsto \mathbf{T} \times (\mathbf{C} \times (\mathbf{S} \times \mathbf{S} \times \mathbf{S}))$, which must be derived from the result of applying the typechecking rules to a well-typed Hatchling specification. When typechecking an implementation, however, Ψ is global and constant.
- $\Phi : \mathbf{C} \mapsto \mathbf{T} \times \mathbf{T}$, which is the same as with the specification program
- $\Xi : \mathbf{F} \mapsto \mathbf{C}$, which maps from functions to their owning function class

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

4.4.1 Typing Semantics

[TODO: If I cut the alloc and ref, then this is essentially just done I think —DG] xxx

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

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

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

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

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

Figure 4.4: Hatchling Implementation Typing Judgment

4.4.2 Operational Semantics

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

4.5 Explication

4.5.1 Using Explication

4.5.2 Caiman IR

4.5.3 Core Algorithm

4.5.4 Controlling the Explicator

4.6 Caiman Engineering

4.6.1 WebGPU Target and Codegen

4.6.2 Stages of Compilation

4.7 Results

4.7.1 Timing

4.7.2 Explication

4.8 Conclusion

4.8.1 Future Work

APPENDIX A

CHAPTER 1 OF APPENDIX

Appendix chapter 1 text goes here

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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