DESIGNING LANGUAGES FOR PARALLEL PORTABILITY OF PHYSICAL SIMULATIONS, USING RELATIONAL ALGEBRAIC ABSTRACTIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Gilbert Louis Bernstein
June 2019

Abstract

The desire to ever more efficiently simulate physical phenomena using computers produces a demand for increased specialization of physical models, geometric & numeric algorithms, and computing hardware/systems. This demand limits itself by trending towards unmanagable complexity; manifesting as an inability to separate disciplinary specialties via distinct parts of the simulation code. For parallelism in particular, the interface along which this separation happens must necessarily take the form of a language, separating compiler-programmers from simulation-programmers. Furthermore, such languages must exploit a more abstract and restricted data model than random-access-memory in order to enable performant portability across different parallel systems.

I propose using a relational (algebraic) abstraction of simulation data in order to give compiler programmers both sufficient freedom and information to successfully port applications onto parallel hardware, while maintaining high performance (time and memory). I investigate three aspects of this problem, organized by the metric-topology distinction from geometry, and explored by constructing language prototypes: (1) Ebb, a language for local stencil computations on metric data; (2) Seam, a language for local modification of topological data (i.e. remeshing); and (3) Gong, a language for generation of topology from metric locality (i.e. collision detection). These languages use a common relational abstraction of data, including coupling of structured and unstructured mesh domains in a single program. Seam additionally leverages the concepts of transactions and views from databases, while Gong leverages the concepts of joins and indices. Database queries are extended by treating actions on queries using parallelizable lock-free effects beyond simple reading and writing.

Experimental evaluation of prototype language implementations (with support for GPUs) shows that the performance of highly-tuned implementations can be matched to within a 1.25× overhead at worst (for an Ebb implementation of a super-computing benchmark; 1.15× for a Gong implementation of an industrial collision detection system). However, the ability to systematically port code to different platforms easily allows $2-9\times$ accelerations of many simulation programs over the next best cpu/multi-core implementations available. Ebb programs are usually less than half as long, while automatic view maintenance in Seam yields 10× reductions in complex pointer manipulation code size.

Acknowledgements

Pat Hanrahan for believing in me and helping me transfer in the middle of my PhD.

Phil Levis for lending an ear and research advice.

Alex Aiken for help branching out into programming languages research.

Gianluca Iaccarino for somehow making every meeting he's at a joy.

Doug James for conversations about simulation practice.

Chris Re for conversations about databases.

Zoran Popovic for encouraging me to explore research outside of my comfort zone.

Clark Barrett for help with SMT solvers on the Seam project.

Wilmot Li for a productive, exciting internship.

Jovan Popovic for conversations and advice.

Lorenzo Alvisi for encouragement and affirming the value of academic scholarship.

Don Fussell for being the best undergraduate advisor one could hope for.

Calvin Lin for encouraging me to take a Parallel Systems course before I had any idea what I was doing.

David Notkin for being a mensch and source of encouragement while I started my PhD at UW.

Mike Ernst for his patience and for teaching me so much of what I know about formal methods.

Dan Grossman for his humor and recommendation to learn about graph rewriting, which was formative on the Seam project.

Chris Wojtan for the puns.

Warren Hunt for conversations about rendering and joins.

Jonathan Ragan Kelley for the post-doc.

Peter Djeu for extolling the virtues of video-games and lunches as fundamental units of economic measure.
Michael Mara for picking up where I left off.
Zach Devito for developing Terra, and for invaluable conversations about compiler construction.
Lenny Truong for teaching me about mentorship.
Crystal Lemire for being a great labmate and collaborator.
Chinmayee Shah for sticking together on a challenging project.
Matt Fisher for welcoming me to the lab.
Mike Roberts for the compliments on just that one presentation.
Niels Joubert for the chainsaw.
Katherine Breeden for the bowtie advice.
Daniel Ritchie for the T-shirt design.
Manolis Savva and Angel Chang for the collaboration and talking to the lawyers.
Manolis Papadakis for the whiteboard chats.
My parents, Marion and Henry without whom I literally wouldn't be here.
My brother David for the spatulas.
Chuck Teller, Betsy Teller, Ruth Pleaner and Allan Pleaner for seder, and for being inspirations.
My grandmother Charlotte Hattenbach for the mandel bread.
My grandfather Walter Hattenbach for the cinnamon toast.
Jerry Teller for the card trick.
and the rest of the family for everything...
Maggie Zhou for the shampoo.
Matt Kehrt and Jacob Nelson for dinner.
Justine Sherry for the bolognese.
Amit Levy for the couch.
Justin Hilburn for the good eats.
Scott Kilpatrick for some logical perspectives.
Joe Reisinger and Yonatan Bisk for the interesting summer.
Mark Reitblatt for the leg.

Buro Mookerji for the squids.

Sarah Burton, Diyang Tang, and Katherine Elliott for the noodles.

Gilbert Slade for going by Gil.

Miriam Estrin for the recommendation.

Nanci Temple-Armstrong for getting us out.

Urvi Nagrani for karaoke.

Tim Hwang for the wrestling.

Lea Rosen for sharing my skepticism.

Borderlands (esp. Z'ev, Alan, and Jude) for the place to write.

And to many others, for all the rest.

Contents

List of Tables

List of Figures

Chapter 1

The Simulation Expression Problem

1.1 The Problem

Simulations of the physical world are inherently expensive computations. Consider a fluid simulation, discretized on a 3-d grid, and integrated using an explicit Euler scheme. Each timestep of this simulation requires computing the new velocity at each grid cell by evaluating an arithmetic function of velocities at neighboring cells (often referred to as a stencil computation). The accuracy of this simulation is limited by its resolution, in space and time. In order to double the resolution (in a single spatial dimension) we must use $2^3 = 8$ times as many grid cells to represent the same volume of space. Less obviously, the timestep length must also be halved in order to maintain the same error bounds, requiring twice as much computation to simulate the same duration of time. Summarizing this line of reasoning, in the general case simulations of the physical world require $O(n^3)$ memory-space and $O(n^4)$ computation-time as the resolution n increases.

Because of these asymptotics, simulations of the physical world can only be made faster and more efficient by specialization to take advantage of phenomenon-specific approximations, of appropriate geometric discretizations, of advanced numerical methods, and of more complex kinds of parallel hardware. While these different specializations are often interdependent (hardware-specific algorithms, numerical techniques for specific discretizations and so on) the relevant methods and expertise are just as often developed independently in different disciplines.

In the source code for simulations, these different disciplinary specialties get slopped together as a tangle of spaghetti code. In order to port a fluid-chemical simulation to a super-computer, a high-performance computing specialist might duplicate, and modify the inner loops of a program, introducing assembly intrinsics. But then if a chemist wants to change the physical model, they must further copy and/or modify these intrinsics whose purpose they have only a loose understanding of. If a numerical analyst then wishes to change the method of integration, they must again transform (and further complicate) this same code.

Repeated modification, adaptation and maintenance of simulation programs results in code that is incomprehensible to anyone insufficiently educated in all the relevant specialities; incomprehensible to anyone who is not a *super-expert*. For instance, the Chemistry professor James Sutherland[1] explained to me and a room of super-computing experts that he must teach his students not only physics, but numerical analysis and high performance computing before they are able to perform any research in his lab. This imperative to teach students a little bit of everything drives a regression towards a least-common-denominator of teachable techaniques across and within professional communities.

Society (whether in formal education or on-the-job) can't really produce super-experts, so instead programs (software code) and programs (training of experts) undergo a mutually reinforcing pressure that inhibits effective specialization across communities[2]. It is my view that these dynamics are the primary limit on the efficiency and utility of physical simulation on average, across applications.

[1]At a supercomputing tools conference called WEST hosted in Albuquerque during February 2016.

[2]It is important to note that many simulation application communities *do* specialize. For instance, the rendering community within Computer Graphics has hyper-specialized around the problem of

I refer to this set of reinforcing problems as the *Simulation Expression Problem* in reference to the classic Expression Problem from the design of programming languages [Wad98]. The classic expression problem considers the multiplicative interaction in a compiler implementation between more types of AST nodes (loops, branches, operators, &c.) and more functions (print, compile, &c.) defined on each node. The expressivity of language features like sub-classing (object-oriented) and type-classing (functional) can then be evaluated by how they *refactor* such code to better modularize and asymptotically curb the amount of code that needs to be specified.

Analogously, the simulation expression problem is how to achieve a separation of concerns in simulation source code, such that programs are better modularized, code growth is asymptotically curbed, and programmers can focus on their specialty. Unlike Wadler's formulation of the expression problem, I have not explicitly specified which concerns must be separated. My criterion is simply that disciplinary specialties[3] must be separated.

1.2 Parallelism

Because all aspects of a simulation program, including the choice of physical model/equations can be changed to improve performance (this was our initial analysis),

light-transport. Largely, the graphics community is uninterested in other kinds of radiative/light-transfer simulation (other applications) and those other physicists/engineers who work on light-transport are unaware of more specialized techniques developed within the graphics community. In this sense, there is a failure of specialization. Work and expertise is unevenly duplicated across these divides in both code-bases and training programs; this is both redundant, and incomplete. Likewise, the replication of work on geometric intersections and spatial joins across graphics, robotics, computational geometry, and databases (GIS) provides a prime case study of this phenomenon. To no avail, members of the Computational Geometry community identified this problem in the 90s[Cha96].

[3]This means that the Simulation Expression Problem is inherently a social problem. Most design problems, and most certainly most questions of how to organize and design software are social. For instance, Conway's Law states that "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." That Computer Science as a discipline chooses to address problems as technical, objective, and scientific (to the exclusion of social dimensions) makes this no less true. However, in keeping with the discipline, my goal will be to reduce—to the greatest degree possible—some aspect of this problem to those (technical, objective, scientific) terms.

we should accept that it is impossible to find a separation between performance as a general concern and any other part of the simulation. What is possible is to separate out particular kinds of performance considerations.

Changing programs to run on different kinds of parallel hardware is one such potentially separable concern. Unfortunately despite four decades of research, parallelizing a program (in the general case) means rewriting the program for each new kind of parallel hardware. In this way, if I have $O(n)$ programs/applications and $O(m)$ distinct kinds of parallel hardware, I will end up writing and maintaining $O(nm)$ distinct pieces of code. (or incurring the opportunity cost of not using some kinds of hardware for some applications)

The demand for parallel computing has been increasing over the last decade, due to fundamental trends in computer hardware. Since CPU clock speeds maxed out around 2006[4][Rup19], chip manufacturers have shifted to more explicit parallelism via multi-core architectures and SIMD vector instructions. At the same time, the alternate architecture provided by GPUs has allowed for even more operations to be performed by a given processor. Further physical limitations[EBA+11] suggest further avenues for parallelism, in addition to even more complex architectures (e.g. FPGAs and custom ASICs) being exposed to the hardware-software interface.

Still, no common, general purpose (i.e. for all kinds of programs) abstraction of multi-core CPUs and GPUs (much less other architectures) appears to exist.

To see why, consider a simulation program written in a common high performance computing language like Fortran. Data is expressed as large arrays that reference each other and store simulation quantities[5]. Then, loop-heavy programs iterate over the data in different ways, reading values, performing arithmetic, and writing values back to arrays.

Consider a loop and the question "is it ok to parallelize this loop?" as well as the question "will this loop be parallelized?" The first question is asked by a compiler programmer, who must necessarily turn their attention to *all* valid programs. The compiler programmer must reason about all programs that *might* be written. The

[4]This is due largely to hitting the Dennard scaling power-wall.

[5]In C or C++ variations on this theme, pointers might be used instead of array indices—to much the same effect

second question is asked by an application programmer, who is trying to predict the performance implications of their coding decisions when writing a *specific* application, for translation to at least one, but preferrably multiple hardware platforms. (Of course, this issue also appears more generally than just for loops.)

One approach to parallel abstractions is to *automatically parallelize* programs written in sequential languages by placing the entire burden on the compiler programmer to develop sophisticated analyses[6]. That is, the compiler tries to figure out which loops are safe to parallelize and then choose which to parallelize. This approach leads to a communication breakdown on both sides. The compiler writer is given the Sisyphean task of determining safety. The application programmer cannot predict whether or not their code will be parallelized, and how well. It should not be surprising that this kind of arrangement leads to a breakdown of trust between these two groups—hence a failure of the abstraction[KKZ07].

Another approach is to not abstract, but require *explicit parallelization* of applications. Every loop must be declared sequential or parallel, including how to break up that parallelism. In effect, this is a non-solution. However, this approach does greatly simplify the burden on the compiler programmer. Just, it does so by placing all responsibility for deciding both how to parallelize and whether it is safe to parallelize in a particular way onto the application programmer. As a result, the application programmer can now introduce arbitrary data races.

One major problem with incorrect (explicit) parallel programs arises when attempting to port existing programs to new hardware. For instance, changes to NVIDIA's recent GPUs exposed subtle issues with the not totally synchronous behavior of warp-level (32-wide "SIMD") instructions, which required fixes in CUDA 9[LG]. In fact, I had to fix bugs arising from this issue during software development for Gong (§6.4.2). In a sense, the tables are turned from automatic parallelism. By placing the responsibility for determining when parallelism is safe with the application programmer, application programmers often come to rely on unspecified behavior that just happens to be the case for the hardware they are currently targeting.

Often programmers argue for creating *expressive* abstractions, with the implied

[6]specifically, this will end up relying on aliasing analysis, which is well-known to be undecidable.

meaning that the *user* of the abstraction is allowed to write a wider range of programs. Unfortunately, this *positive expressiveness* also entails a *negative expressiveness* imposed on the *implementer* of that abstraction. An abstraction for parallelism must reserve the expressive freedom of the implementer to change the execution order of code, and consequently restrict the expressive freedom of the user in what programs they may write. Lastly, this division, or separation of concerns implied by the abstraction must be evident and clearly understood by both the user and implementer. If not, the prior problems are likely to recur.

Now it should be more clear why there are no general purpose abstractions of parallel hardware. There is a basic tradeoff between application generality and generality across parallel hardware[7]. It should come as no surprise then, that restricting the *domain of application* has proven to be a promising approach to developing abstractions for parallelism.

1.3 Domain-Specific-Languages

Rather than trying to abstract across specific kinds of parallel hardware, we can try to abstract across parallelism in specific kinds of applications[8]. Within Computer Graphics and Databases, this strategy has already been effective.

[7]It's worth noting that this argument does not preclude some technically non-general, but pragmatically general abstraction from being discovered, any more than pragmatically general solutions to the halting problem can be discovered. However, in that vein it is worth noting that SMT solvers suffer from problems directly analogous to those I describe under *automatic parallelism.* There are no general rules or guidance that SMT implementers have for users to figure out why certain inputs take a long time or fail to terminate.

[8]Exploring this inversion was the primary approach of Stanford's Pervasive Parallelism Lab (PPL) from 2009-2014. Presentations from PPL members often included a triangle with "performance, productivity, and generality" used as a taxonomy with the claim: you can only have two[Olu11]. The diagram was then illustrated by assigning languages to edges of the triangle. However, this triangle point of view is misleading. My argument for specialization *in language* or *in abstraction* instead follows from a broader strategy of specialization—as necessary for performance. This need for specialization holds independently of trade-offs between performance and productivity, which always exist because optimization always requires more programmer time and effort. Perhaps the slogan ought to be "performance, productivity, generality: pick one!" Trying to classify languages in this way, (as if variation due to programmer or application is negligible) is a conceptual dead end. Asking how a language separates concerns instead pre-supposes variation in programmers (i.e. by discipline) and variation in application.

Shading languages[HL90, PMTH01] are used to specify material properties and lighting models in 3-d rendering. These languages have always been coupled into larger rendering systems/engines, which themselves have often taken advantage of specialized hardware. If we further consider that rendering APIs/libaries, like OpenGL or DirectX are themselves DSLs[9], then the utility of these DSLs in porting programs across different hardware is apparent. GPUs initially started out as almost entirely fixed-function hardware, developed programmable components, and are now being specialized in new ways for neural networks and raytracing. Still, games and applications written over two decades ago can be executed on new hardware, taking advantage of all the different kinds of parallelism on GPUs.

Another recent line of work on image processing DSLs[RKAP$^+$12, RKBA$^+$13, HBD$^+$14] shows similar promise. Halide has managed to subsume a wide variety of different processors and effectively specialize to idiosyncratic characteristics of their memory/cache systems. Using similar abstractions Darkroom allows for writing image processing programs that can be loaded onto FPGAs or taped out as custom hardware.

These image processing languages bear a notable similarity to ZPL[Cha01], a language from the 90s focused on a class of simulation programs built around "stencils". Stencils are local neighborhoods in grids, from which data can be computed. ZPL was built around *data-parallel* operations (add this entire multi-dimensional array to this entire other multi-dimensional array) and a language of offsets and subsets (interior, boundary, &c.) of grids and sub-grids. Unlike more general purpose parallel programming languages, ZPL was specialized only to these stencil computations.

Outside of computer graphics, SQL (Structured Query Language) has proven to be a surprisingly stable and effective abstraction for databases over the last 40 years. SQL is a domain specific language—maybe one of the most important ones to have ever been developed. While originally designed for magnetic drum and tape storage media accessed by sequential processors, the same abstraction has now been extended to run on geographically networked machines, with multiple processors, accessing data

[9]The argument that OpenGL, not just shading languages, is a DSL was first made by Pat Hanrahan circa 2010 to the best of my knowledge

stored on powered memory, and replicated to handle partial system failures seamlessly.

As suggested, all of these DSLs achieve flexibility in implementation by restricting the flexibility of what kinds of programs may be written in them. What is less obvious is how (on what criterion) they manage to effectively abstract applications. SQL is instructive in this regard. It abstracts not just code, but also the data on which that code is executing.

If we turn our attention to the abstraction of data, we can see that all of the DSLs we just considered replace a random-access-memory model with a domain-specific data model. In the image processing languages, this is grids (multi-dimensional arrays) of numbers. In the case of shading languages, this is triangle lists, fragments, textures, and rays. In the case of SQL, the abstraction is relations—a set of structured tuples.

Along with a specialized data model, these languages restrict the kinds of programs that can be written on that data model. The image processing languages impose locality by restricting computations to stencils. Shading languages restrict programs to specifying only the interaction of individual primitives (e.g. how to shade a single fragment alone). Rendering Pipeline languages (and hardware-targeting image languages) impose streaming constraints by restricting the temporal window of computation. Lastly, SQL restricts computations on sets to *set semantics* which requires that computations are insensitive to the order of relation elements.

This last concept of order insensitivity is especially important for parallelism. The ability to commute two operations is a classic stand-in for the ability to execute the operations in parallel[Ber66][10]. Similarly, a function defined on a set properly, not just a list or some other representation of a set, must be invariant/symmetric with respect to the ordering of elements. This property implies potential data-parallelism. It is this focus on set-based operations that underlies the unanticipated success of porting SQL onto new parallel hardware[Hel16].

To put this in other terms, the criterion which allows DSLs to be parallelized

[10]Depending on the specific formulation, simple associativity of operations, or monotonicity may be used in various ways to develop non-commutative criteria for parallelism. None are perfect formal proxies for the concept.

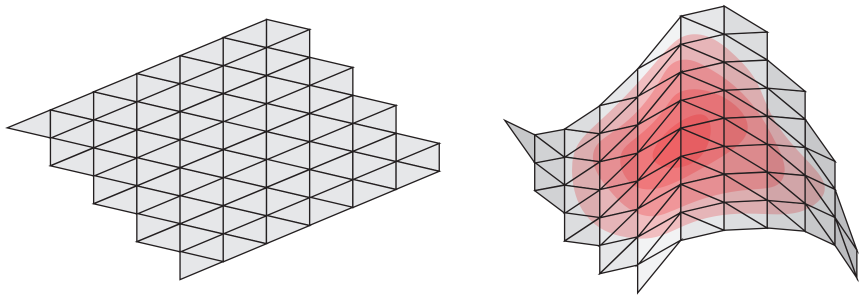


Figure 1.1: Topology vs. Metric: metric data gives this triangle mesh shape, and allows for storing other quantities like temperature. Topology tells us what is connected to what.

is the choice of a data-model and complimentary *access* language[11]. Rather than simply try to parallelize a generic random-access view of data, we need to abstract the application-relevant data-structures[12].

1.4 Data in Simulations

The data modeled by physical simulations tends to be geometric, or geometrically structured. One representative example is fluid velocity fields, and all other kinds of fields used in varieties of continuum-mechanics. These fields must be *discretized* by a choice of basis functions (tied to the discrete elements) covering the space, shape, or region of interest. Another instructive example is discrete, mechanical objects (such as boxes or particles) and tends to arise when modeling robots, rigid bodies, and other kinds of physical systems with a finite number of degrees of freedom (and therefore not needing further discretization).

In either event, we can use a fundamental distinction from Geometry: *metric* vs. *topological* data. Consider a tetrahedral finite-element mesh of an engine cavity, in which we want to model fluid flow. On the one hand, we need to keep track of

[11]A subtler point is that the "query" language actually defines the data model, because it is the interface to it. This is similar to the way that an object-class' method interface, not its representation defines what an object is.

[12]This position was taken by Keshav Pingali's group in their work on Galois[PNK+11]

temperatures, velocities, pressures, &c. at every vertex/node of the mesh—as well as the positions of each vertex. This is what we call the metric data. On the other hand, we need to keep track of the mesh's internal connectivity: which vertices each tetrahedron spans. That is what we call the topological data. In type-theoretic (programming) terms, we can loosely associate topological data with *references* or *pointers*, and metric data with quantity types like counts and floating-point values.

Importantly, topology represents (combinatorially) a geometric concept of *locality*. Locality is what other DSLs have used to improve memory performance and compile to streaming models. Image DSLs define locality on the grid-topology. Rendering pipelines use primitives like triangle-strips or fragment bundles/packets to re-use and avoid redundant work.

Given this division of types of data, we can focus our attention on three different classes of computation.

1. The computation of new metric data from existing metric data found in a topologically local neighborhood. (This includes stencil computations, and most sparse-matrix-vector multiplications.)

2. The computation of new topological data from existing metric and topological data found in a topologically local neighborhood. (This is exemplified by the problem of re-meshing.)

3. The computation of new topological data from existing metric data, absent any restriction to local topologies. (This is exemplified by the problem of collision detection.)

The fourth possibility in this matrix (the computation of metric data from metric data) is either the trivial aggregation of global statistics, or the "embarassingly parallel" computation of new values from strictly-local values, meaning without reference to neighboring data (which would imply topology). That is, this fourth metric to metric case simply reduces to the other cases.

Certainly, simulations make use of many other data structures at a lower level, such as lookup-tables, dynamically resizable arrays, search trees, and the like. However, my focus on the topology vs. metric division arises from questions of geometric modeling, not from implementation details. Therefore, we ought to expect that a programming language for simulation will be at least expressive enough[13] to capture the above manipulations of its geometric data.

1.5 Applying Databases and Relational Algebra to Simulation

The core idea of this thesis is that existing abstractions from databases—such as relational algebra—can be adapted to abstract data in physical simulations. I explore this idea through three prototype compilers, addressing each of the aforementioned classes of computations from a database perspective.

With `ebb`, I show that encoding the geometric domain in a relational schema allows us to implement stencil-like computations on an extensible range of different geometries using a small, fixed data-modeling vocabulary. This includes coupling unstructured meshes, particles, and grids together in single simulations.

With `seam`, I show that remeshing computations (more generally local topological modifications) can be expressed via the concept of a *transaction*, ensuring memory-safety of complicated pointer-manipulation at this aggregated, transactional scale of operation. I further show how leveraging relational-algebraic rewrites allows for implementing localized view-maintenance. Doing this produces combinatorial reductions in how much code needs to be maintained.

With `gong`, I show how collision detection can be cast as a *spatial join* problem. Doing so allows a three-way decomposition between the specification of a join, the safe parallel implementation of its effects, and the kind of acceleration structure/traversal used to asymptotically accelerate the doubly nested loop.

Relational Algebra gives us at least three notable benefits, which stem from the

[13]Note that this is not an argument that the preceding taxonomy is universal for simulation computations in the same meaning as saying a Turing machine is universal. For the sake of specialization, domain-specific languages for parallelism ought to be sub-Turing-complete. Given as much, the question addressed here is "what range of computations ought we strive to support?"

logical/physical data distinction, and from properties of algebras. First, the set-orientation of relational algebra generates ample opportunity for data-parallelism. This allows for straightforward GPU implementations of Ebb and Gong. Second, the abstraction of keys (database-terminology)/references (programming-language terminology) away from indices/pointers allows for analyzing *where* something is happening, and hence ensuring the absence of conflicts such as data-races. Such an analysis is critical for producing high-performance lock-free code. Third, the ability to interpret code via relational algebraic operations allows for making otherwise dramatic code transformations.

I extend these benefits with a more explicit treatment of *effects*, by which I mean the actions taken on the results of a query. At the simplest level, these are reads, writes, reductions, and so on. However, as we will see with Gong, this view of effects can extend to more elaborate schemes for how to safely create and manage new sets of data in parallel, beyond the narrower per-memory-address operations.

In another departure from databases, I will also place emphasis on *locality* of computations, in the sense of stencil computations. By restricting queries (which will mean the same as looping) in this way, I give simulation-programmers a predictable performance model for reasoning about their code execution.

1.6 Contributions

- A programming model (Ebb) for physical simulations on a range of different geometric domains, including the interactions and coupling between these domains. Notably, this includes simulations that mix structured (grids) and unstructured (graph-like) geometric domains.

- A programming model extension (Seam) for re-meshing operations and other local graph edits on the unstructured (graph-like) parts of domain models.

- A programming model extension (Gong) for writing collision detection via spatial joins of these geometric domains, that decouples the join specification from the choice of acceleration-structure/spatial-index.

- A demonstration of how to organize common geometric domains into re-usable and composable libraries, including modeling of triangle meshes, tetrahedral meshes, polyhedral meshes, regular grids, particles, rigid bodies, and half-edge meshes. Furthermore, demonstration of how the set of domains modeled may be extended.

- A common means of ensuring data-race freedom and memory safety, based on a relational analysis of program *effects*.

- A system of *views* and *incremental view maintenance* compatible with the re-meshing/local-graph-edit extensions to the programming model. I show how incremental view maintenance can be implemented as a source-to-source code transformation based on a relational semantics for effects.

- Three prototype compiler implementations: an efficient implementation of the basic programming model on CPUs and GPUs; an efficient implementation of the re-meshing extension on CPUs; and an efficient implementation of the collision detection extension on CPUs and GPUs. (All CPU implementations are serial)

- An evaluation of these implementations in comparison to a number of pre-existing example programs at various degrees of optimization. This evaluation measures comparisons of execution time, memory usage, and lines of code. Execution times were within $1.1\times$ to $1.25\times$ slower than the most highly tuned comparisons; and had up to $8-9\times$ speedups in most other cases.) No significant increase in lines of code are observed, with $2-3\times$ reductions in many cases and $10\times$ reductions for Seam where features were explicitly included in order to obviate simulation-programmer obligations.

Chapter 2

Example Simulations & Programs

This chapter introduces the rest of thesis from the point of view of a simulation-programmer using the proposed languages. Later chapters will assume the viewpoint of a compiler-programmer or language designer, as appropriate.

2.0.1 Lua Primer

All of these examples are presented as Lua-embedded domain-specific languages[1]. However, a very limited knowledge of Lua is necessary to understand the source-code listings.

Lua is a dynamic, prototype-based object-oriented language in the tradition of Smalltalk, most similar to the popular languages Python and Javascript, or the academic language Self. (Locally scoped) variable declaration is accomplished with the `local` keyword. Empty objects (called 'tables'[2]) are written `{}`. Objects may be indexed using arbitrary values as keys, e.g. `x[1]`, `x['foo']`, or `x[y]`. The syntax `x.foo` can be read as sugar for the lookup `x['foo']`. A non-empty table can be defined

```
-- this is a comment
local x = { a = 42, foo = 'bar' }
```

[1]In the interest of aesthetic consistency, syntax has been trivially modified from the original 3 language prototypes throughout this dissertation.

[2]To prevent confusion with my later, database inspired use of the term *table*, I will continue to refer to Lua tables as objects in this primer.

```lua
-- this print prints out 42
print(x.a)
```

which is equivalent to the more explicit, primitive program

```lua
local x  = {}
x['a']   = 42
x['foo'] = 'bar'
print(x['a'])
```

Lastly, a table whose keys are a range of numbers $1, 2, \ldots, n$ is called a list. A non-empty list may be defined by listing the values without any keys

```lua
local x = { 'one', 'red', 32 }
-- prints 'red' because Lua uses 1-based indexing
print(x[2])
```

which is equivalent to

```lua
local x  = {}
x[1]     = 'one'
x[2]     = 'red'
x[3]     = 32
print(x[2])
```

The aesthetic parsimony of Lua is built around these objects. In the following examples, I make use of two additional idioms. First, these objects may be used in place of module/class namespaces. Second, the parser allows for an object literal to be passed as the sole argument in a function call, absent the usual parentheses. This allows for the function calls to appear to have "named arguments" via a simple bit of syntax sugar. For example,

```lua
local MyModule = {}

function MyModule.foo( nm_args )
  if nm_args.op == '+' then
    return nm_args.lhs + nm_args.rhs
  elseif nm_args.op == '*' then
    return nm_args.lhs * nm_args.rhs
  else
    error('unrecognized op')
  end
end

local x = MyModule.foo { op ='+', lhs=19, rhs=23 }
-- prints 42
print(x)
```

and the final call here can be seen as simply sugar for

```
local tmp_arg = {}
tmp_arg.op     = '+'
tmp_arg.lhs    = 19
tmp_arg.rhs    = 23
local x = MyModule.foo(tmp_arg)
print(x)
```

2.1 Ebb Examples (Local, Metric Computations)

2.1.1 Spring-Mass Systems

Ebb allows a programmer to write local *stencil-like* computations on a wide range of geometric domains, including structured (grids) and unstructured (meshes, particles) domains, independently and coupled together. For instance, we may import (`require`) a module (which we will examine shortly) defining tetrahedral meshes, and get a mesh instance from it.

```
local Tetmesh = require 'domains.tetmesh'
local mesh    = Tetmesh.instance()
```

Then, we may extend the tet-mesh data-model for the purposes of our specific simulation. The `ebb` keyword signals that we are switching from Lua into the embedded DSL; `schema` signals that we are in a data-definition block and `SpringMass` names this data-definition.

```
local ebb schema SpringMass
  const   K  : float   = 1.0
  const   dt : float   = 0.0001
  global  E  : float   = 0.0
```

```
    include mesh

    field   mesh.edges.rest_len   : float
    field   mesh.vertices.mass    : float
    field   mesh.vertices.x       : vec3f
    field   mesh.vertices.v       : vec3f
    field   mesh.vertices.force   : vec3f
  end
```

In the above listing, we start by defining 2 `const`ants and a `global` (differing by whether their values are mutable). GPU shader programmers may recognize these globals as `uniforms`. We then `include` the mesh (a previous data definition from the module).

We extend the basic domain (`mesh`) that we included with `field`s: `rest_len`, `mass`, `x`, `v`, and `force`. These fields are defined on relational `table`s, which are explicit, finite sets of objects: `mesh.edges`, `mesh.vertices`. Each field consists of a value for each element of the relation, as in the last line, which says "There is a vector of 3 floats for each vertex, called 'force'."

Spreadsheet Metaphor. This concept—defining data in terms of relational tables—is taken from Databases. We can think of each `table` as a spreadsheet with columns and rows. Each row represents an element of the set in question (`mesh.edges`, `mesh.vertices`, &c.). Each column represents a `field` of a given type.

Given a definition of the way data is organized, (the schema) we can define `ebb` functions that belong to this schema. For instance, we can define a function to compute the resting edge-length of each edge from the position of vertices (defined by the Tet-mesh library)

```
  ebb function SpringMass.initLen( e : mesh.edges )
    var diff     = e.head.pos - e.tail.pos
    e.rest_len  = magnitude(diff)
  end
```

As before, `ebb` signals a transition from Lua into the embedded language. We define the function with a name belonging to the `SpringMass` schema's namespace, making all of the defined data names (`mesh`, `dt`, &c.) available. The function is defined as centered on an edge of the mesh. Here `e.head` and `e.tail` are vertices on either end of the edge, and `pos` is the position of a vertex as defined in the original tet-mesh file. Note that unlike Lua, we use `var` in the embedded language for variable declaration/definition.

Functions are launched as *kernels* over the sets of the data model. For instance, we compute the resting lengths of edges with the invocation

```
mesh.edges.initLen()
```

which really means "run the function `initLen` for every edge in the mesh". In this way, Ebb simulation-programmers expose data-parallelism to the compiler-programmer.

For the sake of making this a simple example, we'll use a simple explicit, forward-Euler integrator. Let x^t be a vector of all the vertex positions at time-step t, v^t their velocities, and a^t their accelerations. Let x_i^t denote the i^{th} vertex, and M^{-1} the inverse mass matrix defined by $M_{ii}^{-1} = \frac{1}{m_i}$ (the reciprocal mass of vertex i). And, let Δt represent the length of a timestep. Then, the equations of integration are

$$\begin{aligned} a^{t-1} &= M^{-1} F^t(x^{t-1}, v^{t-1}) \\ v^t &= v^{t-1} + \Delta t \cdot a^{t-1} \\ x^t &= x^{t-1} + \Delta t \cdot v^{t-1} \end{aligned}$$

Letting K be the spring constant, and r_{ij} the resting edge length of the edge between vertices i and j, the simple frictionless spring force model is

$$F_i(x, v) = \sum_{j \text{ where } (i,j) \text{ is an edge}} K \cdot (r_{ij} - ||x_j - x_i||) \frac{x_j - x_i}{||x_j - x_i||}$$

For a first attempt to write this in Ebb, lets try to write a single function that will compute the time-step for a vertex.

```
-- this function will not compile; see below
ebb function SpringMass.take_step( v : mesh.vertices )
```

```
    v.force     = {0f,0f,0f}

    for e in v.edges do
      var d_x   = e.head.x - v.x
      var m_x   = magnitude(d_x)
      var c     = K * (e.rest_len - m_x) / m_x
      force    += c * d_x
    end

    var acc     = force / v.mass
    v.x        += dt*v.v
    v.v        += dt*acc
  end
```

In this function, notice the `for` loop that draw an edge `e` from the edges connected to a vertex `v.edges`. For the time being, the meaning is clear. When we examine the definition of the Tetrahedral Mesh module, we will see how this kind of abstraction may be set up.

However more importantly, Ebb will *NOT* compile the function `take_step`. Instead, it will complain via an *effect-checker* (similar to type-checking). It will give an error like

```
  READ of 'mesh.vertices.x'
      var d_x   = e.head.x - v.x
                          ^
  conflicts with REDUCE of 'mesh.vertices.x'
    v.x        += dt*v.v
               ^
```

In other words, this function contains a *data race*. Recall that we will invoke `take_step` via a data-parallel loop over the `mesh.vertices`. When we do this, the order in which the vertices are processed is not defined. So, consider two connected vertices v_1 and v_2. If we run `take_step` on v_1 first, then we will update v_1`.x` so that when we run on v_2, `d_x` is computed using the updated, rather than old position of v_1. If we instead run `take_step` on v_2 first, the situation is reversed. The compiler has detected this *inconsistency* as an error. When compiled to various parallel execution models, stranger executions may result, depending on the particular ways in which fragments of computations get interleaved and merged.

The most important feature of Ebb and all of the languages presented in this dissertation is the ability to detect and prohibit these kinds of errors at compile time *independent of the hardware target*. As a result, even if a simulation-programmer develops, debugs, and verifies their code running serially, on the single core of a personal computer, the program is still guaranteed to be safely portable to run on a GPU (as I will demonstrate in this dissertation), to run on a super-computer (left undemonstrated), or on other sorts of parallel hardware that can exploit the data-parallel loops.

How can we write the integration code safely then? We must break it into two functions, storing an intermediate of the computation in a field.

```
ebb function SpringMass.computeInternalForces( v : mesh.vertices )
  for e in v.edges do
    var d_x   = e.head.x - v.x
    var m_x   = magnitude(d_x)
    var c     = K * (e.rest_len - m_x) / m_x
    v.force  += c * d_x
  end
end

ebb kernel SpringMass.applyForces( v : mesh.vertices )
  var acc     = v.force / v.mass
  v.x        += dt*v.v
  v.qd       += dt*acc
  v.force = {0,0,0}
end
```

This `applyForces` function still seems to contain read-write and read-reduce conflicts. So why is it ok? Simply, the different accesses are all performed to a single vertex `v`, and no neighbors. We can view this as granting each iteration of the parallel-for loop an exclusive lock to its vertex's data.

One unfortunate property of explicit forward-Euler integration is its tendency to accrue error as steadily increasing kinetic energy above the analytic solution. For instance, in our frictionless spring model, we would expect perfect conservation of kinetic energy (ignoring gravity). In order to track the empirical divergence, we can compute the kinetic energy

```
ebb function SpringMass.measureKineticEnergy( v : mesh.vertices )
  E += 0.5 * v.mass * dot(v.v, v.v)
end
```

This function reduces a per-vertex quantity into a global variable `E`. Like with non-exclusive field accesses, effect-checking ensures that a global cannot be read and reduced simultaneously, but preserves the possibility to perform this reduction in parallel.

Ebb also enables *performance portability* by abstracting over substantially different implementations of the above reduction. On (at least older generations of) GPUs, the preceding *global* reduction is most efficiently performed using a reduction tree, requiring multiple GPU kernel launches. On networked machines, special MPI reduction primitives (and special in-switch network hardware) can be used to accelerate this kind of reduction.

Finally, to run this simulation, we must initialize the data and perform the basic integration steps in a loop

```
-- Load data and constant fields
SpringMass.mesh.Load('dragon.off')
SpringMass.mesh.vertices.mass.Load('dragon_mass.data')
SpringMass.mesh.vertices.x.Load(SpringMass.mesh.vertices.pos)
SpringMass.mesh.vertices.v.Load('dragon_initvel.data')
SpringMass.mesh.vertices.force.Load({0,0,0})

-- initialize derived data
SpringMass.mesh.edges.initLen()

for i=0,10000 do
  SpringMass.mesh.vertices.computeInternalForces()
  SpringMass.mesh.vertices.applyForces()

  if i % 1000 == 999 then
    SpringMass.E.set( 0 )
    SpringMass.mesh.vertices.measureKineticEnergy()
    print('energy: ', SpringMass.E.get())
  end
end
```

In Ebb, we call the immediately preceding code the *sequential* program, which orchestrates (via Lua or some other host language) the execution of the data-parallel computations defined in Ebb proper, which defines the *parallel* part of the program. In this way, Ebb programs resemble a bulk-synchronous parallel program from the simulation-programmer's point of view.

2.1.2 Embedding Strategies / The Host Program

Across the three prototypes of this thesis, I have used a few different language embedding strategies. We just saw an example of embedding the syntax of Ebb into Lua[3] and *also* sequencing/orchestrating the sub-computations of our program from Lua. An important alternative that I used in my other prototypes ends a Lua/DSL file with a call to compile the `schema` and `functions` as a `C`-library. If we did this instead, the sequential program would be written in `C` or `C++`. It would begin by initializing a `SpringMass` object/store to hold all the data.

This *library generation* strategy looks something like the following. At the end of the Lua/Ebb file, we call a DSL-supplied Lua function to compile the library. (recall that the curly braces here suffice to pass the table of named arguments into the function call)

```
CompileLibrary {
  schema        = SpringMass,
  functions     = { SpringMass.initLen,
                    SpringMass.computeInternalForces,
                    SpringMass.applyForces,
                    SpringMass.measureKineticEnergy },
  c_obj_file    = 'SpringMass.o',
  c_header_file = 'SpringMass.h',
}
```

[3]this was accomplished using the Terra DSL facilities to hijack the Lua lexer-stream at every occurrence of the keyword `ebb`.

This call generates the object and header files to be used by the C build process.

Then, the host program can be written in C something like the following

```
#include "SpringMass.h"
...

int main() {
  SpringMass sm = Create_SpringMass();

  mesh_LoadFile(sm, 'dragon.off');
  mesh_vertices_mass_LoadFile(sm, 'dragon_mass.data');
  mesh_vertices_x_LoadCopy(sm, 'mesh.vertices.pos');
  mesh_vertices_v_LoadFile(sm, 'dragon_initvel.data');
  mesh_vertices_force_LoadConst(sm, vec3f {0f,0f,0f} );

  mesh_edges_initLen(sm);

  for (int i=0; i<10000; i++) {
    mesh_vertices_computeInternalForces(sm);
    mesh_vertices_applyForces(sm);

    if (i % 1000 == 999) {
      mesh_E_set( sm, 0 );
      mesh_vertices_measureKineticEnergy(sm);
      printf("energy: %f\n", mesh_E_get(sm));
    }
  }

  Destroy_SpringMass(sm);
  return 0;
}
```

The Ebb prototype used Lua as a host-language, more aggressively than just presented. Rather than compile the entire schema and functions, it deferred their compilation until necessary, using a JIT compilation strategy. As such, data could be loaded before defining functions on it. Results of computations on that data can then be used to modulate constants and otherwise alter the later-defined functions. Despite this, the entire Lua/DSL environment could be compiled into a C or C++

program in the same way that Lua and Lua/Terra can.

While the tradeoffs between these different hosting and embedding strategies for a DSL are interesting (and important in practice) this dissertation did not set out to learn anything novel about embedding. The reasons for adopting one or the other strategy were incidental.

2.1.3 Coupling Domains — Tracer Particles

Lets look at how to augment an existing geometric domain with additional domains and computations. To do a fluid simulation on a grid, we'll load a grid module instead of the tetrahedral mesh.

```
local GridLib   = require 'domains.grid'
local grid2d    = GridLib.Grid2d.instance()
```

As before we include the schema from the module into our simulation schema. However, this time we add new tables as well as new fields to the schema. In particular, we create a new table to represent a set of tracer particles.

```
local ebb schema StableFluids
  import  grid2d

  table   particles
  field   particles.pos          : vec3f
  field   particles.dual_cell    : grid2d.dual_cells
end
```

As we did for vertices in the spring-mass system, we define a position field for our particles. Yet, when we get to the next line we see something strange. The "type" of `dual_cell` is a table from the `grid2d` schema, not a type as we might normally think. This is really shorthand for the type `row(grid2d.dual_cells)`, which is the type of a reference[4] to some row/element of the table `dual_cells`. We use this field to track which (dual) cell of the grid each particle is currently located in.

[4]In `C/C++` approximately a pointer, absent pointer arithmetic.

Metric vs. Topological Data. Here we see the basic distinction between topological and metric data that runs through most of this dissertation. *Topological* fields (i.e. data) are those fields with `row(...)` type. *Metric* fields are those fields with simple value types (`float`, `int`, `double`, &c.) or structured value types (`vec3f`, `mat4d`, &c.). Topological fields tell us how the different `table`s of the schema are connected together, while metric fields keep track of the *quantities* that represent the state of the simulation.

Suppose we have defined a complete fluid simulator on the grid without referencing the particles. (We will look at how to do this in more detail in the next example.) How do we move the tracer particles in the flow, in order to visualize it?

Well, first suppose that the `dual_cell` data is accurate. Then, we can interpolate the velocities from the four nearby grid `cells`, and advect the particle appropriately.

```
ebb function SpringMass.advect_partcles( p : particles )
  var x1  = fmod(p.pos[0] - 0.5f)
  var y1  = fmod(p.pos[1] - 0.5f)
  var x0  = 1.0f - x1
  var y0  = 1.0f - y1

  var c   = p.dual_cell.cell
  var vel = x0 * y0 * c(0,0).vel
          + x1 * y0 * c(1,0).vel
          + x0 * y1 * c(0,1).vel
          + x1 * y1 * c(1,1).vel
  p.pos  += dt * vel
end
```

Observe that we specify *offsets* from the cell `c` using the syntax `c(xoff,yoff)` to access the 4 cells at the corners of the particle's dual-cell. The interpolation is a simple bi-linear interpolation.

Generally, Ebb does not allow the user to safely change topological data during the simulation (i.e. after initialization). However, Ebb allows us to make one important exception for particle-like data. In order to use this feature, the following things must be true. (Note that `dual_cells` is a `grid`-structured table.)

- The `schema` contains a `table` `P` and a `grid` `G` (a special kind of table, defined soon)

- There is a field `P.g : G`

- There is a way to compute the "position" of each row of `P` in the grid `G`

Given these conditions, Ebb allows us to use a `PointLocate` feature to recompute the topological link.

```
-- add the following field to the schema
local ebb schema StableFluids
  ...
  field   particles.dc_index    : vec2i
end

-- add the following to the particle update
ebb function SpringMass.advect_partcles( p : particles )
  ...
  p.dc_index = vec2i(p.pos + {0.5f,0.5f}) -- offset for dual-cell
end

-- This call will update the dual_cell topological field
StableFluids.grid2d.dual_cells.PointLocate(
  StableFluids.particles.dual_cell,
  StableFluids.particles.dc_index
)
```

Lookups that go out of range of the grid are clamped back into range. (e.g. the lookup to global index $(-1, 1)$ would resolve to looking up cell $(0, 1)$.)

When we get to the gong DSL, this special-case feature will be subsumed in a more general mechanism for updating topology from geometric intersections. However, it's worth noting that special cases can be added without the general feature, and that the special cases can be implemented very efficiently.

2.1.4 Subsets and Boundaries — Stable Fluids

In the previous example we made reference to an undescribed fluid simulator. This simulator implemented Stable Fluids[Sta99], a semi-Lagrangian implicit fluid simulation strategy on grids. In order to accomplish this, we need some way to solve linear systems, to enforce boundary conditions, and to advect the velocity field (the semi-Lagrangian step). We will now investigate these issues.

The state of our fluid simulation requires a per-cell velocity field, and then a few copies of this field in order to orchestrate the computation.

```
local ebb schema StableFluids
  ...

  field   grid2d.cells.vel      : vec3f
  field   grid2d.cells.vel_prev : vec3f
  field   grid2d.cells.vel_temp : vec3f
end
```

Velocity diffusion accounts for viscosity phenomena—the friction of the fluid with itself. Let v denote the velocity field (v_{ij} the velocity at cell (i, j)), $\nabla^2 v$ the spatial Laplacian of the velocity field, and η the viscosity of the fluid. Then the diffusion equation may be written

$$\frac{dv}{dt} = -\eta \nabla^2 v$$

The spatial discretization of ∇^2 on our grid produces the equation

$$(\nabla^2 v)_{i,j} = v_{i,j} - \frac{1}{4}(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1})$$

(assuming a cell width of length 1)

Then, using an implicit backwards-Euler discretization of the differential equation, we get the integration equations

$$\begin{aligned}
v^t_{i,j} + \Delta t\ \eta \left(v^t_{i,j} - \frac{1}{4}(v^t_{i+1,j} + v^t_{i-1,j} + v^t_{i,j+1} + v^t_{i,j-1}) \right) &= v^{t-1}_{i,j} \\
(1 + \Delta t\ \eta)v^t_{i,j} - \frac{\Delta t\ \eta}{4}(v^t_{i+1,j} + v^t_{i-1,j} + v^t_{i,j+1} + v^t_{i,j-1}) &= v^{t-1}_{i,j} \\
Av^t &= v^{t-1}
\end{aligned}$$

where in the last formulation, we have bundled all of the equations into a linear system, to which we will apply an iterative linear solver.

However, first notice that this problem is ill-posed. $v_{i-1,j}$ is only meaningful if i is greater than 0 (or whatever the bottom of the grid range is). In physical terms, we need to decide what happens at the *boundary* of the grid by imposing *boundary conditions*. One simple solution, useful for certain analytic problems is to assume the grid simulation is infinitely repeating and symmetric by allowing the grid to "wrap around". This introduces various periodicities into the simulation results, and so is called *periodic* boundary conditions. We could impose these by providing a directive to ebb telling it to wrap around every out-of-range field access to the grid:

```
local ebb schema StableFluids
  ...
  set periodic[0] grid2d.cells  -- set x-wrapping
  set periodic[1] grid2d.cells  -- set y-wrapping
end
```

However, more often we want to model the boundaries as walls. *Von-Neumann* boundary conditions allow us to accomplish this by specifying the velocity with respect to the walls. We will specify that the velocity field should always be parallel to the walls—that there should be zero velocity in the normal direction to the wall. Skipping the derivation, the integration equation arising from this situation along the negative x direction wall is

$$(1 + \Delta t\ \eta)v^t_{(i,j)} - \frac{\Delta t\ \eta}{4}(2v^t_{i+1,j} + v^t_{i,j+1} + v^t_{i,j-1}) = v^{t-1}_{i,j}$$

This suggests that we will have to perform some special kind of computation on the boundary of the grid, different from the computation on the interior. Ebb provides a *subset* mechanism for this purpose.

```
local ebb schema StableFluids
  ...
  subset  grid2d.cells.interior
  subset  grid2d.cells.boundary
end

...

-- load data as if the subset was a Boolean field
StableFluids.grid2d.cells.interior.Load('interior_flags.data')
StableFluids.grid2d.cells.boundary.Load('boundary_flags.data')
```

Note that rather than provide a sophisticated set of language features for describing subsets, we simply let/require the user load arbitrary Boolean data. (Subset management and computation was not a major focus of this dissertation research)

To solve the diffusion system, we will rely on a Jacobi solver. Let $p = v^{t-1}$ be the velocity at the previous timestep. Since the solver itself is iterative, we will need to name the velocity at the current and next iteration. The current iteration velocity will be denoted v and the next iteration velocity v'. Then given a decomposition of the system matrix $A = D + R$ into its diagonal D and off-diagonal R entries, the iteration equations for the Jacobi solver are

$$\begin{aligned} v' &= D^{-1}(p - Rv) \\ v'_{i,j} &= \frac{1}{1 + \Delta t\ \eta}\left(p_{i,j} - (Rv)_{i,j}\right) \\ &= \frac{1}{1 + \Delta t\ \eta}\left(p_{i,j} + \frac{\Delta t\ \eta}{4}(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1})\right) \end{aligned}$$

In the negative x boundary case, this is modified to

$$v'_{i,j} = \frac{1}{1 + \Delta t\ \eta}\left(p_{i,j} + \frac{\Delta t\ \eta}{4}(2v_{i+1,j} + v_{i,j+1} + v_{i,j-1})\right)$$

Our diffusion solver will simply iterate this stencil computation multiple times for each timestep that we advance the simulation.

```
ebb function StableFluids.diffuse_step( c : grid2d.cells )
  var diag_coeff  = 1 / (1 + dt * viscosity)
  var edge_coeff  = (dt * viscosity) / 4

  var esum    = c(-1, 0).vel + c( 1, 0).vel +
                c( 0,-1).vel + c( 0, 1).vel
  c.vel_temp  = diag_coeff * (c.vel_prev - edge_coeff * esum)
end
```

We will need an additional modified version of the function for the boundary case.

```
ebb function StableFluids.diffuse_boundary_step( c : grid2d.cells )
  var N_x         = grid_size(grid2d.cells, 0)
  var N_y         = grid_size(grid2d.cells, 1)
  var diag_coeff  = 1 / (1 + dt * viscosity)
  var edge_coeff  = (dt * viscosity) / 4

  var esum : float =
      ( (xid(c) == 0  )? c( 1, 0).vel else c(-1, 0).vel )
    + ( (xid(c) == N_x)? c(-1, 0).vel else c( 1, 0).vel )
    + ( (yid(c) == 0  )? c( 0, 1).vel else c( 0,-1).vel )
    + ( (yid(c) == N_y)? c( 0,-1).vel else c( 0, 1).vel )
  c.vel_temp  = diag_coeff * (c.vel_prev - edge_coeff * esum)
end
```

Then in order to solve for the velocity, we sequence these calls as

```
local function diffusion_solve()
  StableFluids.grid2d.cells.Copy { from='vel', to='vel_prev' }

  for i=1,20 do
    StableFluids.grid2d.cells.interior.diffuse_step()
    StableFluids.grid2d.cells.boundary.diffuse_boundary_step()
    StableFluids.grid2d.cells.Copy { from='vel_temp', to='vel' }
  end
end
```

The projection step of Stable Fluids, being a Poisson equation resolves into a similar

Jacobi solver. The advection step however, uses a (possibly non-local, non-neighbor) lookup into the grid in order to move the velocity field along itself. (This is the Lagrangian part of "semi-Lagrangian")

Using (or perhaps abusing) the previously described `PointLocate` feature, we can accomplish advection. We extend the grid cells themselves, rather than the particles with a topological field.

```
local ebb schema StableFluids
  ...
  field   grid2d.cells.adv_index  : vec2i
  field   grid2d.cells.adv_cell   : grid2d.dual_cells
end

ebb function StableFluids.advection_point( c : grid2d.cells )
  -- we lookup backwards because we are using
  -- backwards-Euler integration
  var offset  = -c.vel_prev
  -- center is pre-defined by the Grid library
  return c.center + dt * offset
end

ebb function StableFluids.set_advect_target( c : grid2d.cells )
  -- 0.5f offsets are for dual-cell grid
  c.adv_index = vec2i( advection_point(c) + {0.5f,0.5f} )
end

ebb function StableFluids.advect_velocities( c : grid2d.cells )
  var lc  = c.adv_cell.cell
  var pt  = advection_point(c)

  var x1  = fmod(pt)
  var y1  = fmod(pt)
  var x0  = 1.0f - x1
  var y0  = 1.0f - y1

  c.vel   = x0 * y0 * lc(0,0).vel_prev
          + x1 * y0 * lc(1,0).vel_prev
          + x0 * y1 * lc(0,1).vel_prev
          + x1 * y1 * lc(1,1).vel_prev
end

local function do_advection()
  StableFluids.grid2d.cells.Copy { from='vel', to='vel_prev' }
  StableFluids.grid2d.cells.set_advect_target()
  StableFluids.grid2d.dual_cells.PointLocate(
    StableFluids.grid2d.cells.adv_cell,
    StableFluids.grid2d.cells.adv_index
  )
  StableFluids.grid2d.cells.advect_velocities()
end
```

This example helps demonstrate the versitility of Ebb's primitives. As we might expect from any reasonable language design, the features can be repurposed to new ends beyond their original intention.

2.1.5 Language Interoperability — Custom Solvers

Similar to other graphics languages, Ebb was developed to be used as only one part of a larger application. To this end, we saw different strategies in §2.1.2 for embedding the language. However, another important aspect is to interoperate with existing code.

For instance, highly optimized packages for linear system solving exist on many machines. While calling Ebb code from the host language is straightforward enough, it's less clear how to interact closely with the data model.

In Ebb I exposed data-layout-description meta-data objects to simulation-programmers. While Ebb retains the freedom to layout data as it sees fit internally, these descriptors allow simulation code to query Ebb about the layout it has chosen. When invoked for a particular field these descriptors tell the simulation-programmer whether data is located in the GPU or CPU memory, which order the dimensions of a grid are stored in, as well as the address and strides the simulation-programmer should use to index the field looking for a particular row's datum.

For instance, here is a function designed to compute the diffusion step in the frequency domain, by making use of CUFFT[5], an optimized CUDA FFT library. We could have likewise used FFTW[6] if the data were located on the CPU

```
local gridFFT   = GridLib.Grid2d.instance()

local ebb schema
  ...
  field   grid2d.cells.vel_x    : float
  field   grid2d.cells.vel_y    : float

  include gridFFT
  field   gridFFT.cells.vel_x   : vec2f -- complex number
  field   gridFFT.cells.vel_y   : vec2f -- complex number
end

-- here we use terra as a C-like language to import C code
local C = terralib.includecstring "#include <cufft.h>"

-- definitions of extra ebb functions omitted for brevity
...

local function diffuseProjectGPU()
  -- break velocity field into x and y components
  StableFluids.grid2d.cells.separate_vel()
  -- verify location
  assert( grid2d.cells.vel_x.getDLD().proc == 'GPU',
          "expected cells.vel_x data to be on the GPU" )
  -- get pointer addresses
  local cells       = StableFluids.grid2d.cells
  local FFTcells    = StableFluids.gridFFT.cells
  local x_GPU       = cells.vel_x.getDLD().address
  local y_GPU       = cells.vel_y.getDLD().address
  local x_FFT_GPU   = FFTcells.vel_x.getDLD().address
  local y_FFT_GPU   = FFTcells.vel_y.getDLD().address

  -- convert to freq. domain
```

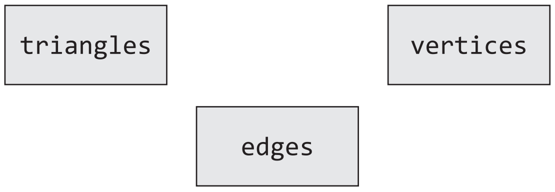
[5]developer.nvidia.com/cuFFT
[6]fftw.org/

```
  C.cufftExecR2C(x_GPU, x_FFT_GPU)
  C.cufftExecR2C(y_GPU, y_FFT_GPU)

  -- diffusion is separable in the frequency domain,
  -- and so becomes a trivially parallel operation
  -- operating on each frequency independently.
  StableFluids.gridFFT.cells.FFT_diffusion()

  -- convert back
  C.cufftExecC2R(x_FFT_GPU, x_GPU)
  C.cufftExecC2R(y_FFT_GPU, y_GPU)
  -- re-pack
  StableFluids.grid2d.cells.rejoin_vel()
end
```

2.2 Geometric Domain Libraries

2.2.1 The TriMesh library

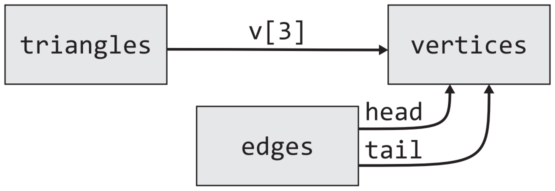
As a running example, I will explain the construction of a standard triangle mesh domain. To begin, we create three relational tables to model the triangles, (directed) edges, and vertices of the mesh. As we do this, I will use a graphical notation to visualize the `schema` we are describing. In the graphical notation, each box represents a different `table`.



```
local ebb schema TriMesh
  table   triangles
  table   edges
  table   vertices
end
```

In fact, we have already seen the declaration of tables in the special case of a table of particles.

The most basic way to connect tables together is through using *key-fields*, aka. topological fields, or `row`-type fields. These fields encode functional relationships such as specifying the `head` or `tail` of an edge. Of course each edge must have one and exactly one head (resp. tail) vertex. We set up these fields simply by declaring them, as we have already seen done with the `dual_cell` field designating the cell in which each particle is located.



```
local ebb schema TriMesh
  ...
  field   triangles.v     : vertices[3]
  field   edges.head      : vertices
  field   edges.tail      : vertices
end
```

The visual notation for a key-field is simply an arrow labeled with the field's name, going from the base table for the field to the table of its destination type. Note in particular here that we allow row-types to be grouped into vectors or matrices to simplify our naming schemes.

In order to make use of a key-field in code, we simply "follow" the field, as in the expression `e.head` or `e.tail`, which we saw in earlier uses of the TetMesh.

Note that loading in initial data for a key-field requires using integer values to address the rows of other tables. While these specific values are important for initializing the data, the compiler-programmer can swap out the representation however they like after that point.

We've already seen the previous concepts (`table`s and topological `field`s) used directly. However, we never did get an explanation for how the `for e in v.edges` construct works.

We call these *Query-loops*. Setting up a seamless query loop like `for e in v.edges` requires both the underlying modeling ideas, plus *macro* functionality to make it look pretty for the user of the geometric-domain library. The full syntax would more accurately resolve to something like

```
for e in edges where e.tail == v do ... end
```

or

```
for e in query(e, edges, e.tail == v) do ... end
```

which mean the same thing, but presents an important syntactic variation, highlighting that `e` is ranging over the results of some "query".

Field Macros. In the latter case, we can see how a macro that replaces `v.edges` with `query(e, Edges, e.tail == v)` would suffice to give us the desired syntax for a user of the geometric domain library. We can accomplish this using a *field macro* feature built into Ebb. The basic idea is taken from Terra's macro design, using a Lua function and quoted code to implement macros. In Ebb we write this...

```
TriMesh.vertices.NewFieldMacro('edges', function(v)
  local edges = TriMesh.edges
  return ebb `query(e, edges, e.tail == v)
end)
```

Now, whenever the "field" `vertices.edges` is accessed, the Ebb compiler will execute the anonymous function bound via `NewFieldMacro`, and substitute the quoted expression (`ebb `...`) in its place.

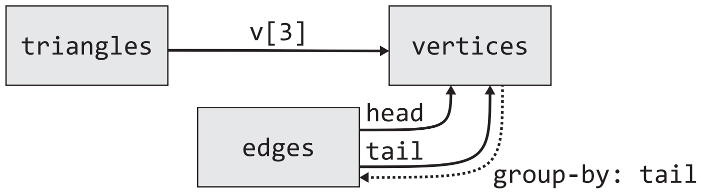
Returning to our discussion of query-loops, we can observe a general idea ("queries, in the sense of relational databases can be embedded into programs via looping constructs") and also a vastly more specific one ("topological fields may be 'inverted' in the sense that we can loop over the pre-image of some element"). One of, if not the central design decision in Ebb is to prohibit the general idea in favor of the specific one.

In Ebb, queries must have the specific form `query(x, X, x.f == y)`, where `X` is a `table`[7], `x : X` an element ranging over it, `y : Y` some specific element of `table Y`,

[7]not a `grid`-structured table either

and finally `X.f : Y` a field linking the two. The query precisely specifies the pre-image of `y` under the field `f` interpreted as a function $X \to Y$.

Ebb further imposes responsibility on the simulation-programmer by requiring them to anticipate and pre-declare the way in which they want to query the data. In order to invert a field `X.f : Y` with a query, the simulation-programmer must have issued a data-modeling directive, telling Ebb to prioritize access to `X` via its `f`-values. (We extend our visual notation with a dotted-arrow, labeled with `group-by : tail`. This symbol is used to indicate where the schema has been prepared for fast querying.)

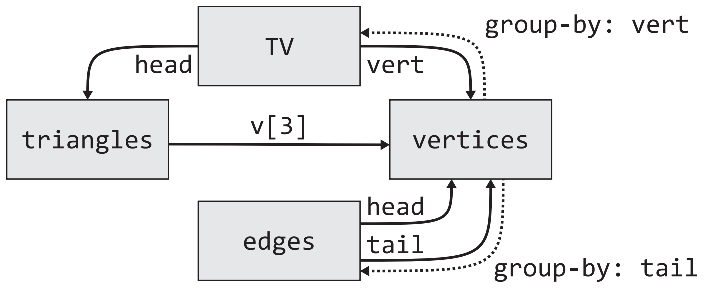


```
local ebb schema TriMesh
  ...
  set   group_by  edges.tail
end
```

Our restrictions and this directive together guarantee that queries can be answered in constant time, using a single memory lookup.

In order to support a more complex query, like "all of the triangles around a vertex" we need to do more extensive work setting up our domain-library. The `triangles.v` field is really three fields, so we can't just take the pre-image of a vertex in `v[0]` or `v[1]` or any other field we come up with, since the relationship between triangles and vertices isn't functional.

Instead we have to represent this relationship as a table itself (the main idea of relational data modeling). In Ebb, I informally refer to these metric-data-free tables (that exist solely for the purposes of navigation) as *auxiliary tables*.



```
local ebb schema TriMesh
  ...
  table   TV  -- triangle-vertex pairs
  field   TV.tri     : triangles
  field   TV.vert    : vertices
  set     group_by  TV.vert
end

TriMesh.vertices.NewFieldMacro('triangles', function(v)
  return ebb `query(tv, TriMesh.TV, tv.vert == v).tri
end)
```

Note the additional access (`query(...).tri`) post-fixed to the query in the macro. This syntax is shorthand to immediately get the triangle component of the query result, rather than a row of a table we're looking to hide. In effect, this desugars as

```
for t in query(tv, TriMesh.TV, tv.vert == v).tri do ... end

-- desugars to
for tv in query(tv, TriMesh.TV, tv.vert == v) do
  var t = tv.tri
  ...
end
```

2.2.2 Data Loading

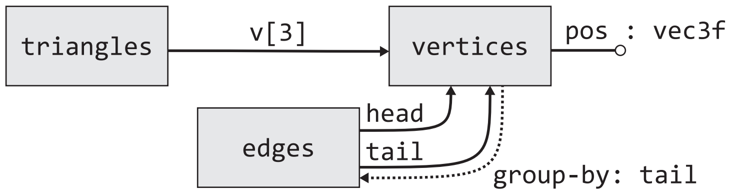
Data Loading is closely connected to questions of host-language, interoperability and DSL embedding—all are "external" to the DSL in some sense. One approach is to focus on supporting certain standard file formats. Such an approach can be good for reducing code complexity. Another approach is to provide a generic mechanism for all file formats, and leave writing the loading/saving code to users. In practice, both are required. However, the latter is in some sense more fundamental, since "standard" formats can be built on top of a generic mechanism.

For each DSL prototype I exposed some choice or combination of *by-row* loading and *by-column* loading. In the row case, a sequence of API calls are made, each of which loads an entire row of a table, specified as a tuple. More useful (and potentially more efficient) is the column case, where a load call simply passes a pointer to an array holding the entire contents of a table column. The by-column approach has the additional benefit that it allows for easier modularization of loading when split between libraries and simulations using those libraries, since simulations may extend the base data model with additional fields.

Loading metric fields is relatively straightforward, but loading topological fields presents an additional challenge. We must have a consistent way of refering to rows, i.e. encoding *keys*. When loading data, we number rows of table `X` from 0 to $n-1$, where n is the number of rows in table `X`. However, in general there is no guarantee that later API access to the data presents the rows in the same order. This presents another subtle tension around interoperability that I left mostly unexplored. Many instances of this issue can be resolved by simply loading an auxiliary `id` field, and using that data to keep track of a consistent external identity. In some cases, this may even be a 64-bit pointer to some "object" of a host program.

For the `TriMesh` library, I provide at least a standard loader from `OFF` files, which consist of a "triangle list" (consisting of triples of vertex indices) and a "vertex list" (consisting of triples of floating point values—coordinates of the points). As such, the standard `TriMesh` library comes equipped with a standard `pos : vec3f` field holding these coordinates.

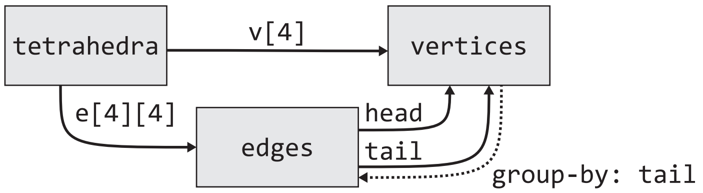
When necessary, metric fields can be annotated in our graphical notation with a labeled, dangling, circle-tipped arrow.



Note that the format of `OFF` files implies an evident but up-to-now ignored point: once the vertices and triangles of a mesh are known, the edges may be derived from that existing data. In Seam (§2.3.2) I will describe a *view* mechanism that exploits this observation directly. However in the more basic, stripped-down data model of Ebb we must handle the edge data derivation ourselves. Consequently, the loader uses DSL-external data structures to build up and de-duplicate a set of mesh edges, using a sparse-matrix/graph encoding.

2.2.3 The TetMesh library

If we simply collect all of the ideas from the triangle mesh and adapt them, we can build our desired tetrahedral mesh. I will leave out extraneous detail, but include some additional data necessary to execute the FEM example from the evaluation section (§7.1.2).

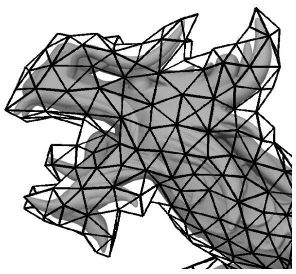


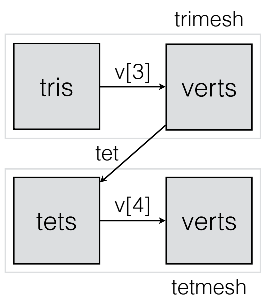
```
local ebb schema TetMesh
  table   verts
  table   edges
  table   tets

  field   tets.v      : verts[4]
  field   tets.e      : edges[4][4]
  field   edges.head  : verts
  field   edges.tail  : verts
  set     group_by    edges.tail
end
```

Similar to the triangle mesh, the tetrahedral mesh has 3 relations, `tets`, `verts`, and `edges`. Key-fields are defined for tet vertices `tets.v[4]`, and edge endpoints `tail` and `head`. We group the edges by `tail`. We make the choice to include not only directed edges, but also a self-loop edge per vertex. Rather than being principally geometric, edges of the tet mesh are used to model the support structure for sparse matrices on the domain, and these self-loops correspond to diagonal entries. Since updates to the edge operator are computed per-tetrahedron and reduced into the appropriate edges, we further augment the tet relation with a matrix-organized key-field `tets.e[4][4]` simplifying indexing code. To encode the stiffness matrix itself, we store 3×3 matrix values per-edge in a stiffness field, producing, in aggregate, a fairly sophisticated sparse block matrix encoding that can be easily addressed and updated from the tetrahedra.

2.2.4 Render-mesh Coupling

Since geometric domains in Ebb are built out of a collection of relational tables, there is no reason that simulation code can't link relations of different domains together in a given simulation. For instance, a common strategy for soft-body FEM simulations in graphics is to embed a high resolution triangle mesh of an object inside of a lower resolution tetrahedral mesh, on which the simulation actually occurs. To do this in Ebb, a programmer only needs to set up one additional field `TriMesh.vertices.tet` of `TetMesh.tets`-type. Then, vertices of the triangle mesh can update their position by interpolating the positions of the containing tet's vertices `v.tet.v[k].pos` for `k=0,1,2,3`.

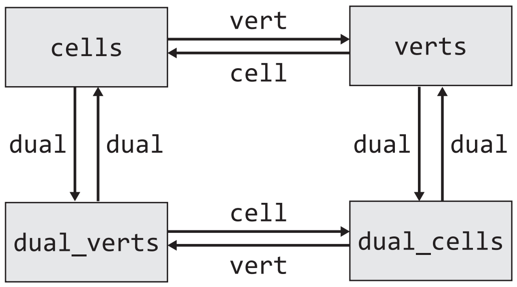




Of particular interest with this example is the ability to provide a standard binding between the triangle-mesh schema and the rendering pipeline. In this mapping `triangles.v` is simply the triangle list and `vertices.pos` the position data. Additional fields on the two tables can be bound as *triangle attributes* or *vertex attributes* respectively. For simulations entirely on the GPU, this coincidence between the standard rendering model and our relational model can allow us to directly render and simulate without any extraneous data motion.

2.2.5 The Grid Library

In the implementation of the regular grid domain, I chose to provide multiple kinds of elements (e.g. vertices, cells, dual-cells, &c.). As we will see, establishing topological connections between grids requires no data. Consequently, creating a system of multiple interrelated grids incurs no storage overhead. Besides a very small, constant amount of metadata, each individual grid only uses space proportional to its number of rows for each *field* defined on it.



```
local ebb schema Grid2d
  grid[2]   cells
  grid[2]   verts
  grid[2]   dual_cells
  grid[2]   dual_verts
end
```

Grid declarations must state the number of grid indexing dimensions. Note that in our graphical notation, we make no special annotations to distinguish grids from tables, although grids cannot have `group_by` directives issued on them as targets, and therefore edges out of grids cannot be inverted[8].

Most connections between grids (including from a grid to itself) are more succinctly and computation-efficiently described by arithmetic. For instance, "one grid

[8]This is because grids must be stored in grid-order. Where absolutely necessary, fields can be inverted using the auxiliary table technique already shown.

cell away in the positive-x direction" corresponds to "increment the x-index by one" and "the minimum corner of this cell in all directions" corresponds to "re-interpret the input indices—unmodified—as indexing a different grid." We saw some of these examples of using such links in the previous Stable Fluids example (§2.1.4). For instance, given a cell `c` we expressed the x-offset as `c(1,0)`, and given a dual-cell `dc` we expressed the minimum of the four cells at its corners as `dc.cell`. As with *query-loops*, these are both syntax macros abstracting a more basic underlying feature.

In general, Ebb allows us to express arbitrary *affine-indexing* transformations. Such transformations cost some arithmetic, but zero memory accesses to perform. Recall that an affine transformation of a vector $c = [x, y]$ or $c = [x, y, z]$ is expressable as multiplication by a matrix, plus an offset vector.

$$Ac + b = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

or

$$Ac + b = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Ebb provides a special primitive built-in function `affine(G2,A,b,c)` where `c` is a row of grid `G1`, a d_1-dimensional grid, where `G2` is a d_2-dimensional grid, `A` is a $d_2 \times d_1$-dimensional matrix, and `b` a d_2-dimensional vector. Furthermore, `A` and `b` must be constant and integer-valued[9].

Affine transformations are the most general set of transformations satisfying two important properties. First, they are closed under composition, meaning that given a sequence of affine transformations, the result can always be rewritten as a single affine transformation. Second, they send linear inequalities on indices to linear inequalities on indices. As a result, the question "is this access guaranteed to be valid?" can be tractably analyzed for accesses to fields defined on grids[10]. Constant offsets alone

[9]While I never reached a conclusive decision on the question, rational-number coefficients in the transformation may also be allowed, with the specification that the transformation output is rounded strictly *down*

[10]The question reduces to satisfiability of an integer linear program

suffice for the majority of applications, reducing `A` to an identity matrix.

In order to package this affine-indexing feature, we can use the existing field-macro feature

```
Grid2d.dual_cells.NewFieldMacro('cell', function(c)
  return ebb `affine(Grid2d.cells, {{1,0},{0,1}}, {0,0}, c)
end)
```

or when we want to use the `c(1,0)` syntax, we can use another macro syntax feature.

```
Grid2d.cells.NewApplyMacro(function(c,x,y)
  return ebb `affine(Grid2d.cells, {{1,0},{0,1}}, {x,y}, c)
end)
```

As already discussed, we can declare subsets or periodicity of grids to enforce boundary conditions, as well as introspect on cell values using `x_id`, `y_id`, &c. built-in functions.

2.3 Seam Examples (Local Remeshing)

Seam allows a simulation programmer to write local topology-editing operations, such as re-meshing. While Ebb assumed (with the exception of point-location in a grid) static topology, Seam complements that model with a way to change topology in-between the execution of Ebb functions.

Beyond the scope of this thesis, Seam provides a sketch of ways to support verified synthesis of cyclic data-structure manipulation code via what I and my co-author Manolis Papadakis have called *local graph edits*. Such an edit is an operation that matches, deletes, replaces, and re-wires a local-neighborhood of some vertex in a graph, or some row in a relational data model. Local meaning that elements touched must lie within a constant number of hops away from the *center* element.

As such, I will first describe the applicaiton of Seam to a social network data model described in the style of Ebb/Seam schemas.

2.3.1 A Simple Social Network

Consider a simple social network with user accounts that follow one another. We can model the `schema` for such a data structure in Seam. As in Ebb, the keyword `table` indicates an unordered set of elements, and `field` defines per-element data stored in that table. For instance, every `Follow` models a directed edge between accounts, by specifying a `src` and a `dst Account`. (The basic outline ought to remind you of vertices and edges in a triangle or tetrahedral mesh)

```
local seam schema SocialNet
  table Account
  table Follow
  field Follow.src : Account
  field Follow.dst : Account
  ...
end
```

Extending the data-model beyond Ebb, Seam programmers are allowed to specify `invariants` by writing a function that `assert`s the properties defining the invariant. Invariants are thus expressed using familiar control flow constructs, rather than a separate logical specification language.

```
local seam schema SocialNet
  ...
  invariant no_follow_yourself( f : Follow )
    assert( f.src != f.dst )
  end
end
```

Importantly, Seam restricts invariants to a single argument. This constraint makes it difficult to write non-local operations that have to doubly-loop over sets, ensuring that invariants can be efficiently checked at data-loading time. The `no_follow_yourself` invariant above prohibits self-loops from the data-model.

Once a schema has been defined, a Seam programmer can write `operations` on that schema, which we distinguish sharply from Ebb `function`s. While Ebb functions are designed to be run in parallel for all rows of a table, Seam operations are expressly intended to be run on a single row to the exclusion of all other rows. For instance, the operation `SocialNet.Remove` of our example schema removes a specific user account from the network. Doing so requires not just `delete`-ing the account, but also all follow relationships with that account as a source or destination.

```
seam operation SocialNet.Remove( a : Account )
  delete a
  for f in Follow where f.src == a do
    delete f
  end
  for f in Follow where f.dst == a do
    delete f
  end
end
```

In Seam like Ebb, `table`s represent unordered sets. Consequently, our operation definitions must be insensitive to the order in which the table elements are looped over. Furthermore, to keep operations local, we only allow looping over elements connected to a variable already in scope. These two decisions reproduce the constrained query-loop construct from Ebb.

In order to simplify the writing of order-independent operations, we take Seam's semantics one step forward and make all effects inside of an operation transactional. That is, all the operation's effects on memory (`delete`, `new`, `update`) are deferred, and applied atomically, after the body of the operation is done executing. So, even though Seam operations are written as pseudo-imperative code, operation semantics are actually declarative. This behavior further frees the programmer from the responsibility of ensuring that effects are correctly ordered. For instance, in the preceding operation `SocialNet.Remove`, `Account a` is deleted before being used in the two loops.

Seam statically checks that all references to the deleted `Account a` have either been removed or re-assigned at compile time. If we were using a database, we could have handled this particular case by declaring an `ON DELETE CASCADE` policy [UGMW02] between `Accounts` and `Follows`. However, such policies have the drawback of defining behavior globally, rather than per-operation. We need only consider another basic operation to see this drawback in practice.

Suppose we realize that our network contains duplicate accounts; we decide to implement an operation to merge accounts. This operation deletes one of the two accounts, but rather than delete the connected `Follows`, it re-routes them to the retained account. In this case, we *update* the dangling references rather than *delete* their holders. As this example shows, different applications, and—more importantly—different operations within the same application, often require different policies to maintain data integrity.

```
-- Incorrect version, will not compile
operation SocialNet.Merge( a1 : Account, a2 : Account )
  delete a2
  for f in Follow where f.src == a2 do f.src = a1 end
  for f in Follow where f.dst == a2 do f.dst = a1 end
end
```

If we are in a rush, we might quickly code the account merge operation as shown above, without realizing that it can violate the `no_follow_yourself` invariant. Seam was designed to allow the application of formal methods via SMT solvers. As a result, the Seam compiler is able to generate precise and complete error messages[11] for operations. Here we receive

```
-- Compiler error message
Invariant 'no_follow_yourself' violated, on input:
  Accounts = { a1, a2 }
  Follows  = { f1 }
  f1.src = a1, f1.dst = a2
```

The compiler responds with a minimal network (containing two accounts) that satisfies the invariant, but will no longer do so after `SocialNet.Merge` executes.

In response we will modify the loops to check for follows between the merged accounts, and delete rather than update those follows. Now we have an operation that successfully compiles. Any schema that satisfies our invariant is guaranteed to satisfy that invariant after any combination of account deletions and mergers, ensuring closure under our defined set of operations.

```
-- Corrected version
```

[11]except in cases where the type-checker times out

```
seam operation SocialNet.Merge( a1 : Account, a2 : Account )
  delete a2
  for f in Follow where f.src == a2 do
    if f.dst == a1 then delete f else f.src = a1 end
  end
  for f in Follow where f.dst == a2 do
    if f.src == a1 then delete f else f.dst = a1 end
  end
end
```

As we add more types of elements, operations or invariants to our data structure, Seam automatically checks for problems arising from their interactions.

group-by? In Seam, I made the decision to suppress the requirement for the simulation programmer to explicity set `group-by` on tables before performing query-loops. As we can see in `SocialNet.Merge`, topology editing (to a substantially greater degree than stencil computations) frequently requires accessing a table via multiple distinct fields. Furthermore, since the goal is to support editing topology, it's no longer reasonable to keep the definition of auxiliary tables externalized. These design tensions will have crucial consequences for the underlying data model implementations (§6).

2.3.2 Triangle Mesh: Edge-Based Remeshing

Now, lets revisit our Triangle-Mesh schema from Ebb using new tools and ideas from Seam. To be explicit, we'll use individual fields for each vertex.

```
local seam schema TriMesh
  table Tri    -- triangles
  table Vert   -- vertices
  field Tri.v0 : Vert
  field Tri.v1 : Vert
  field Tri.v2 : Vert

  invariant non_degenerate( t : Tri )
    assert( t.v0 != t.v1 and t.v1 != t.v2
                         and t.v0 != t.v2 )
  end
  ...
end
```

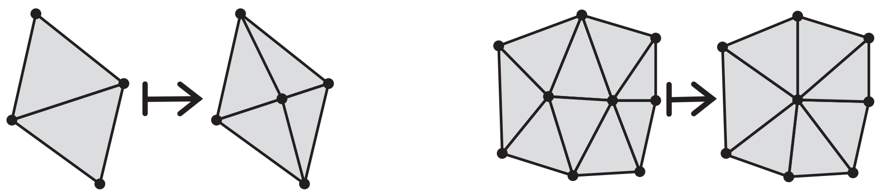
In Seam (as we just saw) we can define invariants. One of the most basic (but often overlooked) invariants of a triangle mesh is that every triangle is combinatorially `non_degenerate`, meaning that all three of each triangle's vertices are distinct.

In the Ebb `TriMesh`, the `Edge`s between vertices were defined as an additional table of explicitly managed elements. However, this was redundant; once triangles are specified, all the edges between their vertices are implicitly defined. To help ease and automate the simulation programmer's work in such situations, Seam leverages the concept of a *view* from databases. A `view` is a set of typed tuples, along with a function (`viewdef`) that computes the view's content based on the content of the basic tables. Such view definition functions take a single argument and use the same control constructs as `invariant`s and `operation`s. However, they use `emit` statements in place of `assert` or `delete`, `new` and `update` effects.

Using views, we may now define the `Edge`s, as well as two auxiliary tables tracking incident triangle-vertex pairs, as well as incident triangle-edge pairs (where each edge is reduced to a pair of vertices).

```
local seam schema TriMesh
  ...
  view Edge : { hd:Vert, tl:Vert }
  view TVV  : { t:Tri, hd:Vert, tl:Vert }
  view TV   : { t:Tri, v:Vert }
  viewdef( t : Tri )
    emit { t, t.v0 } into TV
    emit { t, t.v1 } into TV
    emit { t, t.v2 } into TV
    emit { t.v0, t.v1 } into Edge
    emit { t, t.v0, t.v1 } into TVV
    emit { t.v1, t.v2 } into Edge
    emit { t, t.v1, t.v2 } into TVV
    ...
  end
end
```

Our goal with Seam is to implement *adaptive remeshing*—the continuous adaptation of a mesh, to adjust its resolution and fidelity over the course of a larger simulation [NSO12, NPO13, PNdJO14, WTGT09, WTGT10, BB09, DBG14]. This technique is critical in the simulation of certain phenomena, like folding and fractures. One common approach to remeshing is to define two local edit operations: edge-collapse and edge-split, to reduce and increase the resolution respectively.



(a) `TriMesh.EdgeSplit` (b) `TriMesh.EdgeCollapse`

Figure 2.1: Edge-based remeshing operations for a Triangle Mesh

The edge-split operation creates a new *split vertex*, and then replaces every triangle connected to the edge with the two triangles resulting from splitting that edge. The edge-collapse operation collapses one vertex into another by deleting it (similar to account merging), then re-routing any connected triangles to the remaining vertex.

```
seam operation TriMesh.EdgeSplit( e : Edge )
  let vh = e.hd
  let vt = e.tl       -- endpoints of e, head and tail
  let sv = new Vert   -- vertex splitting the edge
  for t_e in TVV where t_e.hd == vh, t_e.tl == vt do
    let t = t_e.t
    -- the following line is a pattern-match
    let vh, vt, vopp = t.v0, t.v1, t.v2
    delete t
    new t_h : Tri { v0=vh, v1=sv, v2=vopp }
    new t_t : Tri { v0=sv, v1=vt, v2=vopp }
  end
end

seam operation TriMesh.EdgeCollapse( e : Edge )
  let vh = e.hd
  let vt = e.tl       -- edge end-points
```

```
    -- strategy: delete vt and redirect it to vh
    delete vt
    for t_v in TV where t_v.v == vt do
      let t = t_v.t
      if t.v0 == vh or t.v1 == vh or t.v2 == vh then
        delete t
      else
        if     t.v0 == vt then t.v0 = vh
        elseif t.v1 == vt then t.v1 = vh
        elseif t.v2 == vt then t.v2 = vh end
      end
    end
  end
```

This example illustrates how the view mechanism eliminates a combinatorial increase in the amount of code, as an application is extended. The views `TV` and `TVV` are each used in only one of the two operations. Still, the Seam compiler generates code to propagate updates to both views, and adds it to both operations. If we had instead represented these views as base tables, we would have had to write update code for the combination of each view and each operation, i.e. we would have had to maintain a quadraticly rather than linearly growing amount of code, as new operations and views are added.

Now suppose that after using the re-meshing in our code for a while, we discover that re-meshing produces duplicate copies of triangles, which is causing strange behavior in another algorithm. We decide that we want to disallow this behavior, but we're not quite sure why it's happening. With Seam, we have an unusual option of where to begin: add an invariant.

```
local seam schema TriMesh
  ...
  invariant unique_tri_key( t : Tri )
    for t2 : Tri where t2.v0 == t.v0, t2.v1 == t.v1, t2.v2 == t.v2 do
      assert(t == t2)
    end
  end
end
```

This invariant loops over all triangles with the same vertices (a multi-constraint query-loop) and complains if it finds anything other than the original triangle.

Having added this invariant, the `EdgeSplit` operation still compiles, but `EdgeCollapse` doesn't. The counter-example we get back to violate our new invariant is a well-known edge case, which Seam identifies automatically: a triangle mesh on four vertices, forming a tetrahedron (Figure 2.2).

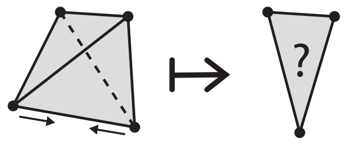


Figure 2.2: Tetrahedral Collapse Edge Case

When one of the edges is collapsed, two triangles are deleted, but the remaining two triangles coincide on the same three remaining vertices. In some papers [DBG14] this issue is addressed by deleting one of the two triangles, effectively merging them. In other papers [BB09] this case is handled by deleting both triangles. Which rule is appropriate depends on whether the mesh is modeling interfaces like the film of a soap bubble (the former) or interfaces like a water-air boundary. Seam directs the programmer's attention to such edge cases, rather than trying to automatically resolve fundamentally application-specific decisions.

Finally, this example illustrates how Seam helps support the safe evolution of data structure code over time, as application requirements change. Consider the case of the ArcSim code base, which was used in the publication of three successive research papers [NSO12, NPO13, PNdJO14], on cloth, paper-folding, and paper-tearing simulation respectively—all relying on triangle mesh edge-based remeshing. As the code base evolved, the schema changed. Initially, the triangle mesh (implemented as a C++ data structure) resembled our `TriMesh` schema, without views. However, by the third paper, a new requirement had been added. The triangle mesh would have to track both its original topology as a planar sheet, as well as its torn topology. A single "node" on the plane could now correspond to one or multiple vertices in the torn topology, depending on whether the node was located on a tear. We could easily model this extension in Seam, by declaring a new `Node` table, and adding a `node` field to the `Vertex` table. Then, instead of having to manually reason about the effects of this modeling change on all existing operations, we could rely on the compiler to direct our attention to the minimal set of edge-cases that need to be addressed.

2.4 Gong Examples (Topology from Collision Detection)

Gong (the third language of this dissertation) is concerned with the computation of new topological data (connectivity/locality) from metric data. Such problems usually arise in the form of *collision detection*. Loosely speaking, these problems tend to have the form "given two sets of objects, find all intersecting pairs" where "intersecting" has a geometric meaning specified by an intersection predicate.

The idea of Gong is to render these computations into a relational formulation as *spatial joins*. This idea (1) allows for a straightforward integration into Ebb and Seam (which rely on relational data models), and (2) exposes potential parallelism by abstracting the question of what we want computed from the precise order or way in which it is computed.

2.4.1 Sphere-Sphere Collision

Consider the following simple collision detection problem: compute all pairs of intersecting spheres, where all spheres are of idential size/radius. While the problem is austere, it lies at the core of a yarn simulation usable by knitters to visualize the consequences of different knitting patterns[LWS$^+$18]. To encode this sphere-intersection problem we must (1) describe the input and output data; (2) describe the desired join computation.

As with Ebb, we describe data via a *schema*. In addition to the table for our input data (`Spheres`), we declare a `Contacts` table to hold the results of the join. Each row of this output table is a different output pair, as indicated by its `s0` and `s1` values. However, our computation can also easily record other useful information from the collision, such as the `normal` vector of the contact between the two objects.

```
local gong schema SphereWorld
  table Spheres
  table Contacts

  field Spheres.pos : vec3f
  field Contacts.s0 : Spheres
  field Contacts.s1 : Spheres
  field Contacts.norm : vec3f

  global radius : float
end
```

Given this schema, we can express a `join` computation as a function, whose first two arguments are rows of the two tables being joined—in this case one and the same table: `Spheres`. We can think of the body of this join as executing inside of a doubly-nested loop over the set of `Spheres`. In the event that the join is a *self-join* between a set and itself, (as it is here) the join will only loop over unique pairs (a, b) and not their duplicate permutation (b, a). The `if` conditions filter out all *non-intersecting* pairs. And then if all of these tests are passed, the `emit` effect is triggered, outputting a row into the set of `Contacts`.

```
gong join SphereWorld.sphere_self_isct(
  s0 : Spheres, s1 : Spheres
)
  if s0 != s1 then
    var d  = s1.pos - s0.pos
    var d2 = d[0]*d[0] + d[1]*d[1] + d[2]*d[2]
    if d2 <= 4*radius*radius then
      emit { s0=s0, s1=s1, norm=d/sqrt(d2) } in Contacts
    end
  end
end
```

2.4.2 Particle-Scene Collisions

The same concepts can be used to encode continuous collision detection between very different kinds of objects. For instance, given a set of moving particles, and a static triangle mesh, we can compute the first triangle (if any) each particle collides with. First, we declare a schema as before. However, rather than form a table of output contacts, we declare a per-particle triangle to encode the output topology[12].

```
local gong schema ParticleScene
  table Particles
  table Vertices
  table Triangles

  field Triangles.v    : Vertices[3]
  -- pos0, p0 at initial time ; pos1, p1 at final time
  field Vertices.pos0 : vec3f
  field Vertices.pos1 : vec3f
  field Particles.p0   : vec3f
  field Particles.p1   : vec3f
  field Particles.tri : Triangles
  field Particles.collide_time : float
end
```

Here we omit the standard ray/edge-triangle intersection arithmetic. More importantly, if a collision is found for a particle, we want to record only the least collision time and for that collision, which triangle was collided with. Gong provides the `argmin=` operator for this purpose.

```
gong function ParticleScene.find_collision(...) ... end

gong join ParticleScene.particle_ccd(
  p : Particles, t : Triangles
)
  var success, time, pt = find_collision(p,t)
  if success then
```

[12]Doing this requires us to store meaningless triangle data to begin with and for every particle that does not collide. If I added `Maybe` types or `nullable` references to Gong, we could express the "not valid" concept directly. In the interest of avoiding unnecessary complexity in the prototype, I deferred this issue.

```
    p.collide_time argmin= { collide_time = time, tri = t }
  end
end
```

As we saw in Ebb, directly reading and writing fields is not generally safe to do in parallel. For each particle `p`, there are potentially as many independent threads of computation as the number of `Triangles`. Therefore, we must somehow safely mediate their parallel contention over the same output value. The `argmin` operation does this by defining the *least* thread with respect to the value of the reduced field `collide_time`. In this example, the *first* triangle (in terms of the simulation time, `collide_time`, not in terms of order of computation) that the particle collides with gets to write its values.

While the particle-triangle continuous collision problem is framed as a collision detection problem, it is very formally similar to ray-tracing/rendering visibility queries. This connection can help illuminate some properties of the `argmin` *effect*. While the term *armin* comes from optimization problems, it has the same computational role here as a Z-buffer does—guarding write-access to a number of other output "framebuffers"/`field`s. Likewise, the same "glitch" affecting early rasterizers affects our definition: Z-buffer fighting. Unless we can guarantee that every particle-triangle pair produces a distinct `collide_time` value, the join output remains underspecified, and hence non-deterministic. Internal row identifiers can be used to help stabilize such non-deterministic behavior.

Lastly, observe that `argmin` may be implemented in signficiantly different ways depending on the kind of hardware support available. If the join computation is only parallelized over the `Particles` table, then no concurrency control is necessary at all. Otherwise, atomic operations can be used to implement lock-free reductions. At the most extreme end, given the ability to design custom hardware support, we could exploit speculative hardware Z-buffer lookahead strategies to early exit from computations.

2.4.3 Persistent Box-Box Collisions

Finally, consider a more complicated example, simplified from a re-implementation of parts of Bullet[Cou15]. In order to focus on the interesting modeling choices and language features, we omit the functions `obb_isct`, `best_match`, and `refresh_depth`. Every contact maintains a cache of up to 4 contact points, acting as a surrogate for a more complex contact surface. Each of these points additionally stores a *Lagrange multiplier* value, which is computed externally to Gong as part of solving for contact forces.

```
local gong struct ContactPt {
  depth   : float
  pos     : vec3f
}

local gong schema BoxScene
  table Boxes
  table Contacts

  field Boxes.pos       : vec3f
  field Boxes.qrot      : vec4f
  ...
  field Contacts.b0     : Boxes
  field Contacts.b1     : Boxes
  field Contacts.norm   : vec3f
  field Contacts.n_pts  : uint
  field Contacts.pts    : ContactPt[4]
  field Contacts.l_mult : float[4]

  set   primary_key(b0,b1)  Contacts
end
```

Note the new `primary_key` directive/constraint. Giving a table a primary key means that each row must be *uniquely* identifiable by the specified field values. In other words, `Contacts` may contain at most one row for a given pair of `b0`, `b1` values.

Even though we abstract the majority of the complicated arithmetic into the `obb_isct` function, `find_contacts` contains a large, complicated piece of update code, replacing `emit` by `merge`. Why is this?

Getting high stability out of contact solvers is a challenging problem. One crucial trick is to propagate solutions from one frame of simulation to the next, a technique called *warm starting*. So long as a scene is static or mostly static, the normal forces (and hence Lagrange multipliers for contacts) will not change or not change much. However, when contacts break or form (a discrete change) this complicates the problem of how to correctly forward propagate the solution from the previous frame.

While working on Gong, I observed that this behavior was crucial to reproducing the behavior of Bullet, and introduced the `merge` effect[13] to account for the behavior. When `merge` is encountered, Gong looks for an existing row in the output table (`Contacts` here) whose `primary_key` value corresponds to the join arguments. Either this row is present (triggering the `update` case) or absent (triggering the `new` case). The `new` case behaves like `emit` in creating a new row. The `update` case binds the existing row to the specified variable (`c` here) and executes the specified code with exclusive access privileges to the row `c`, allowing unrestricted reads and writes to its immediate fields.

Lastly, I discovered that Bullet makes use of special behavior for rows that *were* present in the output table previously, but were not detected in the current execution of the join. These rows, which should be `remove`d, are not always removed. Sometimes, Bullet `keep`s those rows around—when the contact has been only slightly (less than `EPSILON`) broken. The remove case allows us to effect this behavior.

```
gong function BoxScene.obb_isct( ... ) ... end

gong join BoxScene.find_contacts( b0:Boxes, b1:Boxes )
  if b0 != b1 then
    var n_pts, norm, contacts = obb_isct(b0,b1)
    if n_pts > 0 then
      merge in Contacts
        new { norm = norm, n_pts = n_pts,
              contacts = contacts,
              l_mult = {0,0,0,0} }
```

[13]Some database systems call this kind of an operation *update-or-insert*, or *upsert*—names, neither of which I can stomach.

```
          update(c)
            c.norm = norm
            for i=0,n_pts do
              var j, is_eq  = best_match(c,contacts,i)
              c.contacts[j] = contacts[i]
              if not is_eq then c.l_mult[j] = 0 end
            end
            ...
            c.norm  = norm
            c.n_pts = n_pts
          end
          remove(c)
            var max_depth = -100
            for i=0,c.n_pts do
              var new_depth = refresh_depth(c,i)
              c.contacts[i].depth = new_depth
              max_depth max= new_depth
            end
            if max_depth > -EPSILON then
              keep(c)
            end
          end
    end end
  end
```

In order to ensure safe parallel execution of the join, we must impose specific restrictions on what can and cannot be done inside a `merge` effect. First, the table being targeted must have a `primary_key`. This constraint ensures that lookups for updates will never find more than one row, and simplifies the syntax as an added bonus. Second, the `update` and `remove` blocks are given "exclusive" read-write access to their arguments, equivalent to the variable argument of an Ebb function. Third, the `remove` block creates a fresh variable scope, since it must be safe to run *after* and independent of the execution of the rest of the join.

Across these examples, we can see not only considerable variation in what constitutes a collision detection problem/query, but also widely varying requirements imposed on how to output and store the results. This question of query *effects* is where we find most of the complication in determining whether or not a given join is parallel-portable, and in safely re-compiling for different kinds of parallel hardware.

2.4.4 Join Acceleration

The preceding examples specify logically accurate, and parallel portable joins, but do nothing to encode strategies for *accelerating* those joins. Usually we do not want to perform all nm explicit tests in order to find all intersecting pairs between tables of size n and m. Using a spatial data structure and algorithm (e.g. a BVH, k-d tree, grid, &c.) we ought to be able to do much better. How do we account for these *acceleration structures* from the perspective of joins? In databases, this question—how to make data more efficiently accessible—is de-coupled into the choice of *index* structure (e.g. B-tree, k-d tree, R-tree, &c.). While this connection is immediately obvious from listing the data structures in question, it remains implicit at best in the design of most graphics systems.

As a first example, suppose we want to accelerate the particle-mesh collision (§2.4.2) using a BVH of axis-aligned bounding-boxes. In Gong we do this by creating BVH indices on the argument tables, and then specifying a joint traversal of these BVHs. The BVH and traversal code is provided as a *template* specified inside of the Gong compiler. However, this template needs to be filled out with problem specific details.

Here, we supply a volume type (`AABB3f`) representing 3d single-precision axis-aligned bounding-boxes, and functions for interepreting the geometry of these volumes relative to the schema and to each other. That is, we provide an `abstraction` function `ParticleBox` to fit a bounding box around each moving particle; likewise with `TriangleBox`. Then we define the `union` of two boxes, whether or not they intersect (`isct`) and the `midpoint` of a box.

```
local gong struct AABB3f { lo : vec3f, hi : vec3f }

gong function ParticleScene.ParticleBox( p : Particles ) : AABB3f
  return { lo = min(p.p0, p.p1),
           hi = max(p.p0, p.p1) }
```

```
end
gong function ParticleScene.TriangleBox( t : Triangles ) : AABB3f
  var lo = min( t.v[0].pos0, t.v[1].pos0, t.v[2].pos0,
                t.v[0].pos1, t.v[1].pos1, t.v[2].pos1 )
  var hi = max( t.v[0].pos0, t.v[1].pos0, t.v[2].pos0,
                t.v[0].pos1, t.v[1].pos1, t.v[2].pos1 )
  return { lo = lo, hi = hi }
end
local gong function AABB3f_union( a : AABB3f, b : AABB3f )
  return { lo = min(a.lo, b.lo),
           hi = max(a.hi, b.hi) }
end
local gong function AABB3f_isct( a : AABB3f, b : AABB3f )
  return a.hi[0] >= b.lo[0] and
         a.lo[0] <= b.hi[0] and ...
end
local gong function AABB3f_midpoint( a : AABB3f )
  return 0.5 * (a.lo + a.hi)
end
```

Except for the abstraction of `Particles` and `Triangles` (application specific concepts) we can supply the volume and functions via a standard library.

Using these functions we can create indices for each table, and a joint traversal over the indices.

```
local ParticleBVH = BVH_Index {
  base_table  = ParticleScene.Particles,
  volume      = AABB3f,
  abs         = ParticleScene.ParticleBox,
  union       = AABB3f_union,
  point       = AABB3f_midpoint,
}

local TriangleBVH = BVH_Index {
  base_table  = ParticleScene.Triangles,
  volume      = AABB3f,
  abs         = ParticleScene.TriangleBox,
  union       = AABB3f_union,
  point       = AABB3f_midpoint,
}

local ParticleTraverse = BVH_BVH_Traversal {
  left  = ParticleBVH,
  right = TriangleBVH,
  isct  = AABB3f_isct,
}

ParticleScene.particle_ccd.set_traversal( ParticleTraverse )
```

The idea that we can safely plug code into a template without causing parallelism issues or exposing the entire compiler to the template system is of particular interest from a language design perspective. The Gong design accomplishes this feat by imposing the constraint that all template *parameter* functions are granted strictly *read-only* access to the underlying data. By then analyzing (the functions are written in Gong) *which* fields are read, the compiler programmer can enforce that none of the join effects come into conflict with the additional acceleration structure reads. So, even as we experiment with different accelration structure strategies, we are safeguarded from introducing new, unsafe race conditions.

That said, the compiler programmer cannot so easily guarantee the template itself and user-supplied functions are *correct* in the sense that they do not produce false negatives (pairs that are colliding, but are not tested). They must trust that (we) the simulation-programmers correctly wrote, say the `ParticleBox` function, without typos, &c.

To alleviate this tension, the Gong compiler-programmer can supply advanced debugging functionality. e.g. `ParticleScene.particle_ccd.verify_index(true)` will transform the `particle_ccd` join so that it runs twice. First, it runs with the BVH, causing normal effects, but additionally caching every *effectful pair* pair in a temporary output buffer. On the second time through, the join runs slowly, with a naive double-loop and effects disabled. Effectful rows are checked against the buffered results for discrepancies, which are reported back to us. This kind of systematic debugging support can be provided in Gong both because compilers can perform code transformations, and because the language understands (and can diagnose errors relative to) the specific semantics of join computations.

As an example of a different acceleration structure/index, consider our original `SphereWorld` example again. For each sphere, we can define a range (`lo`, `hi`) of grid cells that the sphere overlaps. And then given the integer coordinates for such a cell `key : vec3i`, we can define a function to hash that down into a linear key.

```
gong function SphereWorld.SphereRange( s : Spheres )
  var r     = {radius,radius,radius}
  -- inverse grid cell width
  var inv_w = 1.0f / (2.2*radius)
  var lo = floor( inv_w * (s.pos - r) )
  var hi = floor( inv_w * (s.pos + r) )
  return lo, hi
end

local gong function hash3i( key : vec3i )
  return bit_xor(   key[0] * 73856093,
           bit_xor( key[1] * 19349663,
                    key[2] * 83492791 ) )
end

SphereHash = Hash_Index(
  base_table  = SphereWorld.Spheres,
  key         = vec3i,
  abs_range   = SphereWorld.SphereRange,
  hash        = hash3i
}

SphereTraverse = Hash_Hash_Traversal(
  left        = SphereHash,
  right       = SphereHash
)
```

These functions allow us to define a spatial hash traversal strategy. Here the template and functions are different, but the same overall template architecture applies. We are able to incorporate our problem-specific knowledge (all spheres have identical radius) into the hash function, ensuring that each sphere overlaps no more than 8 grid cells in the worst case.

Chapter 3

Language Design

In the last chapter (§2) I introduced the three DSL prototypes from the perspective of a user—the simulation-programmer. In this chapter I will start to look at the languages from the perspective of the implementer—the compiler-programmer. Besides changing the point of view, I want to zoom out, and outline the limits of these languages. What is or is not expressible? What is a valid program? What are all valid programs?

This chapter provides two other good opportunities. Here, I can make explicit the idea that these DSLs were designed as facets of, as elaboration on a single language. We can ask precisely, "where and how do the languages differ?" And here I can discuss design trade-offs that arise from including or excluding different features. How do different designs shift burderns from the simulation-programmer to the compiler-programmer or vice-versa?

I will forgo a formal semantics in this chapter, relying on descriptions in English supplemented by formal syntactic grammars.

3.1 Base Language

I will describe the three prototype languages as variations on a single base language (Figure 3.1). A "program" will be described as a $\langle$*library*$\rangle$ consisting of a $\langle$*schema*$\rangle$ of $\langle$*schema_stmt*$\rangle$s and any number of supporting $\langle$*func_def*$\rangle$initions; the list of functions to `export` is explicitly stated. Basic ⟨*schema_stmt*⟩s either define a `table`, a `field`, or a `global`. For simplicity, we may assume all constants have been inlined.

Recursive functions are categorically prohibited.

```
     ⟨library⟩  ::=  ⟨schema⟩ ⟨func_def⟩* export ⟨name⟩*
      ⟨schema⟩  ::=  schema ⟨schema_stmt⟩* end
 ⟨schema_stmt⟩  ::=  table ⟨name⟩
                  |  field ⟨name⟩ . ⟨name⟩ : ⟨type⟩
                  |  global ⟨name⟩ : ⟨type⟩
    ⟨func_def⟩  ::=  function ⟨name⟩ ( ⟨arg⟩* ) [ : ⟨type⟩]? ⟨stmt⟩ end
         ⟨arg⟩  ::=  ⟨name⟩ : ⟨type⟩
        ⟨type⟩  ::=  bool | float | double
                  |  int8 | int16 | ··· | uint8 | ···
                  |  row( ⟨name⟩ )
                  |  vector( ⟨int⟩ , ⟨type⟩ )
                  |  struct { [⟨name⟩ : ⟨type⟩]* }
```

Figure 3.1: Basic Language Top-Level

Types are distinguished in two major ways. First, the unstructured types include primitive metric types (`bool` through `uint`) and the topological `row` type, which must name a `table` defined in the `schema`. Second, the types may be structured using arrays of constant size (`vector`s) and/or named `struct`s. For simplicity, we will assume that all row-types are non-structured, and that all metric types are `struct`-free, consisting of a single *base type* potentially packed into a vector, matrix, &c. My experience with varying these criteria across prototypes was that these kinds of details/features were often very important for convenience but not semantically fundamental to the designs.

Statements (Figure 3.2) fall into two groups. On the one hand, there are structural/control statements, and on the other there are the basic effect statements[1]

[1] in the concrete syntax, these effects statements simply appear as assignment and reductions such as `x.f = y` or `g += z`. I distinguish these statements here because it makes the semantics clearer.

```
⟨stmt⟩   ::=  ⟨stmt⟩ ; ⟨stmt⟩
           |  var ⟨name⟩ = ⟨expr⟩
           |  ⟨expr⟩ = ⟨expr⟩
           |  ⟨name⟩ ( ⟨expr⟩* )
           |  if ⟨expr⟩ then ⟨stmt⟩ [else ⟨stmt⟩]? end
           |  for ⟨name⟩ in ⟨query⟩ do ⟨stmt⟩ end
           |  while ⟨expr⟩ do ⟨stmt⟩ end
           |  return ⟨expr⟩
           |  write ⟨expr⟩ . ⟨name⟩ = ⟨expr⟩
           |  reduce( ⟨reduce_op⟩ ) ⟨expr⟩ . ⟨name⟩ = ⟨expr⟩
           |  global reduce( ⟨reduce_op⟩ ) ⟨name⟩ = ⟨expr⟩
⟨query⟩  ::=  query( ⟨name⟩ in ⟨name⟩ where ⟨qcond⟩ )
⟨qcond⟩  ::=  ⟨qcond⟩ and ⟨qcond⟩
           |  ⟨name⟩ . ⟨name⟩ == ⟨name⟩
```

Figure 3.2: Basic Language Statements

(`write`, `reduce`, `global reduce`). The `global reduce` statement must name a `global` variable from the schema. Meanwhile `write` and `reduce` statements target some field (.⟨*name*⟩) of a table T (the left ⟨*expr*⟩ must have type `row(`T`)`)[2].

In a later chapter (§4), we will examine effects explicitly: Which are allowed? Which are allowed together?

`if`, `while`, and `return` have the usually expected meanings. While I have omitted it here, the Ebb and Gong prototypes also included a `for` loop to iterate over fixed ranges (e.g. `for i=0,3 do ...`). This is generally a good idea since it makes compiler analysis and loop unrolling heuristics much simpler—even if only by carrying the form through to the back-end compiler.

[2]Some of my prototypes handled writing or reducing to specific entries of a `vector` or `struct` type; As mentioned, I will avoid these complications for the sake of clearer exposition.

Query loops (`for` over a $\langle query \rangle$) are constrained by the choice of query language. Our basic assumption is that every query filters a set (`x in X`) by a conjunction of inverse-field constraints. Namely, the $\langle qcond \rangle$ conjuncts must have the form `x.f == _`, which amounts to asking for the pre-image of `_` through the field `f`[3].

In addition to the form of query loop, we also disallow *unconstrained* query-loops. That is, a user cannot simply write `for x in X do ...` This constraint makes it difficult to write doubly-nested loops[4]. On the one hand, this design choice prohibits important forms of dense-matrix computations that simulation-programmers may want to express. On the other hand, this choice strongly encourages the simulation-programmer to write *local* computations, since loops must be constrained to explicitly connected elements.

Variable declaration and assignment (`var` and the following form) need to be carefully considered. At one extreme is the most compiler-friendly approach: eliminate the re-assignment statement altogether, turning `var` into effectively write-once `let`-bindings. At the other extreme is totally unrestricted re-assignment. I took a middle ground and allowed reassignment for metric-type variables. However, I disallowed reassignment of topological `row`-type variables, since doing so complicates analysis of the topology and locations of effects.

Disallowing topological re-assignment places an implicit bound (stencil size) on how many "hops" a function can take, since loops cannot be used to chain an arbitrary number of topological field reads. In tandem with the restrictions on query-loops, this constraint forces programs to be local in the sense of (grid-structured and unstructured) stencil computations.

The expressions from the base language (Figure 3.3) are even more straightforward. Besides literals and primitive operations, we have the ability to make function

[3]This specific design choice initially arose from implementation convenience (absent the conjunction) and was generalized to its current form in response to the analysis underlying the Emptyheaded database system[ATOR16]. That system handles even worst-case cyclic joins using a nested-loop structure by cleverly accelerating this primitive: loop over an intersection of pre-image sets.

[4]A clever simulation-programmer may realize that by creating a topological field `T.f` with a constant value `r` in whatever other `table`, the query `query(t in T where t.f == r)` is functionally equivalent to the whole table `T`; consequently the design advocated here does not make "non-local" access impossible—only highly unnatural. More precisely, it requires defining a topology in which everything is local—i.e. with a constant graph diameter.

```
       ⟨expr⟩  ::=  ⟨name⟩ | ⟨number⟩ | ⟨boolean⟩
                 |  - ⟨expr⟩ | not ⟨expr⟩ | ⟨expr⟩ ⟨op⟩ ⟨expr⟩
                 |  ⟨name⟩ ( ⟨expr⟩* )
                 |  { [⟨name⟩ = ⟨expr⟩]* } | ⟨expr⟩ . ⟨name⟩
                 |  { ⟨expr⟩* } | ⟨expr⟩ [ ⟨expr⟩ ]
                 |  read ⟨expr⟩ . ⟨name⟩
                 |  global read ⟨name⟩
  ⟨reduce_op⟩  ::=  + | * | and | or | min | max
         ⟨op⟩  ::=  ⟨reduce_op⟩ | - | / | % | == | != | < | >
```

Figure 3.3: Basic Language Expressions

calls, construct/destruct data, and read data from the schema (with the same typing constraints as for `write` and `reduce`). `struct` and `vector` constructors mimic the built-in Lua construction syntax for Lua-tables. `struct`s are destructured using field-name access, while `vector`s are destructured using indexing.

Unrestricted vector indexing can easily become an Achilles' heel for what is otherwise a memory-safe data-model. As compiler-programmers, either we need to construct a careful system of constraints to ensure that no index can ever be out of range, or we expose this problem to the simulation-programmer. In the prototypes I constructed, I never came to any conclusive decision on how to treat this problem. In Gong, I stress tested this problem by building out more robust support for small tensors than in any of the other languages. Despite additional type-checking I had a nasty bug that arose from an indexing error. One useful fallback for debugging is to support injection of range-bound tests into every access. Because the small tensors (`vector`) have constant size, bounds checking is at least well-defined.

3.2 Ebb (Stencil Language)

The main deviations in Ebb from the base language are to add grids and require explicit group-by directives. (Figure 3.4) Grids are defined `grid[n] foo`, where `n` is a constant specifying the dimension of the grid (usually `2` or `3`). Then, each grid dimension can be made "periodic" (i.e. granted wrap-around indexing) by issuing the `periodic[k] foo` directive to wrap-around the `k`-th dimension of the grid. In mixed situations like an infinite tube, the infinite direction can be made periodic, while the other directions (modeling walls) can be left with the normal behavior.

```
⟨schema_stmt⟩ ::= ...
               |  grid[ ⟨int⟩ ] ⟨name⟩
               |  subset ⟨name⟩ . ⟨name⟩
               |  set ⟨option⟩
     ⟨option⟩ ::= periodic[ ⟨int⟩ ] ⟨name⟩
               |  group_by ⟨name⟩ . ⟨name⟩
```

Figure 3.4: Ebb Top-Level

`subset`s may be declared in Ebb. As mentioned previously, we chose not to try to optimize the expression and handling of subsets[5].

The `group_by` directive may only be issued on `table`s, and may only be issued for at most one field of a table. In Ebb, I made the choice—debatable—to require explicit `group_by` directives. However, in the later prototypes (Seam and Gong), I relaxed this constraint. Let's consider the two positions here.

The first (Ebb) design exposes the fact that table data-layout can only be maximally optimized for access by at most one of potentially many fields at once. In databases, this concept is often refered to as *clustering*. A table may be *clustered* on at most one column, implying that it is sorted (or at least roughly sorted) on the values in that column. Consequently, memory-access locality is tied to the clustered column values. Maintaining a table in this clustered order further eliminates one additional layer of indexing data and one additional memory load (per looped row) from

[5]Of course, bounded range subsets of grids in particular would enable additional implementation and analysis opportunities. For instance, dependent partitioning[TBS$^+$16] in Legion considers a more elaborate algebra of subsets; ZPL especially makes the expression of grid regions a primary concern[Cha01].

the execution of query-loops. An explicit design goal of Ebb was to give simulation-programmers sufficient control to ensure high-performance programs. Hiding these trade-offs would seem to violate this goal.

In Ebb, additional `group_by`s consequently have to be built using auxiliary tables. Suppose we have a table with a field `X.f : Y` which is not the field it is grouped on. Then, we can build a table `XfY` with fields `XfY.x : X` and `XfY.y : Y`. Each row of this table corresponds to a row of `X` (the `XfY.x` value for the row) and has the value `XfY.y == XfY.x.f`. Then this table may be `group_by XfY.y`. The query `query(xy in XfY where xy.y == _).x` then becomes a stand-in for the invalid query `query(x in X where x.f == _)`. This scheme does not really impose additional runtime costs, but is incredibly inconvenient for a programmer.

In Seam and Gong, the tables targeted by these kinds of reverse-lookups are assumed to have dynamic size/contents—elements are being created and/or deleted as a normal part of the computation. By contrast Ebb assumes mostly static topology (excepting point-location). When rows are being created and deleted, the strategy of keeping the table sorted stops being a preferrable default over adding a slight bit of indirection.

$$\begin{array}{rcl} \langle \mathit{expr} \rangle & ::= & \ldots \\ & | & \texttt{affine(} \ \langle \mathit{name} \rangle \ \texttt{,} \ \langle \mathit{expr} \rangle \ \texttt{,} \ \langle \mathit{expr} \rangle \ \texttt{,} \ \langle \mathit{expr} \rangle \ \texttt{)} \end{array}$$

Figure 3.5: Ebb Statements & Expressions

Along with grids, Ebb adds the `affine(G,A,b,c)` expression (Figure 3.5) to transform a key to an element of one grid into a key to another element of the same or another grid. The affine expression takes four arguments

- `G` the output grid. i.e. `affine(G,A,b,c) : row(G)`
- `A` and b a matrix and vector such that the coordinates of the output element are `A * c + b`; Both are required to be constant values, so that the compiler can analyze/inline them.

- `c` the input grid element

If the computed cell coordinates are out of bounds, then in each dimension, the value is either clamped or wrapped around. Enabling `periodic` switches from clamping to wrapping behavior. In this way, memory safe access to grids is ensured.

Execution Model In Ebb, exported functions must have a `row`-type as their first argument. These functions are then compiled to be invoked in parallel over that entire `table` or some `subset` of it. As such, our effect analysis will have to ensure that such paralellization is always safe, regardless of the specific machine being targeted.

3.2.1 Expressivity and Efficiency of Ebb Data Modeling

Ebb's data modeling relies on three key primitives

- topological `field`s express functional relationships in a way that is familiar to most programmers thanks to pointers/indices/references in other languages.
- `query`-loops/`group_by` make a minimal extension to topological fields sufficient to access variable numbers of neighbors.
- Lastly, `affine`-indexing/`grid` data allows for optimized access to highly structured data.

The choice of primitives in Ebb was a departure from most other language designs. Lower-level imperative languages with manual memory management often expose explicit pointers or indices amenable to arithmetic and explicit memory layouts. Higher-level object-oriented and functional languages often rely on opaque references backed by a garbage collector. Meanwhile, databases discard the idea of references altogether, instead relying on relations loosely linked by keys—which may or may not be present in another table.

First, let's consider Ebb relative to a language like `C` or Fortran. In these languages topological relationships like our fields are stored as arrays of either pointers or integers, used to lookup data in other arrays. Both representations allow for arithmetic, and with good reason! If we didn't have indexing arithmetic, it would be very difficult to efficiently access multi-dimensional gridded arrays. However, in the process we have lowered our level of abstraction too severely. As mentioned in the introduction (§1) it is precisely this freedom of data model that makes heap and pointer-analysis an obstacle to paralellizing these programs. Worse, it impedes many useful optimizations by hard-coding access patterns into the code. Data referenced by a pointer cannot be moved or sorted into a better order; nor can indexed arrays be re-arranged.

Higher-level languages with opaque references and garbage collectors have two shortcomings. For simulation data, they lose sight of the fact that elements are not simply objects of a common class, but constitute a set/collection, whose layout can be optimized on that basis. This tends to impede the use of column-storage layouts (parallel arrays) and other tricks. They also obstruct the use of indexing for grids. Still, because arrays are more or less essential for high-performance benchmarks, these languages then expose un-boxed array primitives as an escape hatch. The resulting code is similar to Fortran, and replicates the same problems there—often with less consistent integration into the rest of the language.

Databases achieve a higher level of set-oriented data abstraction, which is exactly what we are seeking. However, performance via SQL or general relational algebra interfaces is highly unpredictable. Besides the problem of ensuring an efficient query-plan—which may produce even asymptotic inefficiencies—SQL interfaces obscure reasoning about the relationship between code and expected memory throughput behavior.

By contrast the Ebb primitives require 1 load per field-access, 2 loads per query-loop and 0 loads per affine-index. Simulation-programmers used to reasoning about the performance of imperative code are unlikely to be surprised. They should be able to clearly relate the results of performance profiles to the code they write. Meanwhile, memory safety is guaranteed.

Expressivity of Ebb may be demonstrated by example, as in the case of the taxonomy of Figure 3.6; it may also be demonstrated formally. For instance, we can show that Ebb can model any directed one-to-many relationship from one `table` `X` to another (potentially the original table) `Y`. Using the aforementioned auxiliary table

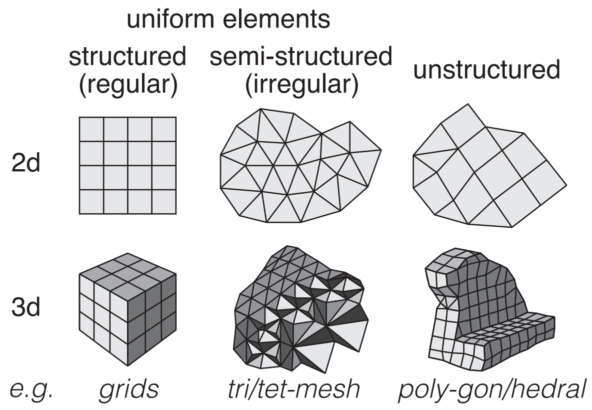


Figure 3.6: Ebb's relational model is sufficiently expressive to encode (at least) all of the geometric domains in this taxonomy, while also taking advantage of specific details like dimensionality or regularity. (This taxonomy is not exhaustive; Ebb can also efficiently express other domains, such as particles.)

strategy, we can create a table `R` with fields `R.x : X` and `R.y : Y`. Then, grouping `R` by `R.x`, we can easily query-loop over all elements `y` of `Y` related by `R`. Thus, we can be assured that Ebb loses no fundamental flexibility to express data relative to a standard relational model.

However, using auxiliary tables in this way is debatable (beyond just the aforementioned question of dropping the explicit `group_by`). On the side of the design I propose here is the principle of parsimony—no new feature need be added to the language. On the countervailing side is the objection that auxiliary tables will need to be created representing "things" that are unnatural to model. For instance, the auxiliary table supporting the concept "all triangles touching a given vertex" requires a list of "triangle-verts" which unlike triangles or vertices are not geometrically self-evident[6] elements of the mesh. Geometric domain libraries can help here to some degree—by abstracting over and hiding these kinds of non-intuitive parts of a data-model.

```
⟨schema_stmt⟩  ::=  ...
                 |  view ⟨name⟩ { [⟨name⟩ : ⟨table_name⟩]* }
                 |  viewdef ( ⟨arg⟩ ) ⟨stmt⟩ end
                 |  invariant ⟨name⟩ ( ⟨arg⟩ ) ⟨stmt⟩ end
   ⟨func_def⟩  ::=  ...
                 |  operation ⟨f_name⟩ ( ⟨arg⟩* ) ⟨stmt⟩ end
```

Figure 3.7: Seam Top-Level

3.3 Seam (Remeshing Language)

In Seam we chose to draw a sharper distinction between topological and metric data. In part this was practical. Allowing metric data to influence the logic of topological operations would have complicated formal verification goals for Seam. Ultimately, that may be a worthwhile compromise to make in service of ease-of-programming for simulation-programmers.

Seam substitutes `operation`s, `view`s/`viewdef`initions, and `invariant`s instead of Ebb's `function`s as the only function-like object. `function`s are retained in a very limited fashion. In order to `write` metric data to fields, simulation-programmers are allowed to call a *read-only* function, i.e. a function which reads but does not write or otherwise modify any data. These functions behave internally with Ebb-like statements and semantics. Having eliminated metric data computation from operations, we may then get rid of variable re-assignment, switching to `let` bindings, and simplify our expression language significantly.

However, Seam code has fundamentally different execution semantics. In Seam, sequencing of two statements s_1 `;` s_2 does not mean "do s_1 and *then* do s_2." Instead it means "Do s_1 and s_2 concurrently." That is, all effects inside of Seam operations are fully re-orderable. This is enabled using *log-execution* semantics: all effects are postponed (logged) until the completion of the operation, at which point they all take

[6]it is interesting to note that these are sometimes called triangle corners, and can serve as a useful basis for functions that are linear on triangles, but frequently discontinuous at edges or vertices.

⟨*stmt*⟩ ::= ⟨*stmt*⟩ `;` ⟨*stmt*⟩
~~| `var` ⟨*name*⟩ `=` ⟨*expr*⟩~~
~~| ⟨*expr*⟩ `=` ⟨*expr*⟩~~
~~| ⟨*name*⟩ `(` ⟨*expr*⟩* `)`~~
| `let` ⟨*name*⟩ `=` ⟨*kexpr*⟩
| `if` ⟨*cond*⟩ `then` ⟨*stmt*⟩ [`else` ⟨*stmt*⟩]? `end`
| `for` ⟨*name*⟩ `in` ⟨*query*⟩ `do` ⟨*stmt*⟩ `end`
~~| `while` ⟨*expr*⟩ `do` ⟨*stmt*⟩ `end`~~
~~| `return` ⟨*expr*⟩~~
~~| `reduce(` ⟨*reduce_op*⟩ `)` ⟨*name*⟩ `=` ⟨*expr*⟩~~
~~| `global reduce(` ⟨*reduce_op*⟩ `)` ⟨*name*⟩ `=` ⟨*expr*⟩~~
| `write` ⟨*kexpr*⟩ `.` ⟨*name*⟩ `=` ⟨*name*⟩ `(` ⟨*kexpr*⟩* `)`
| `new` ⟨*name*⟩ `:` ⟨*name*⟩
| `update` ⟨*kexpr*⟩ `.` ⟨*name*⟩ `=` ⟨*kexpr*⟩
| `delete` ⟨*kexpr*⟩
| `emit` `{` [⟨*name*⟩ `=` ⟨*kexpr*⟩]* `}` `in` ⟨*name*⟩
| `assert(` ⟨*cond*⟩ `)`
| `abort`

Figure 3.8: Seam Statements

effect simultaneously. Therefore, we can think of an operation as reading a single consistent view of the data, and planning a set of changes to make. These semantics free Seam programmers from having to reason about the *intermediate state* of the data in the middle of an operation.

In other words, Seam `operation`s are transactional.

Depending on whether we are in an `operation`, `viewdef`, or `invariant` different effects are available. A `viewdef` may only use the `emit` effect to output rows into `view`s. An invariant similarly may only use the `assert` effect to state conditions which must be true. The remaining effects belong to `operation`s. These are: `write` (metric data), `new` (row in a table), `update` (topological data), `delete` (row in a table), and

```
⟨query⟩  ::=  query( ⟨name⟩ in ⟨name⟩ where ⟨qcond⟩ )
⟨qcond⟩  ::=  ⟨qcond⟩ and ⟨qcond⟩
           |  ⟨name⟩ . ⟨name⟩ == ⟨name⟩
 ⟨cond⟩  ::=  not ⟨cond⟩ | ⟨cond⟩ and ⟨cond⟩ | ⟨cond⟩ or ⟨cond⟩
           |  ⟨kexpr⟩ == ⟨kexpr⟩ | ⟨kexpr⟩ != ⟨kexpr⟩
           |  forall ⟨name⟩ in ⟨query⟩ . ⟨cond⟩
⟨kexpr⟩  ::=  ⟨name⟩
           |  read ⟨kexpr⟩ . ⟨name⟩
```

Figure 3.9: Seam Expressions

finally `abort`. The `abort` effect is provided for convenience. It causes the log to be destroyed without having any effect, and signals to the caller that something went wrong. Formally, it may be replaced with `if` statements, leading to a silently failing operation, but pragmatically it is better to have aborts be explicit.

In Seam `viewdef`s, it is possible to `emit` the same row into the same `view` more than once. Yet, as shown in the example of representing the edges of a mesh as a view, the desired behavior is usually to de-duplicate these resulting rows. We adopt this convention as the standard semantics for all Seam `view`s.

We also add an unfamiliar conditional-expression concept. `forall` works like a query-loop, but wrapped around a conditional-expression rather than a statement. This construct is needed in order to conditionally take an action based on the neighbors of an element—but without also acting once for each of those neighbors. In the negated form `not forall` ... `not` ⟨*expr*⟩ the expression is of course equivalent to "there exists" which allows for useful conditionals like "if there exists an element referencing the current element, then don't delete it." For example, detecting collapsing tetrahedra in edge-collapse operations (§2.3.2) requires this construct.

Note significantly that `query`s remain unmodified from Ebb. The same reasoning about locality applies to Seam. However, the practical consequences are somewhat different in scope. To see why, it helps to explain how each of the major function-like definitions are executed/used.

While invariants and views are kept internal, `operation`s are exported from Seam. Unlike exported Ebb functions, operations are compiled to be invoked on a single binding of its arguments at a time with locked, atomic, transactional access to any data touched. In this context, the locality of query-loops is equivalent to the idea that each operation touches only a local neighborhood of the argument element(s). As a result, these operations are good candidates for scheduling with fine-grained locking schemes.

However, when initial data is loaded into a Seam schema, `invariant`s must be checked and `view`s generated. To ensure these computations can be accomplished efficiently, `viewdef`s and `invariant`s are required to have exactly one `row`-type argument, which is implicitly looped over. Seam thus restricts simulation-programmers to expressing only locally derived data and locally verifiable properties.

In contrast to database systems which might express these concepts with queries or logics, Seam also implicitly requires the simulation programmer to provide an *explicit plan* by virtue of writing views and invariants in the form of pseudo-imperative code. If data takes too long to load, the simulation-programmer can directly reason about why that might be.

Seam has a much more flexible and challenging to implement data model compared to Ebb. Seam has (1) tables of varying size, (2) no `group_by` directives, and (3) will have to somehow incrementally maintain the views. Considered together, these abstractions leave a lot of room for the compiler to change and optimize the data representation. The flip side of this is that the simulation-programmer gets far less specific control over these choices.

Previous experience writing re-meshing code suggested that this was the right tradeoff to make. For most programmers, manually managing a complex mutable graph-like data structure becomes such a challenge in and of itself that little to no attention will be paid to alternative strategies if those strategies require sweeping changes through the code base. By raising the abstraction level to the level of schemas, Seam provides a way to handle complex mutable data without imposing the resulting data-representation complexity on other parts of a simulation.

3.4 Gong (Collision Detection Language)

$$
\begin{array}{rcl}
\langle \mathit{library} \rangle & ::= & \langle \mathit{schema} \rangle\ \langle \mathit{func\_def} \rangle^{*}\ \langle \mathit{acc\_def} \rangle^{*}\ \texttt{export}\ \langle \mathit{name} \rangle^{*} \\
\langle \mathit{func\_def} \rangle & ::= & \ldots \\
 & | & \texttt{join}\ \langle \mathit{name} \rangle\ \texttt{(}\ \langle \mathit{arg} \rangle^{*}\ \texttt{)}\ \langle \mathit{traversal} \rangle\ \langle \mathit{stmt} \rangle\ \texttt{end} \\
 & | & \texttt{set}\ \langle \mathit{option} \rangle \\
\langle \mathit{option} \rangle & ::= & \texttt{primary\_key(}\ \langle \mathit{name} \rangle\ \texttt{,}\ \langle \mathit{name} \rangle\ \texttt{)}\ \langle \mathit{name} \rangle
\end{array}
$$

Figure 3.10: Join Extension Top-Level

As another language largely driven by metric data, Gong remains closer to Ebb in more ways than Seam. It introduces the concept of a `join` instead of either `function`s or `operation`s. At their simplest, these `join`s may be thought of as special *doubly-nested* loops. As such, the first two arguments of each join must be of `row`-type, specifying (similar to exported Ebb functions) which two `table`s are looped over—potentially which single table is looped over twice. This situation changes which effects are safe, and the intended application (collision detection) changes which effects are necessary.

For instance, the inner loop of the double-loop does not grant exclusive access to the first nor second element being looped over, and so directly writing to those rows is disallowed. Reducing is allowed however, which we saw expanded via the added `argmin` construct.

For simplicity of exposition, assume that there is a single join written in Gong, joining some number of *base tables* and potentially changing the contents of some number of *derived tables* whose contents are computed by the join itself. In this view of things, there will generally be topological `field`s on the derived tables pointing back into the base-table portion of the schema, but there will not be fields going the other way. These assumptions may be relaxed as overly strong, but they will let us more easily explain the major kinds of effects joins may have without worrying too much about pathological cases.

Consider a join that has at most a single effect for every input pair (a, b). This effect may either be (1) a reduction `reduce(`*op*`)` *a* `.f = ...` into a field of some base table, an `argmin` reduction into a base table, (2) an `emit` producing an output row of a derived table, or (3) a `merge` controlling updates to a derived table.

In the case of reductions, the effect is safe, so long as the table being reduced into (more precisely the fields being reduced into) is not also being read or targeted by an `emit` or `merge`. In the case of `argmin`, the right-hand-side expression must be a record-literal specifying which fields must be updated and with what values (including the field being minimized over). Unlike Ebb, this construct allows topological fields to be written to. Doing-so may invalidate internal data-structures built for supporting query-loops.

```
⟨stmt⟩  ::=  ...
          |  argmin ⟨expr⟩ . ⟨expr⟩ = ⟨expr⟩
          |  emit ⟨expr⟩
          |  merge ⟨name⟩
                new ⟨expr⟩
                update ( ⟨name⟩ ) ⟨stmt⟩ end
               [ remove ( ⟨name⟩ ) ⟨stmt⟩ end ]?
          |  keep ⟨expr⟩
```

Figure 3.11: Join Extension Statements & Expressions

An `emit` effect will cause its target table to be *cleared* of its contents prior to running the join. Consequently, the emitted table will be re-built every time the join is run. This behavior may cause problems if the target table is used as the base table for another join, since the results of that join will immediately become invalid—they may contain dangling pointers to no longer existing rows. For this reason, I disallowed a table from being both a source of and a target of different joins. Unlike `argmin`, `emit` must generate a value for every defined field.

A `merge` effect targets a derived table which can be uniquely indexed by the pair of keys that the join is looping over. (see below for a longer discussion of `primary_key`) `merge` may occur at most once inside of a join, and may not occur inside of a loop or other construct which could cause it to be executed more than once for a given pair of keys.

Let S denote the set of rows/key-pairs in the table targeted by `merge` prior to running the join. Let S' denote the set of rows/key-pairs which reach the merge statement when executing the join. Then $R = S - S'$ is the set of rows to be `remove`d; $N = S' - S$ is the set of `new` rows; and $U = S \cap S'$ is the set of rows to be `update`d.

The `new {` $name_1 = expr_1, \ldots$ `}` behaves like `emit` for any previously absent row. It must specify the values of all fields on the target table except the `primary_key` fields, whose values are just the join parameters. (They are automatically filled in.)

The `update(c)` block runs for every already existing row and grants exclusive access to this row, allowing for arbitrary read/write effects on its fields. These read-/write effects may not change the `primary_key` of the row.

The `remove(c)` block runs without any enclosing lexical scope on every row to be removed. Because it runs without this outer scope, it may only refer to the row to be removed and data accessible on it alone. It may optionally choose to `keep` the row, which will prevent it from being removed. In that event, the remove block may arbitrarily rewrite the row's data, excepting the `primary_key` which remains immutable.

Ensuring that `primary_key`s are maintained requires that *all* joins maintain them. Therefore if a table has a `primary_key`, then all joins `emit`-ing into that table are subject to similar restrictions as `merge`. Only one such `emit` may exist in a join, and strictly outside of loops. Furthermore the values of the primary key fields are omitted from the record specifying the contents of the emitted row—they are instead inferred from the join parameters

3.4.1 Acceleration Structures

Every join must be supplied with a traversal strategy, which itself refers to acceleration structures. By default, this is simply `scan`, which performs the naive double-loop. If a `join` is a self-join, then for a given pair (a, b) that is scanned over, (b, a) will not also be scanned over. The scan traversal strategy has the virtue of always being correct. Therefore it can serve as a point of comparison for differential testing.

In principle any number of other spatial data structures and traversal algorithms could be incorporated. However, I chose to supply two parameterized strategies: bounding volume hierarchies (BVHs) and hash tables. The latter is sufficient to represent both grids and spatial hashing strategies depending on how it's parameterized.

```
  ⟨acc_def⟩  ::=  scan ⟨name⟩ ( table     = ⟨table name⟩ )
               |  BVH  ⟨name⟩ ( table     = ⟨table name⟩
                                volume    = ⟨type⟩
                                abstract  = ⟨func name⟩
                                vol_union = ⟨func name⟩
                                point     = ⟨func name⟩ )
⟨traversal⟩  ::=  scan_scan ( left = ⟨acc name⟩, right = ⟨acc name⟩ )
               |  bvh_bvh   ( left = ⟨acc name⟩, right = ⟨acc name⟩
                              vol_isct = ⟨func name⟩ )
```

Figure 3.12: Join Scan/BVH-Acceleration Structures & Traversals

The BVH interface (Figure 3.12) is based around `abstraction` of rows from a `table` into `volume`s. The rest of the `function`s supplied make the designated `volume` behave like an object class in an object-oriented language. The `volume` type may be arbitrarily selected. For instance,

- A 3-d axis-aligned bounding-box
  `AABB3f = struct { lo : vector(3,float), hi : vector(3,float) }`

- A 7-degree-oriented-polyhedron
  `DOP    = struct { lo : vector(7,float), hi : vector(7,float) }`

- A bounding sphere
  `Sphere = struct { pt : vector(3,float), radius : float }`

If `table = A` and `volume = V`, (`V_L` and `V_R` distinguishing which side of a traversal a volume is from) then the types of the different functions are required to be

```
 abstract  :  A         -> V
vol_union  :  V x V     -> V
    point  :  V         -> vector(3,float)
 vol_isct  :  V_L x V_R -> bool
```

Some volumes and associated functions are supplied in the standard Gong library (e.g. axis-aligned bounding-boxes) reducing the minimum application-specific interface to supplying the `abstract` function. Even if we specialized the interface down towards axis-aligned bounding-boxes (or some other scheme) the simulation-programmer will always have to supply at least some information to allow their idiosyncratic data schema to be interpreted by any given acceleration structure scheme.

The full set of constraints for these functions to be sound is too tricky to check via most any kind of formal method, even if they can be stated formally. Consider a predicate $P(a,b)$ which is true whenever the pair $(a,b) \in A \times B$ causes some effect in a given join. In general the traversal/acceleration scheme must ensure that it traverses over every pair (a,b) where $P(a,b)$. Such a traversal is *sound*.

Let $\alpha(x)$ be shorthand for `abstract(x)`, $\cap(x,y)$ be shorthand for `vol_isct(x,y)`, and $\cup(x,y)$ be shorthand for `vol_union(x,y)`. We say that `vol_isct` is sound relative to `abstract` iff. $P(a,b) \implies \cap(\alpha(a),\alpha(b))$. We say that `vol_union` is sound relative to `vol_isct` iff. $\forall a,a',b.\ \cap(a,b) \implies (\cap(\cup(a,a'),b) \wedge \cap(\cup(a',a),b))$. The lack of commutativity and/or associativity for `vol_isct` and `vol_union` may lead to non-determinism in the effectiveness of the traversal, but should not invalidate soundness. The behavior of `point` helps guide the acceleration structure construction, so while it may affect quality it ought not affect soundness even if it just returns a constant coordinate triple.

Currently, the choice of traversal and build strategies are left up to the implementation without simulation-programmer input or control (outside of the above functions).

The BVH builds a tree by recursively covering objects with bounding volumes. By contrast Hash structures attempt to partition the set of intersections, potentially

```
  ⟨acc_def⟩  ::=  ...
              |   Hash ⟨name⟩ ( table        = ⟨table name⟩
                                key          = ⟨type⟩
                               [abs_range    = ⟨func name⟩]?
                               [abs_point    = ⟨func name⟩]?
                                hash         = ⟨func name⟩
                               [brute_force  = ⟨func name⟩]? )
                               [BIN_TO_ROW   = ⟨number⟩]? )
⟨traversal⟩  ::=  ...
              |   hash_hash ( left = ⟨acc name⟩, right = ⟨acc name⟩
                              hash_first = ['left'|'right'] )
```

Figure 3.13: Join Hash-Acceleration Structure & Traversal

introducing duplication of rows. Usually, this splitting is based on a partition of space—which is why some objects may straddle multiple bins of the partition.

The Hash interface (Figure 3.13) relies on a `key` of `int` or `vector(k,int)` type, expressing a logically distinct hash location. A given row of the `table` is abstracted into a single key (`abs_point`) or range of keys (`abs_range`). Then keys are converted into linear bin ids via a `hash` function. As an escape hatch, a row of the `table` can be identified as requiring `brute_force` instead, preventing it from being converted into an unreasonably large range of keys (e.g. preventing hashing a ground-plane object). The size of the hash-table (i.e. number of linear bins) can be set as a linear multiple (`BIN_TO_ROW`) of the number of rows in the underlying `table`.

A traversal between two hash-tables must have identical choices of `key`, but may have differing choices of `abs_point` or `abs_range`, as well as different `brute_force` predicates. The choice of which side to `hash_first` will affect which hash table gets materialized; which side probes the other side's table.

If `table = A` and `key = K`, then the types of the functions in the interface must be the following. Either `abs_range` or `abs_point` may be supplied, but never both. And `brute_force` is optional—when not supplied, it is as if `brute_force` always returns false.

```
  abs_range  :  A -> K x K
  abs_point  :  A -> K
       hash  :  K -> uint
brute_force  :  A -> bool
```

As with the BVH, `hash` functions may be supplied in a standard library, but any function of the underlying `table` (i.e. `abs_range` or `abs_point`) inherently depends on the simulation-programmer's idiosyncratic schema, and must be supplied. Also like the BVH, this may lead to incorrectness in the traversal beyond the ability of formal methods to automatically check for problems. Still we may give soundness criteria relative to $P(x, y)$.

Given a hash on table X, let $A_X(x, k)$ be a relational-predicate between rows x of X and key-values k. If `abs_range` is supplied, then $A_X(x, k)$ is true iff. k is in the range of `abs_range`(x). If `abs_point` is supplied, then $A_X(x, k)$ is true iff. `abs_point`$(x) = k$. Let Y be the right-hand-side of the join and $A_Y(y, k)$ be the associated relation-predicate. Then we may say that the two hashes are jointly sound relative to the predicate P iff. $P(x, y) \implies \exists k.\ A_X(x, k) \wedge A_Y(y, k)$. In other words, there must always be some matching key shared between two rows that ought to be joined.

Chapter 4

Effects

I use effects to abstract the ways in which a piece of code accesses the data model. Such an abstraction serves at least two purposes. First, it abstracts the data model from the perspective of the simulation code. This allows the compiler-programmer to reason about whether or not a given set of effects is safe to execute in parallel—without having to be aware of the details of the data model's implementation. So, as compiler-programmers, we can implement *effect checking*, a sound race detection analysis, independent of our choice of data structure and compile target. Second, it abstracts the code from the data model, so that the compiler-programmer knows which primitive operations they need the data structure to expose without having to worry about control, looping and other code structures.

Especially, note that given a collection of Ebb, Seam and Gong programs, the aggregation of all effects used defines a set of requirements for a common schema representation that could be shared between them.

4.1 Basic Effects

The following is a list of all basic effects that we will want to keep track of. These effects may be parameterized by the table (`tbl`) they act on, the field (`fld`) they act on, and the specific `row` they act at, among other parameters. On the one hand, we can view these as argument parameters to code macros, provided by the data model to the code generator. On the other hand, by statically abstracting dynamic values (e.g. `row`) we can view these effects as symbolic representations of privileges.

`SCAN(tbl)`	loop over `tbl` in parallel
`IDX_SCAN(tbl,fld,row)`	loop over entries in `tbl` where `tbl.fld == row`
`READ(tbl,fld,row)`	read the value of `tbl.fld` at `row`
`REDUCE(op,tbl,fld,row,val)`	`op`-reduce into `tbl.fld` at `row`
`WRITE(tbl,fld,row,val)`	write into `tbl.fld` at `row`
`GLOBAL_READ(glob)`	read the value of `glob`
`GLOBAL_REDUCE(op,glob,val)`	`op`-reduce into `glob`
`NEW(tbl)`	create a new row in `tbl`
`UPDATE(tbl,fld,row,val)`	write a new topological value into `tbl.fld` at `row`
`DELETE(tbl,row)`	remove the `row` from `tbl`
`ASSERT_ERROR()`	raise a failed assertion error
`EMIT(tbl,new_rec)`	add a row to `tbl` with the contents `new_rec`
`MERGE(tbl,row0,row1)`	see below
`KEEP(tbl,row)`	see below
`ARG_MIN(tbl,fld,row,val,new_rec)`	overwrite `tbl.fld` at `row` if `val` is smaller, and overwrite the field values in `new_rec` too. Similarly for `ARG_MAX`

Figure 4.1: Effect Definitions

4.1.1 Bagging up Effects

To a more precise degree than in either Ebb or Gong, Seam's semantics allow us to describe not just effects occuring, but to describe *multisets/bags/relations* of effects caused by an operation, view, &c. That is, rather than an isolated invocation of an effect on an isolated row, we can describe an entire relation with each row corresponding to another invocation of the given effect. For instance, deleting every "follow" connected to an "account" a in the social network schema could be expressed as

$$\{\ \texttt{DELETE}(\texttt{Follow}, f) \mid f : \texttt{Follow}, f.\texttt{src} = a\ \}$$

In Ebb and Gong, we care much less about this kind of precise quantification of where an effect is occurring. Instead, we simply want to know if it is happening (1) multiple times, and (2) whether it is happening at the primary loop-variable row or not.

Given a `function` that loops over `a : A`, we say that `EFFECT(a)` is *centered*. If the effect occurs within a loop, we say it is *looped* and if it occurs more than once by any other means we say it is *repeated*. Using this vocabulary, we can describe the rules for effect-checking.

4.2 Effect Checking

Effect-checking is based on a syntactic pass over the code to extract effects associated with specific statements. These translations are succinctly captured by a simple tabulation. (Figure 4.2) Depending on the language and especially its code execution model, we have to make different considerations for which combinations of effects are safe to occur simultaneously/in-parallel.

Tagging Effects When doing effect-checking in Gong or Ebb, we need to annotate the effects with additional information. If an effect occurs inside of any loop, we tag it with the `looped` modifier. In an Ebb function, we track the parameter `a:A`, allowing us to tag all reads, writes, and reductions to it. Such effects are tagged `centered`. In a Gong join, we do not track the parameters `a:A,b:B` for centering, since we expect the same row to be referenced in multiple threads of the double-loop. However, once we enter an `update` or `remove` block in a `merge`, that parameter `x:X` is tracked the same way as the first argument to an Ebb function, tagging effects that use it as `centered`. We also track whether effects occur inside of a Gong `update`/`remove` block, tagging them as `in_update` or `in_remove`, respectively.

⟨*stmt*⟩	*effect*
function (a : A, ...) ... end	SCAN(A)
for x in X where x.f == y do ... end	IDX_SCAN(X,f,y)
write r.f = v, (*with* r : R)	WRITE(R,f,r,v)
reduce(*op*) r.f = v, (*with* r : R)	REDUCE(*op*,R,f,r,v)
global reduce(*op*) g = v	GLOBAL_REDUCE(*op*,g,v)
read r.f, (*with* r : R)	READ(R,f,r)
global read g	GLOBAL_READ(g)
new x : X	NEW(X)
update r.f = v, (*with* r : R)	UPDATE(R,f,r,v)
delete r, (*with* r : R)	DELETE(R,r)
emit *rec* in X	EMIT(X,*rec*)
assert(false)	ASSERT_ERROR()
join (a : A, b : B, ...) ... end	SCAN(A), SCAN(B)
argmin r.f = *rec*, (*with* r : R)	ARGMIN(R,f,r,*rec*.f,*rec*)
merge X	MERGE(X,a,b,*rec*)
new *rec*	
update (x) ... end	
[remove (x) ... end]$^?$	
keep r, (*with* r : R)	KEEP(R,r)

Figure 4.2: Effect Extraction

4.2.1 Ebb

Consider effect-checking of a `function` to be exported, with first parameter `a : A`. Under the Ebb execution model, we should imagine that this function will be simultaneously called for every `a` element of `A`. Assume for simplicity of exposition that all other function calls have been in-lined so that we are dealing with a single statement-block of code.

In Ebb, we have a collection of read-only effects `SCAN`, `IDX_SCAN`, and `READ`, in addition to `REDUCE` and `WRITE` effects, with global variants of read and reduce. As such, we can produce potential conflicts between reads, reductions and writes. The following 3 cases outline the only assuredly safe situations:

- If a data location is only read, there is no conflict.
- If a data location is only reduced to using a single operator, then there is no conflict.
- If a data location is only ever accessed by one thread, then there is no conflict.

In all other cases, re-ordering the execution of individual effects will cause the resulting value at the data location or the result of a read-effect to differ.

(`SCAN` and `IDX_SCAN` are mostly inert here, because they do not touch the actual field data. In the case of `IDX_SCAN` the field(s) used by the query may be considered "read" although since we have ruled out modifications to topological fields, the point is moot.)

Analysis may proceed on a strict per-field basis. For each field, we first check whether all effects are `centered`. If so, we have exclusive access and are done. Failing that, if all effects are reductions using the same operation we are ok. Failing that, if all effects are reads we are ok. Otherwise an error is reported. Two conflicting effects may be selected for the error report, directing the simulation-programmer's attention to two definitely conflicting sites in the code.

Global reads and reductions follow the same rules, with the exception that centered accesses are impossible.

After the development of Ebb, I realized that there is a further, more subtle problem posed by the semantics of query-loops, which remained unhandled. Centered effects inside of a query-loop might cause the result of the computation to depend on the order in which the query loop executed. So, while the results obtained from the prototype were likely to appear deterministic, the semantics specify more flexibility. The query-loop should be able to iterate the contents of the query in any order.

One way to resolve this shortcoming is to imagine that a thread encountering a query-loop forks into multiple threads for each iteration of the loop body. As such, the formerly centered argument variable `a` is not centered—in the sense that it is in one-to-one correspondence with the threads. This can be hacked back into the analysis by dropping the `centered` tag whenever the `looped` tag is present.

This approach is sound, but now rules out some perfectly safe computations. Consider the following:

```
ebb function TriMesh.F_Update( v : Vertices )
  v.F     = {0,0,0}
  for e in v.edges do
    v.F  += K * (e.hd.pos - v.pos)
  end
end
```

The proposed modification to effect-checking will prohibit this function because it now contains a centered-write combined with an uncentered-reduction of the field `F`.

The limitations of the informal approach to effect-checking presented in this chapter should now be more apparent. A more careful, formal treatment of this notion of effect-checking might uncover other inconsistencies or needless restrictions.

4.2.2 Gong

Consider effect-checking of a `join` to be exported, with parameters `a:A` and `b:B`. Unlike with Ebb functions, it would be incorrect to tag effects on these parameters with `centered`. In the Gong execution model, a join may be thought of as one thread simultaneously launched for each `a`,`b` *pair*, meaning that each `a` or `b` value is necessarily referenced by multiple threads. As with Ebb, we may assume that all functions used in the join have been inlined for simplicity of exposition.

Acceleration Structures. Unlike Ebb, we have a unique feature: acceleration of double-loops (i.e. `join`s) using spatial data structures. These structures are parameterized by Gong functions which are passed to them, which cannot simply be inlined for the purposes of effect analysis. These functions are constrained to be *read-only* in their effects. Therefore, we can abstract their effect by adding all of their extracted read effects into the effect analysis of every join using the acceleration structure.

We have read, reduce and write effects—like in Ebb. However, we also have `EMIT`, `MERGE` and `ARG_MIN` effects, which significantly complicate our analysis. We have to revisit our dismissal of `SCAN` effects in light of this. On top of that, we have the additional `primary_key(f,g)` constraint on the data model that must be enforced.

Essential to this analysis is adding a refining concept to the data schema—one which we can extract automatically. We tag a table as `derived` if it is the target of an `EMIT` or `MERGE` in any join. The schema is then considered invalid if there is any `derived` table `Y` and some field `X.f : Y`. In other words, derived Gong tables may not be referred to by other tables. This constraint ensures that the deletion of rows induced by `EMIT` and `MERGE` does not produce dangling references/pointers/keys.

An `EMIT` targeting a table is mutually exclusive with any other kind of effect on the table, including all reads, reductions and writes, as well as `SCAN`/`IDX_SCAN` and `MERGE`. Likewise, a `MERGE` is mutually exclusive with any other kind of effect on the table—with a notable exception for effects contained within the `update` and `remove` cases within that `merge`. As mentioned previously, a merge on a table without a `primary_key` is prohibitted.

The `primary_key` constraint then acts back on the `EMIT` and `MERGE` effects, constraining their usage. Of note, the `EMIT` effect will always fill in the join parameters `a`,`b` for the primary key, as will the `MERGE`. This constraint (enforced by making those entries of the new-record implicit) ensures that no two threads of the join can emit/merge rows with identical primary-keys. Thus we recover an analogue of Ebb's centered accesses. In order to make this concept sound, we additionally prohibit multiple or `looped` `EMIT`/`MERGE` effects within a single join that target tables with `primary_key`s.

Digging into the body of `update` and `remove` cases of the `MERGE` effect, we are able to perform any number and kind of centered accesses to fields of the merged table `X`. We may additionally walk from there into the rest of the schema (though not query-loop back into `X`) where data accesses are handled by the prior uncentered rules. Of particular note, a `KEEP` effect is only valid when issued in a remove block on a centered variable.

Finally, we may reduce `ARGMIN` to an elaborate form of a reduction effect. Specifically, construct a signature consisting of `ARGMIN`, the field begin minimized over, and then every other field being written. Treat this signature as its own special reduction operation, and then apply the usual reduction rules. That is, an argmin will conflict with any other effect which is not identical in the sense of minimizing the same field and writing the exact same set of fields. Every field touched by the argmin may then be considered to be "reduced" by this special reduction operator.

These rules (excepting the schema constraints of `derived` tables) are collected by the following list.

- If a field is only read, there is no conflict.
- If a field is only reduced to using a single operator, then there is no conflict.
- A field may only have `KEEP` performed on it inside of a `remove` clause, and on a `centered` variable.
- A table targeted by `EMIT` or `MERGE` may not be `SCAN`/`IDX_SCAN`ned.
- A table targeted by `MERGE` may only have centered accesses performed upon it.
- A table with a primary key may be targeted by at most one `MERGE` or `EMIT` effect, which may not be looped.

4.2.3 Seam

Unlike Ebb and Gong, Seam has a very fine-grained analysis of effects, tied to a different execution model and more precise analysis of safe topology manipulations. While Seam has "derived" tables (i.e. `view`s), it seeks to allow complicated manipulations of the "base" tables directly—which is often unsafe.

With Seam, the idea that we can form multi-set relations of effects becomes explicit rather than partially-characterized by `looped`, `centered` and other tags. This translation is accomplished by a symbolic execution of Seam code[PBS+17]. The means of performing that translation and ensuring safety lie outside the scope of this thesis. However, I will state their guarantees relative to the execution model here.

Unlike Ebb or Gong, whose `function`s and `join`s can be viewed as loops, Seam `operation`s are applied one at a time, executing on specific elements of the data; the external view of a seam operation becomes sequential rather than data-parallel. At the same time, Seam changes how we interpret the code in the body of an operation relative to functions and joins. Seam operations are assumed to be fully atomic and transactional. Any output/mutable effects are postponed until the end of the operation's execution—when they are assumed to take simultaneous effect.

Therefore, we can think of a Seam operation as generating an effect log—a list of effect calls with specific values filled in. From a parallelism perspective, this allows us to encapsulate the safety of Seam programs by three maxims:

- The mutating effects of a Seam operation (i.e. not reads or scans) must not conflict—that is, any reordering of a valid effect log must cause the same change in the underlying data model. (note certain unavoidable constraints on the need to create `new` rows before referencing them.)
- The resulting data model must continue to be well defined. All tables must represent sets and all fields must represent (total) functions. e.g. there may not be any dangling references, since that would result in a partial function.
- The resulting data model must satisfy any user-specified invariants.

These principles and the effects of the code can be translated into first-order predicate logic on set-and-function models. In this way, SMT solvers[DRK$^+$14] can be leveraged to check the properties.

In the following chapter we will see how the ability to state clear multiset semantics for Seam code can also be used to implement incremental view maintenance as a source-to-source code transformation.

Chapter 5

Relational Semantics & View Maintenance

In this chapter, I will explain how the relational semantics of Seam programs enable an interesting approach to incremental view maintenance. To do this, I extend Christoph Koch's recasting[Koc10] of relational algebra (on multi-sets) as operations on a ring[1] of databases.

5.1 Preliminaries

5.1.1 Normalized Seam Views

In order to explain this transformation, we must first re-formulate the grammar of Seam views. I will assume certain normalizations of the code prior to the transformation. Using `let` expressions, all complex $\langle kexpr \rangle$essions will be broken down into primitive forms, analogous to single-static-assignment. `not` may be fully pushed down into $\langle cond \rangle$itions. `forall`-expressions are disallowed inside of `view`s.

Lastly, disjunctions of the form `if` A `or` B `then` s_1 `end`, pose an interesting point of flexibility for multi-set semantics. Because Seam views are de-duplicated, we can rewrite the conditional on a disjunction into a pair of conditionals

[1]in the abstract algebra sense

```
if A then s_1 end ;
if B then s_1 end
```

The simple set semantics of the two programs (assuming deduplication) are equivalent, even though their multi-set semantics are not. Therefore, we are permitted to perform this transformation so long as it is done in a consistent way for all code prior to multi-set interpretation—in both view-generation and view-maintenance.

Conjunctions in conditionals can be broken down without these worries by simply nesting if-statements. e.g. `if` A `and` B `then` s_1 `end` becomes

```
if A then
  if B then s_1 end
end
```

Using the prior tricks we may reduce the grammar of Seam views into a normalized form. (Figure 5.1) (You may assume for simplicity of exposition that each view-def `emit`s into a single, rather than multiple views.) In this grammar, I use ⟨*var*⟩, ⟨*field*⟩ and ⟨*table*⟩ as more informative synonyms for ⟨*name*⟩.

Note that this grammar converts `let` bindings from a terminal statement into a wrapping statement, analogous to `if` and `for`. This leaves the `emit` effect as the sole terminal statement. This transformation is justified by observing that each `let` binding is implicitly scoped over the remainder of the statement block it's contained in.

$$
\begin{array}{rcl}
\langle \textit{viewdef} \rangle & ::= & \texttt{viewdef(}\ \langle \textit{var} \rangle\ \texttt{:}\ \langle \textit{table} \rangle\ \texttt{)}\ \langle \textit{stmt} \rangle\ \texttt{end} \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{stmt} \rangle\ \texttt{;}\ \langle \textit{stmt} \rangle \\
& | & \texttt{let}\ \langle \textit{var} \rangle\ \texttt{:}\ \langle \textit{table} \rangle\ \texttt{=}\ \langle \textit{var} \rangle\ \texttt{in}\ \langle \textit{stmt} \rangle \\
& | & \texttt{let}\ \langle \textit{var} \rangle\ \texttt{:}\ \langle \textit{table} \rangle\ \texttt{=}\ \langle \textit{var} \rangle\ \texttt{.}\ \langle \textit{field} \rangle\ \texttt{in}\ \langle \textit{stmt} \rangle \\
& | & \texttt{if}\ \langle \textit{cond} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle\ [\texttt{else}\ \langle \textit{stmt} \rangle]^?\ \texttt{end} \\
& | & \texttt{for}\ \langle \textit{var} \rangle\ \texttt{in}\ \langle \textit{table} \rangle\ \texttt{where}\ \langle \textit{qcond} \rangle\ \texttt{do}\ \langle \textit{stmt} \rangle\ \texttt{end} \\
& | & \texttt{emit}\ \texttt{\{}\ [\langle \textit{field} \rangle\ \texttt{=}\ \langle \textit{var} \rangle]^*\ \texttt{\}}\ \texttt{in}\ \langle \textit{name} \rangle \\
\langle \textit{qcond} \rangle & ::= & \langle \textit{qcond} \rangle\ \texttt{and}\ \langle \textit{qcond} \rangle \\
& | & \langle \textit{var} \rangle\ \texttt{.}\ \langle \textit{field} \rangle\ \texttt{==}\ \langle \textit{var} \rangle \\
\langle \textit{cond} \rangle & ::= & \langle \textit{var} \rangle\ \texttt{==}\ \langle \textit{var} \rangle\ |\ \langle \textit{var} \rangle\ \texttt{!=}\ \langle \textit{var} \rangle
\end{array}
$$

Figure 5.1: Normalized Seam View Definitions

5.1.2 Multi-sets

First, let's consider the (abstract algebra) ring of multi-sets. The elements of the algebra are multi-sets. Each multi-set has a type-signature given by an unordered tuple of row-type variables. For instance, `{t : Tris, v0 : Verts, v1 : Verts}`. Then we say that a function which assigns a *count* integer for every assignment of row-values is a multi-set. For instance, Suppose we have a database with 2 triangles (`Tris` = {`t_A`, `t_B`}) and 4 vertices (`Verts` = {`v_0`, `v_1`, `v_2`, `v_3`}). Then there are $2 \times 4 \times 4$ possible assignments of row-values to the variables `t`, `v0`, `v1`. A multi-set assigns a count to each such joint row-value, with count 0 implied if no other count is specified. For instance, a multi-set R might be specified by $R(t = t_A, v0 = v_1, v1 = v_2) = 3$, $R(t = t_A, v0 = v_2, v1 = v_3) = 1$ and $R(t = t_B, v0 = v_2, v1 = v_1) = 1$. Although I will not make use of it, we can neatly tabulate multi-sets in the following way, making their connection to databases and tables explicitly visible.

`t`	`v0`	`v1`	$\mathbb{Z}$
t_A	v_1	v_2	3
t_A	v_2	v_3	1
t_B	v_2	v_1	1

Multi-sets are counting functions, $Tuple \to \mathbb{Z}$ (for appropriate interpolations of $Tuple$), which we can use to define the standard relational operators on them: union, (Cartesian) product, join, selection and projection.

projection. Consider a multi-set R with type-signature $\{a : A, b : B\}$. The projection of R onto $a : A$, written $\pi_{a:A}R$ is defined as a multi-set with type-signature $\{a : A\}$ defined by the equation

$$(\pi_{a:A}R)(a) = \sum_{b:B} R(a, b)$$

This operation may be extended to larger tuples with multiple variables retained/projected-onto and multiple variables summed/projected-out.

selection. Consider a multi-set R with type-signature $\{a : A, b : B\}$. And consider some logical predicate $P(a, b)$ that may ignore one or more of its arguments. Then selection by the predicate P is written $\sigma_P R$ and defined as another multi-set with identical type-signature to R defined by the equation

$$(\sigma_P R)(a, b) = \begin{cases} R(a, b), & P(a, b) \\ 0, & \text{otherwise} \end{cases}$$

This operation may be trivially extended to wider or narrower type-signatures.

(Cartesian) product. Consider two multi-sets R and S with disjoint type-signatures: $\{a : A\}$ and $\{b : B\}$. Then the product multi-set has signature $\{a : A, b : B\}$ and is written $R \times S$. It is defined by the product-of-counts of the components, given in the equation

$$(R \times S)(a, b) = R(a) \cdot S(b)$$

As with other operators, this one may be extended to different size type-signatures. However, we will mostly be interested in the case where one side of the product is a singleton type-signature, as this corresponds to the addition of one variable name.

join. Joins may expressed as simply a (Cartesian) product followed by a selection, using a predicate that spans the two sides of the product. Given predicate $P(a, b)$, the join by P of R and S (as given in the product definition) is written $R \bowtie_P S$, meaning $\sigma_P(R \times S)$.

union. Consider two multi-sets R and S with the same type-signature: $\{a : A, b : B\}$. The union under multi-set semantics differs crucially from the standard set semantics of relational algebra by tracking the number of duplicates encountered. This is the key definition allowing us to treat relational algebra as defining a ring proper, with additive inverses. We write union as $R \uplus S$ to emphasize this distinction. It is defined by the equational rule

$$(R \uplus S)(a, b) = R(a, b) + S(a, b)$$

The operator may be trivially extended to wider or narrower type-signatures.

negation and minus. By allowing negative counts, multi-sets gain the ability to be negated $(-R)(a, b) = -(R(a, b))$ and subtracted $R - S = R \uplus (-S)$.

Multi-sets form a ring with $\uplus$ as the addition operator and $\times$ as the product. Associativity and commutativity of both operators are inherited from addition and multiplication of integers, as is distributivity between the two operators. The constant zero-function (i.e. the empty multi-set) for a given type-signature functions as a zero in the algebra[2]. Crucially, we may have multi-sets with negative counts, and therefore

[2]The fact that we have multiple zeros, coupled with the fact that type-signatures force our operators to be partial functions on the collection of all multi-sets means that properly speaking

may define $-R$ for any relation R, such that $R \uplus -R = 0$; this property gives us additive inverses, the last major requirement for a ring. We may define a special multi-set on the empty type-signature as a kind of multiplicative unit. We will write $\{\}$ for this multiplicative unit.

As we will see shortly, these properties allow us to define a finite-difference operation that resembles an algebraically defined derivative on the ring.

5.1.3 Multi-set Combinators

The syntactic behavior of multi-set operators will be inconvenient to work with as an intermediate representation in the compiler. Consequently, we define multi-set combinators instead. Each of these combinators will take a multi-set as input and return a multi-set as output. Let's see how we may translate each operator into a combinator.

projection. $\pi[a : A](R) = \pi_{a:A} R$

selection. $\sigma[P](R) = \sigma_P R$

product. Let R be a relation with a signature not already using variable name a. Then, $\times[a : A](R) = \{a : A\} \times R$ where $\{a : A\}$ is a multi-set version of the set A with the specified type-signature and count 1 for each element of A. We could have defined product as a binary operation on combinators. However, this asymmetric version of product will be more useful for translating to and from code.

composition. Let F and G be multi-set combinators with the output signature of F the same as the input signature of G. Then $(F \circ G)(R) = G(F(R))$. This syntactic convention of "do the left, then the right" will be important later, even if it feels counter-intuitive.

this is not a ring. The issue is analogous to the difference between a group and groupoid. I am not sufficiently well versed to say exactly what this is. However, the idea of an "algebroid" seems to apply.

join. This is now defined directly as a composition of combinators. Let R and $a : A$ be as in the definition of product. Then $\bowtie [a : A|P](R) = (\times[a : A] \circ \sigma[P])(R)$.

union. Let F and G be multi-set combinators with the same input and output signatures. Then $(F \uplus G)(R) = F(R) \uplus G(R)$.

In order to incorporate effects, we will allow for special terminal effect combinators to be mixed in. These combinators may be interpreted as outputting a special "effect"-typed multi-set to which no further operations may be applied. Crucially, this slight abuse of notation will allow us to write down formal "sums" of effects, such as $\texttt{EFF1}[a] \uplus \texttt{EFF2}[a]$.

5.2 Translating Between Views and Combinators

This translation uses two basic ideas. First, execution contexts may be abstracted by multi-sets. Second, the semantics of a program statement is just a multi-set combinator.

Imagine a Seam program in the middle of execution. At a given point in the program, a set of variables have been bound to values of given types, defining an *execution context*. Every time the execution returns to this point in the program, the same set of variable names (and types) will be bound—potentially to different values. In other words, the set of executions that reach a program point may be abstracted as a multi-set of execution contexts.

Now consider two such program points separated by a single statement. For instance, consider the execution before and after a `for`-loop.

```
viewdef( z : E )
  let y = z.tl in
  ------------------------------
  for x in E where x.hd == y do
  ------------------------------
    ...
  end
end
```

$$\{\mathtt{y} : V, \mathtt{z} : E \mid \mathtt{y} = \mathtt{z.tl}\}$$

$$\{\mathtt{x} : E, \mathtt{y} : V, \mathtt{z} : E \mid \mathtt{y} = \mathtt{z.tl} \wedge \mathtt{x.hd} = \mathtt{y}\}$$

The multiset of execution contexts before and after the for loop are related. For each context that reaches the loop, we extend it with a binding for `x : E`, provided that $\texttt{x.hd} = \texttt{y}$. This may produce no inner loop bodies if there are no `x : E` with `y` pointed to. It may also produce a very large number of execution contexts when many `x` satisfy this condition. Whichever way, this is the correct multi-set to express the set of executions reaching this program point.

Now, consider the statement itself (the for loop) independent of the specific multi-set expressions that precede and succeed it. Its meaning can be precisely captured by the combinator $\times[\texttt{x} : \texttt{E}] \circ \sigma[\texttt{x.hd} = \texttt{y}]$, or even more succinctly

$$\bowtie [\texttt{x} : \texttt{E} \,|\, \texttt{x.hd} = \texttt{y}]$$

Each statement in our normalized Seam view program grammar may be given a combinator translation. (Figure 5.2)

$\llbracket$ `viewdef(` x `:` X `)` s `end` $\rrbracket$	$\times[x : X] \circ \llbracket s \rrbracket$
$\llbracket s_1$ `;` $s_2 \rrbracket$	$\llbracket s_1 \rrbracket \uplus \llbracket s_2 \rrbracket$
$\llbracket$`let` x `:` X `=` y `in` $s\rrbracket$	$\bowtie [x : X \,\vert\, x = y] \circ \llbracket s \rrbracket$
$\llbracket$`let` x `:` X `=` y`.`f `in` $s\rrbracket$	$\bowtie [x : X \,\vert\, x = y\,.\,f] \circ \llbracket s \rrbracket$
$\llbracket$`if` x `==` y `then` s `end`$\rrbracket$	$\sigma[x = y] \circ \llbracket s \rrbracket$
$\llbracket$`if` x `!=` y `then` s `end`$\rrbracket$	$\sigma[x \neq y] \circ \llbracket s \rrbracket$
$\llbracket$`for` x `in` X `where` x`.`f `==` y `and` ... `do` s `end`$\rrbracket$	$\bowtie [x : X \,\vert\, x\,.\,f = y \wedge ...] \circ \llbracket s \rrbracket$
$\llbracket$`emit {` f_1 `=` x_1`, ... } in` $Y\rrbracket$	$\pi[x_1, ...] \circ \texttt{EMIT}(Y, \{f_1 = x_1, ...\})$

Figure 5.2: Seam translation from view programs to multi-set combinators

This translation preserves the tree-structure of the program's AST. Nesting is translated to $\circ$ composition, which reads cleanly in left-to-right fashion. Branching due to concurrent effects (i.e. `;`) translates to $\uplus$ which sticks together branches terminating in `EMIT`.

Thanks to the algebraic properties of combinators, we can now consider rather drastic rewrites of programs. All joins can be broken down into their constituent Cartesian product and selection combinators, allowing selection combinators to float freely up and down the tree—subject only to name-scoping constraints. Likewise, the products may be distributed inside of sums or factored out of them. For instance, the earlier `viewdef` translates to the following combinator:

$$\times[z : E] \circ \times[y : V] \circ \sigma[y = z.\texttt{tl}] \circ \times[x : E] \circ \sigma[x.\texttt{hd} = y] \circ \cdots$$

If we re-arrange this combinator's order of variable introduction, we get an equivalent combinator

$$\times[y : V] \circ \times[z : E] \circ \sigma[z.\texttt{tl} = y] \circ \times[x : E] \circ \sigma[y = x.\texttt{hd}] \circ \cdots$$

which translates back out into code as

```
viewdef( y : V )
  for z in E where z.tl == y do
    for x in E where x.hd == y do
      ...
    end
  end
end
```

It may come as some surprise that a let-expression has been transformed into a for-loop. Yet it computes.

5.3 Updates via Derivatives of Combinators

Having settled on the multi-set interpretation of views, we may alternatively think of `EMIT` effects as modifying a multi-set representation by *incrementing* the count associated to a particular set of keys. Let `INCR(tbl,rec,count)` be used to refer to this kind of effect.

The fundamental idea of treating an incremental view update as a *derivative* is to express the change in a view as a function of some change to the underlying data. Let R_1 be the view multi-set after this change and R_0 the view multi-set before the change. In terms of multi-sets, we want to compute ΔR such that $R_1 = R_0 \uplus \Delta R$.

ΔR contains exactly the `INCR` operations we must perform to update the view in response to the underlying change.

More precisely, what we have prior to loading any data is not really any of these multi-sets, but a combinator for computing them from base data. That is, $R_0 = R(\{\})$ when R is supplied with base tables and fields via a collection of implicit arguments M. $R_0 = R[M_0](\{\})$ and $R_1 = R[M_1](\{\})$. Suppose then that M_0 and M_1 are related by a primitive change such as an individual `new`, `delete`, or `update`, which we could (waving our hands a bit) notate as $\Delta M = M_1 - M_0$. Then, what we really want to find is a combinator $\Delta R[M, \Delta M]$ so that $\Delta R[M, \Delta M](\{\}) = R_1 - R_0$. Ideally such a combinator (once translated back to executable code) will be much cheaper to compute than simply recomputing the whole view from scratch.

At this point a very non-obvious but crucial digression is necessary. View updates may be performed *before* or *after* the update to the underlying data. However, the computation will differ in subtly important ways depending on which formulation we use. In the case of `delete` operations it is much simpler to perform the view update before hand because the element we want to delete still exists in the underlying data with a stable identifier. In the case of `new` operations the situation is reversed. Unless we have already executed the `new` operation we have no clear identifier for the element to be added. (The `update` situation is more ambivalent.) To clarify this whole situation, I will assume that the view update for each `new` operation will be processed immediately after the `new` and otherwise right before the effect.

5.3.1 deletions

Consider `delete dx` with `dx:X`. After the effect the updated set will be $X_1 = X - \{dx\}$. Given a combinator R for the view, parameterized by the set X, we have the equation $R[X_1] = R[X_0] \uplus \Delta_{-\mathtt{dx}} R[X_0]$ as a definition of the derivative.

Before we figure out how to arrive at the isolated Δ expression, we observe that we can write down $R[X_1]$ without reference to X_1. We simply replace every instance of $\times[x : X_1]$ with the expanded $\times[x : X_0] - \bowtie [x : X_0 \,|\, x = dx]$. Alternately, we could collapse these terms into the joint $\bowtie [x : X_0 \,|\, x \neq dx]$. Both forms will be useful.

Obtaining the Δ expression then, involves recursively separating out a copy of $R[X_0]$ from the remaining terms which express the update proper. Doing so will be trivial except in the case of the product.

Consider a combinator of the form $R[X] = \times[x : X] \circ S[X]$. Using the preceding equations in a recursive form, we can re-express $R[X_1]$:

$$\begin{aligned}
R[X_1] &= \times[x : X_1] \circ S[x : X_1] \\
&= \times[x : X_1] \circ (S[X_0] \uplus \Delta_{\texttt{-dx}} S[X_0]) \\
&= \times[x : X_1] \circ S[X_0] \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{-dx}} S[X_0] \\
&= (\times[x : X_0] - \bowtie [x : X_0 \,|\, x = dx]) \circ S[X_0] \\
&\quad \uplus \bowtie [x : X_0 \,|\, x \neq dx] \circ \Delta_{\texttt{-dx}} S[X_0] \\
&= \times[x : X_0] \circ S[X_0] \\
&\quad - \bowtie [x : X_0 \,|\, x = dx]) \circ S[X_0] \\
&\quad \uplus \bowtie [x : X_0 \,|\, x \neq dx] \circ \Delta_{\texttt{-dx}} S[X_0]
\end{aligned}$$

The first term of this expression is of course just $R[X_0]$, meaning the latter two terms are $\Delta_{\texttt{-dx}} R[X_0]$.

To perform this separation recursively, we write a transformation which takes a combinator as input and returns a combinator representing the offset. This transformation is a "derivative."[3] (Figure 5.3)

In order to remove negations from the combinator, they may be pushed down until we arrive at $-\texttt{INCR}(Y, \{f_1 = x_1, ...\}, c)$, which is simply equivalent to $\texttt{INCR}(Y, \{f_1 = x_1, ...\}, -c)$, i.e. we need simply flip the sign of the increment count.

As written, this transformation will cause at most a quadratic increase in code size of the combinator. We can also read out the logic of the branching structure

[3]Note that the derivative of a product produces something quite different than the Leibniz rule. We must account for a "higher-order" effect by adding the $x \neq dx$ condition on the further derivative branch.

$$\begin{array}{rcl}
\Delta_{\texttt{-dx}}[\![\sigma[p] \circ S]\!] & \to & \sigma[p] \circ \Delta_{\texttt{-dx}}[\![S]\!] \\
\Delta_{\texttt{-dx}}[\![\pi[x, \ldots] \circ S]\!] & \to & \pi[x, \ldots] \circ \Delta_{\texttt{-dx}}[\![S]\!] \\
\Delta_{\texttt{-dx}}[\![\times[y : Y] \circ S]\!] & \to & \times[y : Y] \circ \Delta_{\texttt{-dx}}[\![S]\!] \\
\Delta_{\texttt{-dx}}[\![\times[x : X] \circ S]\!] & \to & \left(\begin{array}{l} - \bowtie [x : X_0 \,|\, x = dx] \circ S \\ \uplus \;\bowtie [x : X_0 \,|\, x \neq dx] \circ \Delta_{\texttt{-dx}}[\![S]\!] \end{array} \right) \\
\Delta_{\texttt{-dx}}[\![R \uplus S]\!] & \to & \Delta_{\texttt{-dx}}[\![R]\!] \uplus \Delta_{\texttt{-dx}}[\![S]\!] \\
\Delta_{\texttt{-dx}}[\![\texttt{INCR}(Y, \{\ldots\}, c)]\!] & \to & \texttt{NONE}
\end{array}$$

Figure 5.3: `delete` derivative of a combinator

induced. At every variable introduction (product combinator) which could possibly be the deleted variable, either it is the first occurrence of the deleted variable, or it isn't the deleted variable. At the very end, if we have repeatedly decided that none of the variables are the deleted variables, then there is no possible effect.

The resulting combinator leaves the variable dx free. As we will see shortly, this fact will allow us to rewrite the combinator as a function of dx—ultimately a localized function.

5.3.2 new/insertions

For an individual new-effect `new dx : X`, the updated set equation is $X_1 = X \uplus \{dx\}$. Given a combinator R for the view, parameterized by the set X, then the derivative is similarly defined by $R[X_1] = R[X_0] \uplus \Delta_{\texttt{+dx}} R[X_1]$. Note crucially that for new-effects, we express the derivative as a function of the updated set, and not the original set.

Given $\times[x : X_1]$, we can rewrite it as $\times[x : X_0] \uplus \bowtie [x : X_1 \,|\, x = dx]$, which has the desired form. However, we will see that the derivation will be a bit more complicated.

Consider a combinator of the form $R[X] = \times[x : X] \circ S[X]$.

$$\begin{aligned}
R[X_1] &= \times[x : X_1] \circ S[x : X_1] \\
&= \times[x : X_1] \circ (S[X_0] \uplus \Delta_{\texttt{+dx}} S[X_1]) \\
&= \times[x : X_1] \circ S[X_0] \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= (\times[x : X_0] \uplus \bowtie [x : X_1 \,|\, x = dx]) \circ S[X_0] \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= \times[x : X_0] \circ S[X_0] \\
&\quad \uplus \bowtie [x : X_1 \,|\, x = dx] \circ S[X_0] \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= R[X_0] \\
&\quad \uplus \bowtie [x : X_1 \,|\, x = dx] \circ (S[X_1] - \Delta_{\texttt{+dx}} S[X_1]) \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= R[X_0] \\
&\quad \uplus \bowtie [x : X_1 \,|\, x = dx] \circ S[X_1] \\
&\quad - \bowtie [x : X_1 \,|\, x = dx] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&\quad \uplus \times [x : X_1] \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= R[X_0] \\
&\quad \uplus \bowtie [x : X_1 \,|\, x = dx] \circ S[X_1] \\
&\quad \uplus (\times[x : X_1] - \bowtie [x : X_1 \,|\, x = dx]) \circ \Delta_{\texttt{+dx}} S[X_1] \\
&= R[X_0] \\
&\quad \uplus \bowtie [x : X_1 \,|\, x = dx] \circ S[X_1] \\
&\quad \uplus (\bowtie [x : X_1 \,|\, x \neq dx]) \circ \Delta_{\texttt{+dx}} S[X_1]
\end{aligned}$$

With a similar product rule in hand, we may similarly express the derivative with respect to a newly created element. (Figure 5.4)

$$
\begin{array}{rcl}
\Delta_{\texttt{+dx}}[\![\sigma[p] \circ S]\!] & \to & \sigma[p] \circ \Delta_{\texttt{+dx}}[\![S]\!] \\
\Delta_{\texttt{+dx}}[\![\pi[x,\ldots] \circ S]\!] & \to & \pi[x,\ldots] \circ \Delta_{\texttt{+dx}}[\![S]\!] \\
\Delta_{\texttt{+dx}}[\![\times[y:Y] \circ S]\!] & \to & \times[y:Y] \circ \Delta_{\texttt{+dx}}[\![S]\!] \\
\Delta_{\texttt{+dx}}[\![\times[x:X] \circ S]\!] & \to & \left(\begin{array}{rl} & \bowtie [x : X_1 \,|\, x = dx] \circ S \\ \uplus & \bowtie [x : X_1 \,|\, x \neq dx] \circ \Delta_{\texttt{+dx}}[\![S]\!] \end{array} \right) \\
\Delta_{\texttt{+dx}}[\![R \uplus S]\!] & \to & \Delta_{\texttt{+dx}}[\![R]\!] \uplus \Delta_{\texttt{+dx}}[\![S]\!] \\
\Delta_{\texttt{+dx}}[\![\texttt{INCR}(Y, \{...\}, c)]\!] & \to & \texttt{NONE}
\end{array}
$$

Figure 5.4: `new` derivative of a combinator

This transformation has a nearly identical form to deletions, with only the sign changed—even though the precise meaning and derivation differ substantially.

5.3.3 updates

Updates induce changes to some selection combinators, but not to the Cartesian product. Consider the `update` `dx.f = dy1` effect for field `X.f : Y`. We will let dy_0 represent the value at $dx\,.\,f$ before the update and dy_1 represent the value afterwards. This will induce a change to the atomic predicate $x\,.\,f = y$, replacing it with

$$((x\,.\,f = y) \wedge \neg(x = dx \wedge y = dy_0)) \vee (x = dx \wedge y = dy_1)$$

As a truth table over variables x and y, we can think of this as making two spot updates, flipping the $x = dx/y = dy_0$ entry from true to false and the $x = dx/y = dy_1$ entry from false to true. This implies certain logical dominance and independence relationships between the three components of the predicate. In turn, that means that we can safely drop certain terms from the inclusion-exclusion expansion of the predicate:

$$\begin{aligned}
P_0 &= x\,.\,f = y \\
P^- &= x = dx \wedge y = dy_0 \\
P^+ &= x = dx \wedge y = dy_1 \\
\sigma[P_1] &= \sigma[(P_0 \wedge \neg P^-) \vee P^+] \\
&= \sigma[P_0] - \sigma[P^-] \uplus \sigma[P^+] \\
&= \sigma[P_0] \uplus (\sigma[P^+] - \sigma[P^-])
\end{aligned}$$

Using the same derivation technique as before for the recursive rule, we consider a multi-set combinator $R[f] = \sigma[x\,.\,f = y] \circ S[f]$ and attempt to re-express $R[f_1]$ in terms of $R[f_0] \uplus \Delta_{\mathtt{f}} R[f_0]$.

$$\begin{aligned}
R[f_1] &= \sigma[P_1] \circ S[f_1] \\
&= \sigma[P_1] \circ (S[X_0] \uplus \Delta_{\mathtt{f}} S[f_0]) \\
&= \sigma[P_1] \circ S[f_0] \\
&\quad \uplus\ \sigma[P_1] \circ \Delta_{\mathtt{f}} S[f_0] \\
&= (\sigma[P_0] \uplus (\sigma[P^+] - \sigma[P^-])) \circ S[f_0] \\
&\quad \uplus\ \sigma[P_1] \circ \Delta_{\mathtt{f}} S[f_0] \\
&= \sigma[P_0] \circ S[f_0] \\
&\quad \uplus (\sigma[P^+] - \sigma[P^-]) \circ S[f_0] \\
&\quad \uplus\ \sigma[P_1] \circ \Delta_{\mathtt{f}} S[f_0]
\end{aligned}$$

In order to rule out exponential increases in code size we would need to keep combinator expressions like $(\sigma[P^+] - \sigma[P^-])$ together rather than split them apart (and duplicate $S[f_0]$). I chose not to tackle this issue when implementing the prototype. I simply expanded sums and pushed negations down to the leaves. However, fixing this issue exposes interesting questions about the interpretation of combinators as code.

This transformation can be packaged into a full, recursive derivative transformation as follows. (Figure 5.5)

$$
\begin{array}{rcl}
\Delta_{\mathtt{f}}[\![\sigma[x = y] \circ S]\!] & \rightarrow & \sigma[x = y] \circ \Delta_{\mathtt{f}}[\![S]\!] \\
\Delta_{\mathtt{f}}[\![\sigma[x\,.\,g = y] \circ S]\!] & \rightarrow & \sigma[x\,.\,g = y] \circ \Delta_{\mathtt{f}}[\![S]\!] \\
\Delta_{\mathtt{f}}[\![\sigma[x\,.\,f = y] \circ S]\!] & \rightarrow & \left(\begin{array}{r} (\sigma[P^{+}] - \sigma[P^{-}]) \circ S[f_0] \\ \uplus\ \sigma[P_1] \circ \Delta_{\mathtt{f}} S[f_0] \hspace{5em} \end{array} \right) \\
 & & \begin{array}{lrcl} & P_1 & = & ((x\,.\,f = y \wedge \neg P^{-}) \vee P^{+}) \\ \text{where} & P^{-} & = & (x = dx \wedge y = dy_0) \\ & P^{+} & = & (x = dx \wedge y = dy_1) \end{array} \\
\Delta_{\mathtt{f}}[\![\pi[x, \ldots] \circ S]\!] & \rightarrow & \pi[x, \ldots] \circ \Delta_{\mathtt{f}}[\![S]\!] \\
\Delta_{\mathtt{f}}[\![\times[x : X] \circ S]\!] & \rightarrow & \times[x : X] \circ \Delta_{\mathtt{f}}[\![S]\!] \\
\Delta_{\mathtt{f}}[\![R \uplus S]\!] & \rightarrow & \Delta_{\mathtt{f}}[\![R]\!] \uplus \Delta_{\mathtt{f}}[\![S]\!] \\
\Delta_{\mathtt{f}}[\![\mathtt{INCR}(Y, \{...\}, c)]\!] & \rightarrow & \mathtt{NONE}
\end{array}
$$

Figure 5.5: `update` derivative of a combinator

5.4 Re-rooting the derivative combinators

At this point, we have all the tools to create multi-set combinators that correctly express the multi-set of increments that need to be performed to maintain a given view. However, we want to transform these combinators back into code. Specifically, we want this code to be a function parameterized by the free variables introduced by the derivative. The resulting function can then be invoked as the effect log is played back to incrementally maintain the view.

At the same time we want to try to maintain the *locality* property of the view code that we constrained the programmer by. Each new variable ought to be introduced via a join to already defined variables. Doing this will prevent any unbounded loops. So long as the data structure graph has a non-trivial diameter[4] the view updates, like the view expression and the operations themselves will touch only a small fraction of the entire data-structure.

Note that this locality is possible because the original constraints on the view program ensure that all variables are defined by connection to other variables. (Other than the intial argument variable, which simply gives a starting point.)

In order to build intuition for the desired transformation, first consider the case of a perfectly linear combinator expression R without any $\uplus$ branches in it. Let dx be the variable we want to re-root the combinator at. Then, somewhere in R there is a first selection $\sigma[p]$ referencing dx, of the form $\sigma[dx = z]$, $\sigma[dx\,.\,f = z]$, or $\sigma[dx = z\,.\,f]$. There is also a $\times[z : Z]$ occuring somewhere before that. Then, if R' is the combinator R with those selection and product combinators snipped out, $R = \bowtie[z : Z \,|\, p] \circ R'$. In other words, we can simply percolate out the first variable joined to dx—or to any other defined variable for that matter.

The last observation allows us to bootstrap the transformation recursively from the free-variables. If we start a downward search through a combinator chain for something connected to a defined variable, then we can recursively add that new variable to the set of defined variables and repeat the search on the remaining part of the combinator.

This idea works for linear combinator chains, but not for $\uplus$. As a naive but sufficient strategy we simply break apart $\uplus$ branches as we encounter them. Often this will increase code size yet again. I simply did not have reason to explore more sophisticated code optimization that is sensitive to this concern. However, note that the duplication of breaking apart a tree cannot increase code size by more than a quadratic amount as a worst-case bound.

Let $\top\!\!\!\!\!\!\;\;\overline{}\; [\![R \,\|\, N \,\|\, S \,\|\, V]\!]$ denote the re-rooting transformation. R is the combinator being re-rooted; N is a set of variables introduced by product combinators somewhere in the stack of selectors; S is a stack of selector combinators for holding

[4]diameter meaning the maximum number of hops between any two elements. Of course, just connecting everything through a super-node makes "locality" less meaningful.

un-transformed parts of the expression; and V is a set of variables defined outside this working context. In order to re-root a combinator R at variable dx we would invoke $⏚\llbracket R \,\|\, \emptyset \,\|\, id \,\|\, \{dx\} \rrbracket$. If we have more than one free variable (as in the case of an `update`) we may throw all free variables into the initial defined variable set.

For a given primitive predicate p let $Var(p)$ be the set of variables referenced in the predicate. Note that $\#Var(p) = 2$, so $V\#p = \#(Var(p) \cap V)$ is either 0, 1 or 2. We will apply different rules depending on this tri-partite division.

$$
\begin{array}{rcl}
⏚\llbracket R_1 \uplus R_2 \,\|\, N \,\|\, S \,\|\, V \rrbracket & \rightarrow & ⏚\llbracket R_1 \,\|\, N \,\|\, S \,\|\, V \rrbracket \uplus ⏚\llbracket R_2 \,\|\, N \,\|\, S \,\|\, V \rrbracket \\
⏚\llbracket \times[x : X] \circ R \,\|\, N \,\|\, S \,\|\, V \rrbracket & \rightarrow & ⏚\llbracket R \,\|\, N \cup \{x : X\} \,\|\, S \,\|\, V \rrbracket \\
⏚\llbracket \sigma[p] \circ R \,\|\, N \,\|\, S \,\|\, V \rrbracket & \rightarrow & \sigma[p]\circ\ ⏚\llbracket R \,\|\, N \,\|\, S \,\|\, V \rrbracket \\
 & & \text{where } V\#p = 2 \\
⏚\llbracket \sigma[p] \circ R \,\|\, N \,\|\, S \,\|\, V \rrbracket & \rightarrow & ⏚\llbracket R \,\|\, N \,\|\, S \circ \sigma[p] \,\|\, V \rrbracket \\
 & & \text{where } V\#p = 0 \\
⏚\llbracket \sigma[p] \circ R \,\|\, N \,\|\, S \,\|\, V \rrbracket & \rightarrow & \bowtie [x : X \,|\, p] \circ \\
 & & ⏚\llbracket S \circ R \,\|\, N - \{x : X\} \,\|\, id \,\|\, V \cup \{x\} \rrbracket \\
 & & \text{where } V\#p = 1 \text{ and } Var(p) - V = \{x\} \\
⏚\llbracket \pi[...] \circ R \,\|\, \emptyset \,\|\, id \,\|\, V \rrbracket & \rightarrow & \pi[...] \circ R
\end{array}
$$

Figure 5.6: re-rooting transformation for multi-set combinators

Note that the last rule here relies on the specific structure of combinators that we generate from the prior transformations. Projection *only* occurs at the leaves of the combinator tree expression, right before an effect, and right before every effect.

The last rule also makes a true claim which I do not prove. By the time the transformation reaches a leaf, all of the intermediate stacks have been cleared—all variables have been defined/re-ordered successfully.

5.5 Translating Back to code

The translation from a transformed combinator back into code simply inverts the translation already given in Figure 5.2. Before doing that, a very minor optimization pass attempts to collect as many selection predicates as possible that could be merged into the defining predicate for each variable (without breaking any more $\uplus$ structures). For a product $\times[x : X]$ if any of these selection predicates are of the form $\sigma[x = e]$ for any expression e, then we merge that selection alone as $\bowtie [x : X \mid x = e]$, which will be code-generated back into a `let`-binding. Otherwise, as many selections of the form $\sigma[x \,.\, f = y]$ as possible are gathered together, to give the generated query-loop the most possible material to work with later on.

Chapter 6

Prototype Implementations

I constructed[1] three prototype compilers for this thesis. All three languages were prototyped as Lua-embedded domain specific languages using Terra[DHA$^+$13]. There are two complimentary ways to view these prototypes. First, each prototype is its own software artifact with important idiosyncracies of its design; this view is important so that I can accurately describe the basis for the experimental evaluations in the next chapter. The second way of viewing the prototypes is as continued iterations approximating an unconstructed joint language. In this second view the code generation of a data model representation is of particular interest. In aggregate, what is required and required by different DSLs of a common data model?

6.1 Common Compiler Features

As mentioned, all compilers were implemented in Lua, making use of the Terra[DHA$^+$13] metaprogramming framework for both code-generation and Lua parser hijacking.

The front-end of all the compilers mimiced design features of the Terra language itself, being organized into a series of relatively standard passes: (1) parsing, (2)

[1]I inherited the Ebb compiler, which was heavily the work of Zach DeVito, Crystal Lemire, and Chinmayee Shah as well. The Seam compiler (especially the SMT translation) was built along with Manolis Papadakis.

specialization, (3) type-checking, and (4) effect-checking[2]. Aside from parsing, which happens when the Lua file itself is parsed, all of these passes occur at *definition-time*[3] meaning the point in the execution of the enclosing Lua script when the thread of control passes over the lexical occurrence of Ebb, Seam, or Gong code. This definition returns a Lua object which encapsulates the processed, typed, and effect-analyzed AST. The object is bound to a Lua variable for later use.

The front-end processes and reduces out syntax sugar, macros and other convenience features. In particular, the addition of the specialization pass (mimicing Terra) makes it easier to resolve names and capture values from the enclosing Lua environment of the definition. The pass is defined by this responsibility to resolve and remove all free variable references in the code. Like with Terra, macros are resolved at type-checking time so that all arguments to a macro may be typed prior to invocation.

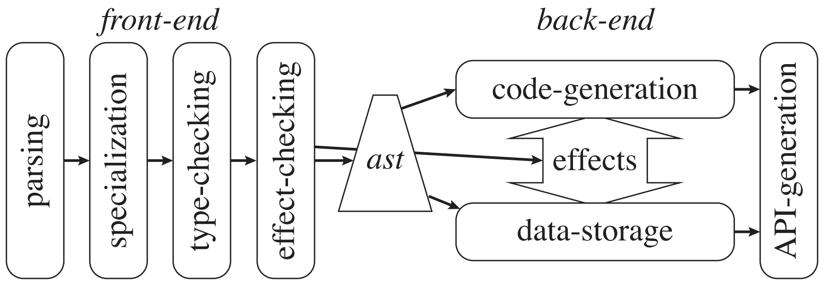


Figure 6.1: Compiler Architecture. Effect-analysis acts as a stronger form of type-checking and governs the abstraction between data-structure generation and code-generation in the back-end.

The backend of the compilers varies much more wildly. For instance, Ebb maintains a dynamic runtime that interleaves execution with the Lua script, whereas Seam and Gong generate a `C`-library for later use.

However, all of the back-ends are structured around two interacting components: the code-generator and the data-store-generator. (Data-store runtime for Ebb) These two components are largely intermediated by the effects of Chapter 4. The effects

[2]In Ebb this was called phase-checking in keeping with Liszt terminology. The meaning is identical.

[3]In Seam, the more sophisticated effect-checking is post-poned till compilation.

essentially specify the signature of calls from the code-generator into the data storage component. The data storage returns a snippet of code that implements the desired effect while hiding the details of data-representation from the code generator module.

6.2 Ebb

In Ebb, schema construction was exposed to the Lua script as a series of API functions to register new relations, add fields, &c. This meant that the data structure representation and code generation were required to be *extensional*. The program could always add new tables and fields to the schema, or new functions to compute on the data. Where choices had to be fixed (e.g. `group_by` or `periodic`) the implementation had to have ways of committing those decisions and raising errors on attempts to change those settings after functions had been defined on the tables.

6.2.1 Data Layout and Management

An Ebb data store may be thought of as Lua metadata (for the schema) backed by explicitly memory managed arrays for the data. Each `table` (or `grid`) is given a Lua object, with Lua objects for each `field` defined on it. The field objects additionally keep pointers to arrays of backing data. Each `global` is given a Lua object that keeps a pointer to a chunk of data in which the global is stored. On the GPU, the pointed-to data is mirrored.

Keys. One fundamental implementation question is simply "how are references to rows of a table represented?" Answering this question will constrain the layout decision from there. I used simple unsigned integers to represent indices in Ebb. However, both `grid`s and the dynamic runtime exposed opportunities for optimization of this representation. The Ebb prototype requires the simulation-programmer to supply the fixed, static size of each table (or grid dimensions) when they declare it. With this information, the Ebb runtime chooses whether to use `uint8`, `uint16`, `uint32`, or `uint32` values to encode keys. For grids, the compiler packs these into a struct, so that for instance a $128 \times 128 \times 1024$ grid can be packed into a `{ _0:uint8, _1:uint8, _2:uint16 }` struct, which neatly fits into 4-bytes.

For grids there is an additional question of how these multi-dimensional key values get linearized. I used a simple lexicographic order between the coordinates, without any tiling or blocking.

Column Storage. This layout corresponds to column-storage in databases, also known as struct-of-arrays in GPU programming. This approach contrasts to row-storage (or array-of-structs). For instance, given a table with two fields `X.f : int32` and `X.g : double`, the row-storage pattern determines the layout of a row (4 bytes for `f`, followed by 4 dead bytes, followed by 8 bytes for `g`) and then repeats this pattern n times for a table with n rows. The column-store instead lays out a contiguous region of $4 \cdot n$ bytes to store `f` as n `int32` values and separately a contiguous region ($8 \cdot n$ bytes) for `g`.

Column storage has two important benefits in Ebb's design. First, "struct-of-arrays" is recognized as good practice on GPUs because it allows for coallesced data reads and writes to or from a field. The 32 different threads in a CUDA warp will often all hit the same read instruction. Pulling a contiguous $4 \cdot 32$ byte line out of cache or DRAM will induce less wasted memory traffic than if the read values are strided. This only becomes more true the more fields are defined on a given table.

Second, column storage allows the data representation to become extensional with respect to adding fields to a table after computations have already been performed. Unlike row-storage, column storage allows for the different columns to be separately allocated by a memory manager.

CPU/GPU data transfer. The Lua metadata objects must keep track of whether the CPU, GPU, both or neither backing memories are valid. When data is written either on the CPU or GPU, the alternate memory must be invalidated. Given these tags, a simple lazy data movement policy is implemented. If data is needed for a GPU or CPU computation but that copy of the data is not valid, then the data is copied over. Generally I did not try to analyze *write-over* behavior inside of Ebb functions as distinct from *read-write* access. This certainly leads to some level of superfluous data movement. However, data movement was already sufficiently complicated that I didn't want to complicate it any further.

Subsets. In addition to fields, tables may define subsets. Subsets are provided with two possible representations: masked or indexed. The implementation chooses between these two representations based on the size of the subset. If the subset consists of less than $\frac{1}{10}$ of the rows of the original table, then it is indexed. Otherwise it is masked.

Masked subsets are represented by additional hidden fields on the table of type `bool`, encoded somewhat wastefully as 8-bit values. Whenever a function is launched on the subset instead of the full table, a different verison of the function is compiled, with an "if this row is not masked out" guard around the body.

Indexed subsets are represented by a list of keys for the row in question, so that there is one key for each row in the subset. Because subsets were primarily used to handle special boundary region processing, most subsets tended to be either almost the entire table (e.g. the interior of a grid) or a fraction of the table much smaller than $\frac{1}{10}$ (e.g. the boundary of a grid). On the GPU, using a mask-based encoding would ensure that most 32-wide warps would have either 0 or 2 active lanes, yielding roughly $\frac{1}{16}$ occupancy before considering data or control divergence between warp-lanes.

Indices. In Ebb, a table `X` with a `group_by X.f` option set requires extra data to be stored in order to facilitate fast execution of `query`-loops. These bits of data are tricky to associate to an individual table. Viewed from one point of view the index of `X` by `f` ought to belong to `X`. But there is an equally good claim that the index ought to belong to `Y` when `X.f : Y`: namely that the index consists of an array of size $n + 1$ when `Y` is a table of size n. To avoid the predicament, it is better to think of indices as edge objects connecting the table objects in a graph.

The contents of the array representing a given index are a compressed and inverted representation of the `f` field. For row j of `Y`, Let I_j and I_{j+1} be the j^{th} and $j + 1^{\text{th}}$ entries of the index array. Then, the rows $I_j \leq i < I_{j+1}$ of `X` all have j as the value

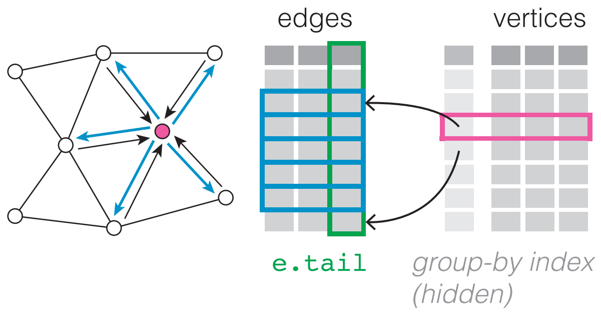


Figure 6.2: When the (directed) edges of a triangle mesh are grouped by their `tail` field, which contains vertex keys, the runtime (1) sorts the edges relation/table by the `tail` field/column and (2) constructs a hidden index parallel to the vertices relation/table. This index allows us to efficiently execute query-loops by simply looking up the loop iteration bounds for each vertex.

of `f`. A query-loop simply looks up these two subsequent index values and loops over the specified bounds.

This storage strategy coincides identically with the CSR (Compressed-Sparse-Row) sparse matrix storage format when a table like `Edges` is grouped by one of its endpoints.

6.2.2 Compilation & Code Generation

Code generation is implemented via Terra metaprogramming. For the most part this is extremely straightforward. Where it isn't, the issues have to do with complications in orchestrating meta-programming or working with the underlying LLVM interfaces correctly. To compile for GPU, I used the PTX backend of the LLVM compiler suite, exposed by Terra. CPU code was compiled for x86. By default CUDA kernels are launched with 64 threads per-block without tiling.

One interesting complication of Ebb is the ability to call a given `function` in different contexts. For instance, a function may be launched on the CPU or the GPU; it may be launched on the entire table or on a particular subset. These different variations of the function often require it to be recompiled in different ways. In order to manage this, code-generation was wrapped by a memoized function, keyed on each of these compilation-sensitive parameters. Because Ebb lazily JIT compiles code, this approach concisely allowed for the potential compilation of a large number of variants (without ever having to generate them all) without spurious recompilation.

Reductions Reads and writes can be trivially translated into load and store statements, but special care is necessary to handle reduction statements in Ebb when code is run in parallel on the GPU. Reductions fall into two categories: (1) reductions to an element of a *field*, and (2) reduction to a single *global* value. In the case of field reductions, relatively few threads will write to a memory location, so I compile these to PTX atomic reduction intrinsics. If no special PTX instruction is available, I use the compare-and-swap instruction in a loop to emulate the reduction.

In the second case of *global* reductions I used a variant of the two-pass reduction described in [Wil13]. The algorithm was modified in two ways so that the first-pass kernel can be fused into the original Ebb kernel code: (1) Rather than read the data to be reduced from global memory, reduction operations inside the kernel body directly reduce values into a per-block region of shared-memory. (2) Code is appended to the end of the kernel to reduce this shared memory array and write the result to global memory. The second kernel is then immediately launched to reduce the per-block values into the global value using the reference algorithm identically. With improvements in the CUDA atomics, this approach has become unnecessary.

6.3 Seam

Seam switched from Ebb's runtime model to generating libraries from special compilation calls. This switch simplified compiler implementation, since the compiler-programmer is now allowed to assume they have all schema and code that will be needed without possibility of extension.

For Seam, this switch was necessary. The nature of re-meshing (the problem Seam was built to explore) is that adding additional data, `view`s, or `operation`s interacts non-trivially with the existing code and schema. It is precisely how Seam checks all of these interactions, and derives potentially quadratically many view-updates and indices which makes the language valuable. It addresses the problem of safely extending code that is in no way trivially extensible.

In the prototype Seam compiler, I did not attempt to provide a GPU back-end. While such a back-end is possible (Seam would allow the compiler programmer to derive locking code according to whatever discipline is chosen) the CPU-only implementation already poses many interesting challenges and choices.

Among these are: (1) data layout and representation, given the high degree of mutability; (2) orchestration of log-playback, and (3) view-maintenance.

6.3.1 Data Layout & Management

In Seam the entire data storage is lowered into `C`-style data structures. The main such data structure (the `Store`) functions as a meta-data container for managing all the column arrays. While the column-storage strategy is preserved, the need to make the row-set of each table dynamic, and the need to support multiple indices forces any compiler-programmer to make different decisions than in Ebb.

Keys. In Seam all keys are stored as `uint32` values. The maximum value ($2^{32} - 1$; all 1-bit) is used as a special `NULL` key value.

Sparse Tables. The implementation of Seam tables is designed (unlike Ebb) under the assumption that elements of the table will be dynamically allocated and deleted. To support this behavior a hidden `_is_live` field of `bool`eans is created to track whether a given key-value is mapped to an element or simply to free-space.

Another important difference is that the Seam prototype exposes external handles to individual rows of tables. Because `operation`s may delete these rows, the host-program must wrestle with the problem of invalidated references/keys. To support host-program queries and prevent aliasing between deleted and new rows, I added a second hidden field: the `ref_count`. This counter of type `uint32` keeps track of the number of extant references held by the host program. When a row is deleted with a non-zero `ref_count`, the Seam implementation will wait until the last references is released to release the key-position for re-allocation.

Tables maintain a free-list of free, un-allocated rows by re-using the `ref_count` space to thread the free-list in. When the end of the free-list is reached, all fields in the table are re-allocated at double their current size. For the prototype I did not implement any shrinking or de-fragmentation policies.

Indices. Unlike Ebb, Seam de-couples table ordering from indexing, and allows multiple fields to reverse-index a table. Therefore the kind of compressed range indexing from Ebb is no longer applicable. For the index associated to a field `X.f : Y`, we must maintain a data structure representing the map $y \mapsto f^{-1}(y)$. This map will generally have support over most of `Y`. Therefore, we start with an array parallel to table `Y`. At each (live) entry, we must store an arbitrary size set of keys from `X`. I used a dynamically re-sizable vector (holding keys in unsorted order) similar to the `C++` `std::vector` container. Insertion has amortized $O(1)$ cost, and removal amortized $O(m)$ cost for a set of average size m. Better data structures would be desired in practice. However, this was sufficient for the simple application I tested, where m was of constant size on average.

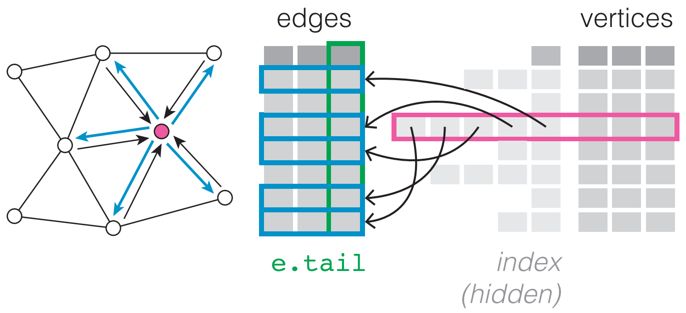


Figure 6.3: When the contents of the indexed table (`edges`) may change, the previous indexing data structure does not work. Crucially, there is no guarantee that the indexed rows are kept sorted according to the indexed field (`edges.tail`). Therefore, each row of the indexing table (`vertices`) must store a list of key representations, using dynamically sized arrays or some other such data structure per-row.

Views. Materialized views are stored in a similar manner to indices on the base tables. Let `V` be a view table with fields `k1:X1`, `k2:X2`, `k3:X3`. Based on analysis of code using query-loops over the view, a materialized copy is built for each field[4] used to loop over the view.

Consider a materialization of `V` built to support query looping over field `k1`. This data structure is built as a map from $X_1 \to X_2 \times X_3 \times \mathbb{Z}$, which groups all of the tuples and counts in `V` by their k_1 value. Increments and decrements of full tuples are performed by first indexing with k_1 and then binary searching for the (k_2, k_3) entry in the dynamic array stored at k_1. While this is the same as the Index backing structure, it is maintained in sorted order now. If a tuple can't be found, the operation takes $O(m)$ amortized copies to shift the array. If the count reaches zero, the operation takes $O(m)$ operations to compact the array. Lookups happen in $O(\log m)$ reads, and scans for query loops remain cheap. Increment and decrement operations are repeated for each materialized view copy.

6.3.2 Code Generation & Log Playback

Any `operation` maintains a log of effect operations during primary execution. This log contains separate lists for `new`, `update`, `write` and `remove` operations to help aid playback. When a `new` effect is encountered, a key must be immediately allocated in order to permit continuing with primary execution. This value is needed to assign in other `update` effects. Therefore `new`s are split into two-phases. During primary execution, a row is pre-allocated (using the `ref_count` rather than `is_live` field to track this) without performing any dependent updates to indexing data structures. These `new`s are committed during secondary execution (log playback).

Log playback (secondary execution) occurs after successful completion of the operation's primary execution. First `new` effects are committed (see preceding), followed by `update`s and `write`s. Finally `delete`s are processed. All of these operations are complicated by the need to carefully maintain the state of sparse table membership, indices and materialized views. Because indices and views are designed to never directly reference each other, this can be accomplished straightforwardly, even if it is

[4]when multiple fields are jointly used by a query loop, the compiler picks the "first" field according to a global ordering. This choice signficantly reduced the number of copies of a view that were materialized.

complicated to orchestrate the necessary code generation.

If an `abort` is encountered, then all logged `new`s are rolled back, releasing the associated memory and the log is discarded.

Finally, to maintain the views, the derived incremental view update code (Chapter 5) must be executed at the right time. After each `new dx` is committed, the associated $\Delta_{\texttt{+dx}}$ code is executed. Then, before each `update dx.f = dy` or `delete dx` effect is applied from the log, the corresponding $\Delta_{\texttt{f}}$ or $\Delta_{\texttt{-dx}}$ view maintenance code is executed. Interleaving updates in this way is necessary to ensure correct behavior because the derived code may invoke query-loops using the indices in the process of being updated.

View Generation. View definition code (normalized as described in Chapter 5) is code-generated wrapped in a for loop over the first argument, similar to Ebb `function`s. After all initial data is loaded (in bulk) into a Seam store by the host-program, this view generator is executed to populate the materialized views.

Invariant Checking. At the same time as view generation (after a complete initial copy of the data structure has been committed by the host-program), the code for each invariant is executed to check whether the invariant is satisfied for the loaded data structure. This works just like view generation—wrapping the body in a loop over the argument to the invariant.

Additionally, basic data consistency is checked on load. Every topological field value must refer to an existing row in the specified table.

6.4 Gong

Gong uses an execution model more similar to Ebb than Seam, based on wrapping code inside of looping constructs. However, the tables must also be designed to allow for creating and destroying rows—leading to data structure choices more similar to Seam. Wholly new (compared to Ebb or Seam), the Gong compiler must manage acceleration structure data and loop generation. The compiler programmer must also manage some novel effects, compiling them for both the CPU and GPU.

Like Seam, code is generated at a single compilation point, outputting a library and interface (whether `C`, `C++` or `terra`). While a dynamic and extensional approach (similar to Ebb) ought to be possible in principle, (there are not the kinds of circular dependencies as when extending Seam schemata) this choice is usually simpler for a compiler-programmer to architect the code around.

6.4.1 Data Layout & Management

Gong tables come in essentially three varieties. First, there are base tables, for which we expect no size changes except at load time. Second, there are derived tables (the target of an `emit`) that do not have a `primary_key` pair declared. Finally, there are derived tables with `primary_key` pairs, which may be the target of a `merge`. While both of the latter table types change size, only the merge tables need to be indexable by their primary keys[5]. As an additional simplification, I implemented the first class of tables via the second class, since (as we will see) the implementation performance should be identical to Ebb tables when no `emit`s are performed.

Keys. As with Seam, all keys are stored as `uint32` values. No special `NULL` value is reserved.

Non-Indexed Tables. Base tables and tables without `primary_keys` resemble the behavior of `C++` vectors. They may be appended to in amortized constant time (using a size-doubling re-allocation heuristic), may be cleared back to zero, and have a dense allocation of their key space. On the CPU (using serial execution) keys may be allocated in this way.

GPU-Resident Tables. Dynamic memory allocation and memory management on GPUs and in CUDA is generally a hard problem. For the prototype, I avoided

[5]For the purposes of the prototype, I avoided writing examples where Gong would have to manage indices for query-loops. As the `merge` tables suggest, this does not fundamentally exclude the relevant questions, but helps simplify them enough to explore in a prototype.

it altogether by requiring users to supply a maximum estimate of the size of any generated table (in terms of a function of input). The Gong prototype just CUDA-allocates this maximum space and crash-fails the entire library if the maximum is exceeded. The CUDA-allocated space is managed by maintaining a table-size counter. In order to allocate a row, this counter is incremented using an atomic increment instruction.

Indexed Tables. Derived tables with `primary_keys` must maintain a primary key index, which is inherently more complicated than the indices considered in Ebb because it must model a map $Y_1 \times Y_2 \to X$ when the table X has a primary key on the pair of fields `X.f1 : Y1` and `X.f2 : Y2`. I use an array dimensioned over the first domain Y_1 with each entry storing a sorted dynamic-size array (`C++` standard vector) holding pairs from $Y_2 \times X$. As with other indexing structures used in this thesis, this strategy is suceptible to skew (i.e. a non $O(1)$ number of rows at one or more Y_1 values). It was merely sufficient for the modest performance goals of the prototype constructed.

`merge` allows rows of tables with `primary_keys` to be both created and destroyed. Therefore, I maintain a free-list allocation policy (like Seam) threading the free-list through the `X.f1` values of unused rows. An `_is_live` field is used to track which rows are used when scanning over the table. (Note that no `ref_count` field is needed.) In order to support more consistent interaction with host code, I also provide a `sort` routine that compacts and canonicalizes the order of the storage. This can be accomplished in $O(n)$ time by simply scanning through the index structure and permuting rows.

Indexed Tables on GPU. GPU-resident tables with `primary_keys` use the same allocation strategy as all other GPU-resident tables: the simulation-programmer is expected to provide a sizing function. However, incrementally maintaining an index like the CPU one is prohibitively complicated. Instead, joins using `merge` and `emit` effects are augmented to sort the table by the primary-key values. Having done this, a compressed indexing array (similar to Ebb) is constructed to support fast lookups.

More details follow in the discussion on code generation.

6.4.2 Code Generation and Effects

Most of code generation follows ideas from the Ebb compiler. The main notable difference was that I used atomic operations for both field and global reductions, rather than use a two-stage global reduction tree.

Arg-Min Execution. I did not implement `argmin` reductions in the prototype compiler. One subtle issue is whether and how to ensure a stable result when multiple threads produce the same minimum result. Let `X.f` be the field being minimized on, and `a:A`, `b:B` the thread-id being executed in. Let v be the value currently being minimized into `x.f`. Then, construct a temporary hidden field parallel to `X.f` capable of storing `(a,b)` values[6]. If the minimization is instead performed with respect to the value $(v, (a, b))$, under a lexicographic ordering, then any ambiguity about which thread gets to write is eliminated—the "least" thread writing the ultimately minimum value wins. On the CPU, this encompases the extent of implementation complexities.

On the GPU, `argmin` is more complicated because the compiler-programmer is obligated to simulate an atomic operation on a number of dis-continuous memory locations. As a first note, observe that the contention can be significantly reduced by perfoming the atomic-`min` reduction as if the conditional write part was absent. Using the returned value from these calls, most threads can correctly identify that they do not carry the true minimum value. Other threads, succeeding in the min-operation would log their writes to a buffer (sized and managed similar to tables). This log could then be replayed with the true-minimum value in hand to govern which write actually succeeds. Note that more clever locking tricks are highly likely to violate a GPU's weak memory-model, which lacks global synchronization primitives.

On the other hand, it is worth noting that GPUs have long had at least one highly performant implementation of an `argmin` effect: the Z-buffer. The Z-buffer controls conditional writes to the rest of the framebuffers. Not only that, but it functions as

[6]The reduction is often being performed into some `A.f` on `a` the join parameter itself. In this case, the key could be reduced to `b` alone as an optimization.

a means of governing speculative execution. We can see shades of these semantics in the preceding early-exit trick. More significantly, the semantics for the `argmin` effect suggest a potentially novel concurrency abstraction for SIMD machines like GPUs; one with precedent and compelling applications.

Merge Execution `merge` statements are compiled into lookups using the index maintained on the destination table's `primary_keys`. If the lookup fails, then a row is allocated using the previously discussed data management policies: on cpu, an item is allocated off the free list. On the GPU, a row is allocated at the end of the active block.

On the GPU, `merge` operations make use of extra bits in the `_is_live` field in order to track whether a given row is *unvisited* (the default), *visited*, or *newly-allocated*. All of the newly-allocated rows occur in a block after the pre-existing rows. This distinction is then used to efficiently post-process the results of the merge.

Post-processing for a merge on the GPU consists of five steps (each of which is roughly a GPU kernel launch). In stage (1) a kernel maps over all live rows. If the row was *unvisited*, then the `remove` code block is run on the row at this time. If `keep` is executed, then the row is marked as *visited*. Regardless, a compressed primary key id is generated for each row. Unvisited rows that did not have `keep` invoked are assigned a maximum-value surrogate id. In stage (2), the compressed ids are sorted, generating a by-product parallel permutation array; and compacting the live rows by arranging all dead rows at the end. In stage (3), the compressed primary key ids are uncompressed, and then in stage (4), the index array is recomputed (see below). In stage (5) the rest of the data fields/columns are permuted into the new order using the by-product permutation array.

The index array is recomputed using binary search. A GPU kernel is launched with one thread mapped to each entry of the index (i.e. for each row of `Y` where the first field of the primary key is `X.f1 : Y`). Then, the i^{th} thread binary searches on the sorted `X.f1` to locate the first occurrence of key i. The key for this row of `X` is written to the index.

Emit Execution By contrast to merge, `emit` execution is much simpler. The semantics of `emit` have us first clear out the destination table of all contents. Then, new rows are simply appended at the end of the table during the join. If the table has a `primary_key` and is being computed on the CPU, then the index is built up by inserting entries as rows are added to the table. If on the GPU, then rows are simply added. A slightly simplified version of the above five-step post-process is carried out to regenerate the index after the operation. In particular, there is no need to track whether rows are visited or to run `remove` code.

Migration Experience While working on Gong, I changed the machine, CUDA, LLVM, and Terra versions I was building on top of. However, due to changes to CUDA[LG], I was no longer able to rely on the same PTX ISA instructions. Previously, I had implemented a warp-level contention reduction for atomic increments using special warp-ballot instructions—this was used to implement row-allocation for `emit` and `merge` effects. It had since been discovered that these operations were not safe. I switched to a naive per-thread increment operation instead. For whatever reason this appeared to now have no discernable performance penalty.

It is worth pointing out here that the semantics of the Gong effects remain unambiguous despite the flux caused by CUDA's weak memory model. One major reason for using higher-level domain-specific abstractions is the ability to buffer the simulation-programmer from these kinds of tricky details of parallel programming that regularly surface in high-performance abstractions that remain close to hardware.

6.4.3 Looping and Acceleration Structures

Most systems for collision detection abstract acceleration structure choice in some manner. However, this abstraction is often leaky. One important challenge for Gong was to figure out how to make this abstraction less leaky, so that data-storage abstraction and effect-checking remain valid. For this purpose, we adopt a template strategy. This strategy is a design trade-off. The compiler-programmer has the freedom to write arbitrary code in the template, but must manually port each template to each parallel target.

From the point of view of data-storage and code-generation modules, acceleration structure templates must implement the following stubs:

`StructLayout(DataAPI)`	The template generates any private storage it wants included in the data-store, representing the acceleration structure.
`PreLoop(DataAPI, ptr)`	The acceleration structure may do any maintenance needed to update itself whenever some join is called. For instance, a BVH is re-built or re-fit to the data.
`Invalidate(DataAPI, ptr)`	The acceleration structure is informed if any table or field data that it depends on has changed.
`LoopGen(DataAPI, ptr, row0, row1, body)`	Two acceleration structures are traversed, resulting in calls to the code-generated join `body` with appropriate `row0`, `row1` values bound.

All of these routines rely on a `DataAPI` that abstracts the data-storage from the perspective of the acceleration structure. This API has three main functions: `Size(tbl)`, `Scan(tbl,row,body)`, and `GetFunction(gong_fn)`. The last routine allows the template to get a handle to compiled user-defined functions that it was supplied with. (for instance, the `abstract` function supplied when creating a BVH index)

Scan

The scan template is very simple. On the CPU it nests a loop over the two tables. On the GPU it launches a kernel over every pair of keys. The CUDA blocks of 64 threads each are used to tile the product domain. That is, each block is assigned 8 keys from the first table and 8 keys from the second table. Tables not divisible by 8 simply mask off threads at the end.

Due to a choice of key-pair encoding in 32 bits, the scan strategy will cause the library to crash if executed on tables A and B such that $\#A \cdot \#B \geq 2^{32}$. For self-joins, this means that GPU scans are limited to tables of size $2^{16} = 64,000$.

BVH

The Bounding-Volume-Hierarchy template supports both CPU and GPU versions, albeit with very different construction and traversal algorithms. The CPU uses a joint-traversal of two trees, and a median-split top-down construction. The GPU meanwhile uses a spatial coding construction algorithm and scans one side of the join, inserting into the tree on the other side. Neither of these strategies are intended to be optimal choices. However, there is also no clear consensus on what constitutes optimal acceleration structure choices or algorithms in collision detection.

On both the CPU and GPU, the compiled `abstract` function is mapped over the base table (using `DataAPI.Scan`) along with `point` to compute a bounding volume and point representative for each row of the table.

On the CPU, the construction algorithm builds a binary BVH by the following recursive algorithm. Choose an axis (cycle through the options as the recursion descends). Find the median point/object along that axis using quick-select—this partitions the array into the items before the median and after the median in the process. Recursively partition these two sides until one arrives at 8 or fewer objects, which are collected in a leaf node. The resulting tree is a balanced binary tree. Volumes stored at internal nodes are computed in an upwards return pass using the `vol_union` function. After the tree structure has been computed, it may be more quickly updated by only performing the `vol_union`-based re-fitting. When a BVH is repeatedly re-used in joins, I use a naive policy to amortize build costs. Every 8th update to the acceleration structure triggers a re-build, with only re-fits in-between.

CPU traversal works (starting with the left and right root nodes) by alternately expanding the left-hand or right-hand node until a pair of leaf-nodes are arrived at. Then, their contents are scanned over. At each new pair of nodes, a `vol_isct` test is performed, and the traversal branch is terminated if the two volumes are not intersecting.

On the GPU, the construction algorithm sorts the data according to a Morton code (i.e. interleaving xyz coordinate bits as $\ldots x_1 y_1 z_1 x_0 y_0 z_0$). Then, splits are chosen in a single kernel using Tero Karras's trick[Kar12]. Finally, volumes are fit recursively. This results in a tree built by splitting objects along the octree-like divisions in space.

Following Karras, I chose to handle traversal asymmetrically. A BVH is only used for objects on one side of the join. The other table is mapped to GPU kernel threads, one-to-one. Each of these threads then descends the BVH tree independently, maintaining its own explicit stack as it goes. This choice may lead to a somewhat dramatic load imbalance between threads, especially in the case of relatively large geometric primitives such as ground planes.

Spatial Hash

I implemented the Hash template only on the CPU. It uses a relatively naive hash-table implementation, storing collisions using grouped-linked-lists of items. Let *bin* refer to a hashed-location and *slot* to a group of entries stored at a bin. Slots are sized to hold a `key`, a pointer to the `next` slot in the linked list, and as many row-ids[7] as will fit (along with the first two values) in 64 bytes. `NULL` values (where needed) are encoded as all 1 bit-patterns. A fixed-size array of bins (each holding a `uint32` reference to a *slot*) is allocated, sized according to `BIN_TO_ROW`. The slots are allocated and managed dynamically using a backing dynamic vector for storage.

For simplicitly, initially consider the case of a `Hash` structure supplied with an `abs_point` function and no `brute_force` function. To construct the table, the base table is `Scan`ned over, with `abs_point` evaluated on each row r producing a value k. Then, this value is converted to a bin location b by feeding it to `hash`. If the value at b is `NULL`, then a new slot is allocated with key k and next-pointer `NULL`. Its first row-id is set to r and the rest of the row-ids to `NULL`. If a slot is already allocated at b, then the linked list is followed until either a slot with matching key k is found with empty row-id space, or the end of the list is reached, triggering another slot allocation.

The above procedure is extended trivially to the case of `abs_range` by iterating

[7] I will refer to table keys as row-ids here to avoid the overloaded terminology.

over a set of key values for each row rather than just handling a single insertion. When a `brute_force` function is supplied, any row r where `brute_force` returns true is inserted into a supplementary `brute_force` list instead of the hash table itself.

Traversing a spatial hash consists of simply looking up values in the table without inserting values. At this point, every matching row must be found by scanning through the entire linked list at a bin. In the special case of a self-join, this can be accelerated by simply scanning over the slots themselves and following the linked list at each slot we scan. Since the allocation policy ensures that linked lists will only be threaded monotonically through the array of slots, we can safely assume that each pair of slots in the same list is visited exactly once using this strategy.

If both sides of the join use hashes with `abs_range`, then an additional deduplication control is necessary. This can be accomplished by (for each pair of objects encountered) running `abs_range` to find the minimum key values (potentially multidimensional) for those objects. By taking the component-wise maximum of those minimum keys, we arrive at a unique key position for which to test the intersection of the objects.

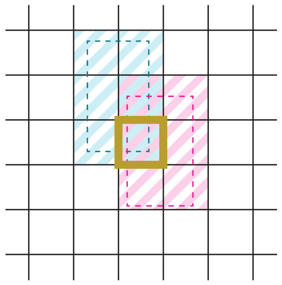


Figure 6.4: Hash Deduplication. Two boxes (dashed-lines in blue and magenta) are converted into key-ranges in the grid of keys (diagonally shaded cells). A test for intersection is *only* performed at the *least* cell of overlap (thick yellow border in the center).

6.5 Discussion of Compiler Organization

Over the course of constructing the three prototype compilers, I realized that the data-storage module was the most unwieldy and ill-defined component of these kinds of compilers. By comparison, consider the front-end of a compiler and how it progressively transforms text into a well-typed AST. Not only is this process well explained in introductory texts—to the point of being standard—we possess a great deal of wisdom about specifically how to structure this code. For instance, lexers are customarily written as stream-processing functions, and parsers are organized around the specificity of a BNF-grammar. The type-checking and other front-end passes are implemented as structurally recursive functions on trees in a purely functional way. This last point allows us to isolate our reasoning about compiler bugs/correctness, and check data for correctness in-between each pass.

How then, should we think about writing a data-storage module? What are the crucial programming disciplines to leverage?

My experience suggests that data-storage generation should be thought of in the following ways

1. A complete description of the data structure should be assembled in the most abstract possible way before beginning. This constitutes not only the `schema`, but also any directives about how that data should be stored (e.g. `grid`, `primary_key`, &c.)

2. If schemas are maintained as user-visible objects, this complete description should be generated as a copy of the schema, maintaining the immutability of the user-schema by the compiler code.

3. While `field`s can be organized as *children* of `table`s, indices cannot be. Indices should always be created as a set of secondary objects referring to the primary objects in a graph-like data structure. This data-structure can then be augmented in standard ways to allow other code to quickly locate all indices connected to a table or field in particular ways.

4. A data-store should be generated from this description in two passes. In the first pass, the description is used to establish the specific layouts for each component of the data-store. Then in the second pass, functions for looking up data and modifying it are generated. Without this distinction, one ends up in circularity problems trying to refer to layouts that haven't yet been created. This becomes especially evident when inserting a row in a given table triggers dependent operations on various indices.

5. The data-store's layout decisions should be abstracted from the rest of code-generation to the greatest possible extent. For instance, decisions about index-arithmetic and key-representations should not leak out into the entire back-end of the compiler.

In general, this level of modularity requires passing chunks of code between code generation and the data storage modules. Rather than try to dogmatically keep all code generation out of the data-storage, it is impotant to selectively place code-generation that is tightly coupled to data layout (e.g. table scanning) into the data-storage module.

This resulted in the idea that the interface between these two modules might be best described by an expansion of the set of effects. Under this discipline, the code to implement a given effect should be placed into the data-storage module by default. By the time I got to implementing the Gong prototype, this approach was yielding a much cleaner separation of code. Ultimately, these ideas made the `DataAPI` interface to the acceleration structure templates possible.

Chapter 7

Experimental Evaluation

7.1 Ebb

Ebb aims to capture a wide range of simulation domains, produce high performance code, and support interoperation with existing libraries. To evaluate whether we[1] achieved these goals, we selected four different problems that use different simulation domains — a fluid simulation, two finite element problems, and a hydrodynamics simulation.

FluidsGL [Goo07] is a semi-Lagrangian Stable Fluids simulation on a 2D grid, using a fast Fourier transform (FFT) at each step to solve the diffusion and projection system solves. Implementing FluidsGL in Ebb tests support for grids and ability to interoperate with external (FFT) code. Vega [SSB13] is a general purpose deformable soft-body simulation library using finite element method; our evaluation focuses on the code paths responsible for supporting the St. Venant-Kirchoff elasticity model on tetrahedral meshes. We use Vega to demonstrate support for ad-hoc data model specialization (§2.2), and the benefits of refactoring responsibility for parallelization into the compiler; Ebb runs considerably faster than Vega despite the sunk cost of the Vega team in rewriting and maintaining redundant multi-threaded code in their system. FreeFem++ is a high level language designed to solve partial differential equations using finite element methods [Hec12]. Instead of numerical algorithms, the

[1]Moreso than for the other prototypes, Ebb was a group effort.

programmer supplies a variational formulation of their simulation. In our comparison, we execute a deformable Neo-Hookean elasticity model in both FreeFem++ and Ebb, demonstrating that Ebb outperforms an equivalent FreeFem++ implementation by a huge margin and with modest code size. Lulesh [LUL12] is a hydrodynamics simulation on a hexahedral mesh used for evaluating different programming models for writing high performance scientific computing simulations. It has highly-tuned implementations for different architectures, providing the most rigorous stress test of performance out of the simulations we consider.

All code was compiled for and executed on a machine with an Intel i7-4790 CPU and an Nvidia Titan Black, GK110 Kepler architecture GPU. Reference code was compiled with `gcc` 4.9 with optimizations (-O3) enabled; reference CUDA code was compiled with `nvcc` 6.5. Along with total run times and code sizes, we also provide overhead of JIT compilation, which occurs once when the code executes for the first time, and the total memory used for storing data including constants, key fields, and hidden fields over relations. To measure memory allocation, we instrument Ebb directly, use a CUDA profiler for reference GPU code, and a malloc counting tool for reference CPU code.

7.1.1 FluidsGL

FluidsGL [Goo07] is a simulation of the Navier-Stokes equations for incompressible fluid flow written in CUDA, that ships with the NVIDIA CUDA 6.5 SDK. We chose it as an example because it is simple, short and uses a regular grid, allowing easy hand-tuning on the GPU. For these reasons, it serves as a good test for the quality of code Ebb produces. Furthermore, the simulation is based on an approach to Stable Fluids which requires a fast Fourier transform [Sta99]. FFTs are commonly used, heavily optimized, and exhibit communication patterns not well expressed as data parallel Ebb kernels. To get good performance, Ebb code needs to interoperate with an external library for calculating them.

The code calculates the velocity of a fluid as values on a uniform grid. Each iteration, the fluid is transformed from the spatial domain to the frequency domain where the diffusion and projection steps are computed, and then transformed back into the spatial domain. Point location is used to perform a cell-to-cell lookup when advecting velocity, and again for advecting a set of particles used to visualize the flow. In the reference implementation, the CUFFT library is used for the Fourier transforms.

Ebb also uses the CUFFT library to transform data into and out of the frequency domain—which is modeled by a second grid relation. Using the Ebb API, we request a direct view of the GPU resident field data managed by the runtime. In this way, CUFFT can operate directly on the memory, avoiding any additional cost of marshaling the velocity field data.

We compare the performance of our implementation to the original code (Figure 7.1) across a range of grid sizes. Despite using a relational abstraction for managing the data, Ebb is able to produce code that runs no more than 19% slower than the hand-optimized CUDA implementation. The total overhead for JIT compiling code is a constant 0.25 seconds, regardless of problem size.

Ebb's ability to interoperate with external libraries is important in this example. In both implementations, CUFFT accounts for around 30% of the total compute time of each simulation iteration, a sizable but not dominant part of the runtime. Overall performance is a combination of both fast simulation code and fast FFT code. Providing interoperability allowed us to use the best implementations for each part of the application.

For problems sizes of 512^2, 1024^2, 2048^2, and 4096^2 cells, Ebb uses 30MB, 100MB, 370MB and 1300MB respectively, while the reference uses 30MB, 70MB, 250MB and 900MB. Breaking this down into memory usage per-cell, Ebb uses approximately 80B-per-cell, while the reference uses approximately 53B-per-cell. Taking a careful tally of the underlying problem, an optimal implementation can get away with a 32B-per-cell overhead. (16B for a double-buffered velocity field, 8B for frequency-domain storage and 8B for particle position—there is one particle for each cell) Of the 48B of Ebb overhead, 16B is due to since-removed system inefficiencies, 24B is due to redundant representation of data within the user code, and a final 8B is unavoidable due to the programming model—specifically the need to maintain

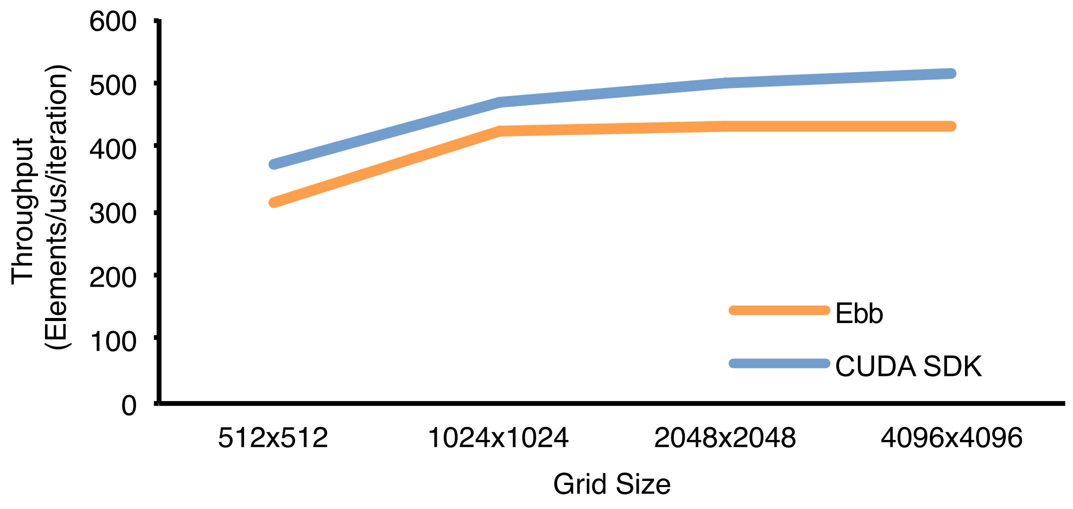


Figure 7.1: Our Ebb implementation of FluidsGL, an incompressible fluid flow simulation, compared against the performance of the implementation in the CUDA SDK.

explicit key-fields from the cells to themselves (semi-Lagrangian lookup) and from the particles to the cells. The reference code uses an excess ∼21B, at least 8B of which is attributable to superfluously storing particle velocity. As evidenced by this breakdown, neither FluidsGL, nor the Ebb version were written with much attention to optimizing memory footprint. While no greater degree of attention was paid to memory usage in the remaining comparisons, Ebb nonetheless consistently used half or less memory than the reference code.

7.1.2 Vega

Vega [SSB13] is a popular C/C++ physics library for simulating 3D elastically deformable solids. It supports a variety of integrators, elasticity models, as well as both tetrahedral and hexahedral domains. To make a comparison, we configured our Vega simulations to use implicit backward-Euler integration and the Saint Venant-Kirchoff elasticity model on a tetrahedral domain. After slicing out the relevant code paths, we found that Vega used 2500 lines of code for its single-core implementation. As such, we chose it as a demonstration of the performance Ebb can achieve on a larger program. Unlike the simpler FluidsGL, Vega's size makes it inherently more difficult and costly to optimize. While the library authors care enough about performance to have added a multi-core CPU implementation, no GPU implementation of it currently exists (to the best of our knowledge).

We wrote an implementation of the described Vega code path in Ebb. In doing so, we took care not to exploit the opportunity to refactor code across VEGA's abstraction boundaries. Details of the domain model topology are described in §2.2. Position, velocity, force and displacement fields are stored on the vertices and material properties on the tetrahedra. We store the stiffness matrix as a 3×3-`double`-matrix-valued field on the edges, effectively recreating the custom sparse block matrix data structure coded in Vega out of relational primitives.

At each time step, Vega computes internal elastic forces, stiffness, and damping, constructing a linear system that is solved to compute vertex velocities and displacements, subject to external forces. In Ebb, we solve this system using a Jacobi-preconditioned conjugate gradient solver (PCG) written entirely in Ebb kernels; as such, we can run this same PCG solver on the GPU. When run on a single core, Vega uses a similar PCG solver; on multi-core we have Vega use the multi-threaded Pardiso solver from Intel MKL [KLS13] as well as the separate multi-threaded force model and integrator implementations. Solvers often have different performance characteristics than other simulation code. To ensure they run well in Ebb, we allow writers to optionally annotate kernels with underlying characteristics such as block size, which can increase performance by up to 2x for some solver kernels.

Figure 7.2 compares the performance of Vega in Ebb to the original Vega code for a few different meshes. The simulations in Ebb take 1.75 seconds to JIT compile once at the beginning, which is excluded from the figure. Note that our single core implementation of Vega matches the performance of the original code. We are also able to use Ebb's CUDA backend to generate a GPU implementation of Vega. Previously Vega could not run on GPUs. Our automatically-generated implementation runs 9 times faster than the serial implementation, and from 4 to 9 times faster compared to the best multi-threaded implementation on our CPU. For simulation library developers, this approach is much simpler and easier to maintain than translating reference

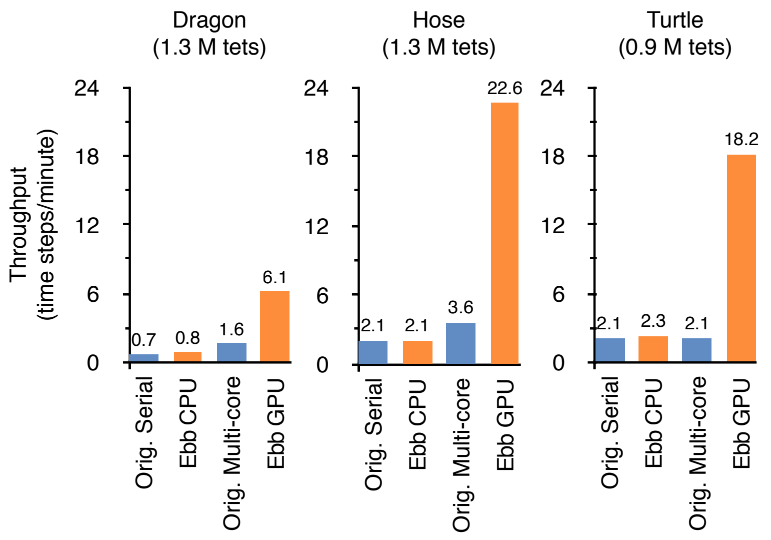


Figure 7.2: Ebb and reference average times per iteration, for 3 different meshes. Ebb GPU performs 9 times faster than reference serial, and 4 to 9 times faster than the best of serial and multi-threaded reference implementation over 8 cores.

code to GPU code by hand, which would require rebuilding data structures for GPU, rewriting the solver, and carefully considering the implementation of reductions to avoid race conditions.

Abstracting the domain via relations also simplified our implementation of Vega. The original C/C++ code slice consists of over 2.5K lines of code for single-core computation, over 400 lines for loading tetrahedral meshes, and an additional 800 lines of code to implement multi-threading, not including the external solver. In total VEGA takes over 3.7K codes to implement the exercised code paths. Our Ebb application is under 1K lines of simulation code, plus a 400 line tetrahedral mesh domain library, resulting in less than 1.4K lines total. Ebb code is more concise due to automated memory management, and the abstraction afforded by relational primitives (encapsulated in macros). The reference code, on the other hand, explicitly encodes the mesh as arrays augmented with specialized indexing structures for adjacencies. Furthermore, it requires the implementation of a separate sparse matrix class with fast indexing structures. In our Ebb implementation, we were able to embed this sparse matrix structure as a matrix-valued field over edges, resulting in fewer lines of code. By avoiding redundant representations of the data, Ebb ends up using only 1.09GB, 1.06GB and 0.36GB for the dragon, hose and turtle meshes, compared to 2.35GB, 2.26GB and 1.46GB respectively for the reference code.

7.1.3 FreeFem++

FreeFem++ is a high level language designed for solving partial differential equations over meshes [Hec12]. Problems in FreeFem++ are modeled using a variational formulation, which is closer to how physicists model partial differential equation problems. By comparing a FEM simulation in Ebb with one in FreeFem++, we evaluate the productivity vs performance tradeoffs of using a higher-level abstraction than the one Ebb offers.

We obtained a deformable simulation using a Neo-Hookean elasticity model, that was written in FreeFem++, with input from the FreeFem++ developers on the correct way to use their system. The simulation uses a conjugate gradient solver to perform implicit integration. We implemented the same elasticity model, with the same external conditions, numerically in Ebb. Similarly to the Vega comparison, we store position, velocity, force and displacement on vertices, material properties on the tetrahedra, and stiffness matrix on the mesh edges. We reused our implicit integrator and conjugate gradient solver from the Saint Venant-Kirchoff example. Counting both this integrator/solver and new code, our Neo-Hookean simulation written in Ebb requires about 800 lines of code. We also reused the tetrahedral mesh domain library from the Saint Venant-Kirchoff comparison, which was 450 lines of code. While typical FreeFem++ problems take tens of lines of code, the various tensor components introduced by the Neo-Hookean model made the FreeFem++ code about 700 lines long.

We use FreeFem++ version 3.36 for evaluation. FreeFem++ takes about 41.3 seconds to run one time step on a tetrahedral mesh representing a sphere, with 2.4K tetrahedral elements. On a large bunny mesh with 78.7K tetrahedral elements, FreeFem++ takes about 28.3 minutes to run one time step. Ebb completes each time step on the sphere mesh in 0.006 seconds (6800 times faster) and on the bunny mesh in 0.26 seconds (6500 times faster), on a CPU. These meshes are too small to give any significant speedups on GPU—we get an additional speedup of 2 for the bunny mesh on GPU. The one-time overhead to JIT compile kernels is about 0.22 seconds when running on GPU and 0.4 to 0.6 seconds when running on CPU. Though Ebb can run this simulation on larger meshes, with an even larger speedup on GPU, we did not evaluate FreeFem++ with larger meshes due to the large running time.

Ebb performs better than FreeFem++ because the Ebb simulation computes elastic forces and stress using an algorithm that is specialized for the Neo-Hookean model. This is possible in Ebb, even with relatively concise code, because of the sufficiently low abstraction level that Ebb offers, while nonetheless relieving the user from the burden of memory management, low-level data structure construction, and parallelization.

7.1.4 Lulesh

We also implemented a version of Lulesh (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [LUL12], which is a standard benchmark for evaluating the performance of simulation code across a variety of programming models. It has highly optimized implementations for a variety of languages and platforms [KBC+12], allowing us to evaluate the quality of the code Ebb produces relative to highly-tuned implementations. Lulesh also uses a semi-structured hexahedral mesh, exercising another domain model.

Lulesh models the propagation of a Sedov blast wave, using explicit integration. It stores thermodynamic variables, such as energy and pressure, mapped over hexahedral elements, and kinematic values, such as position and velocity, mapped over nodes. The iterative algorithm consists of a phase that advances node quantities, a phase that advances element quantities, and a phase that computes all values at the next time step.

Figure 7.3 summarizes the results for Ebb compared with implementations of Lulesh that were hand-optimized for serial and GPU execution. The serial version

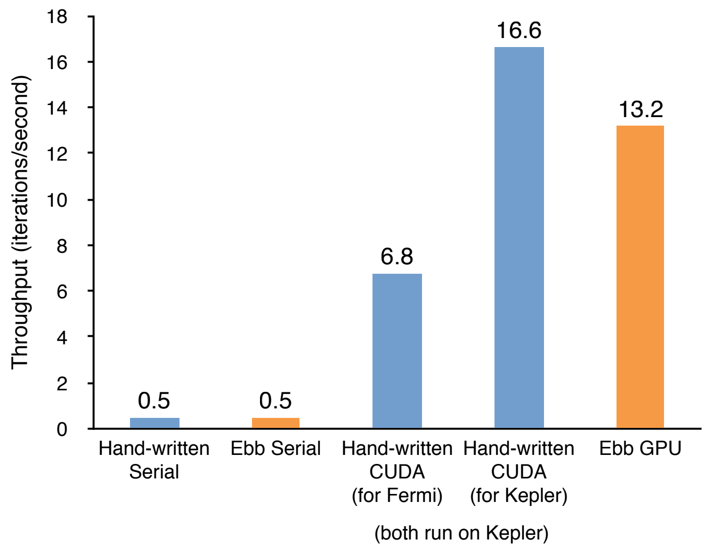


Figure 7.3: Ebb and reference GPU implementations of Lulesh, compared against the reference serial implementation. The simulation was run over a 150x150x150 hexahedral mesh for 2,527 iterations.

of Ebb performs at the same speed as the hand-written reference. Our GPU implementation runs about 24 times faster than the serial code, and within 27% of the performance of GPU code hand-tuned specifically for the Kepler architecture. Performance characteristics of GPUs can change over generations. The original CUDA version of Lulesh code was written for the previous Fermi GPU architecture and performs worse than the Ebb implementation. One advantage of writing simulations at a higher level is that changes in architecture can be handled by the compiler rather than by hand. For instance, the addition of new atomics or changing best practices for reductions can be addressed within the Ebb compiler.

The time to JIT compile Lulesh code in Ebb is about 1.5 seconds. Ebb uses 1GB memory, compared to 2GB memory by reference code, for the 150^3 hexahedral mesh.

We perform well compared to other domain-specific language implementations. On a 45^3 sized mesh, the Liszt[DJP$^+$11] language can run Lulesh at 176 iterations per second, while Ebb can run the same simulation at 340 iterations per second. Liszt struggles with the Lulesh benchmark since its coloring-based approach to reductions does not work well for compute-heavy kernels [KBK+13]. Furthermore, Liszt uses an unstructured mesh as its only built-in domain, making it difficult to express code that assumes each element is a hexahedron. The unstructured mesh model also requires more memory to represent (Liszt could not fit a 150^3 mesh into memory on our GPU).

Ebb makes implementing Lulesh easier compared to hand-written GPU models. Ebb code (1.3K lines) is less than half the size of the GPU implementations (around 3.5K lines), and about the same size as a serial implementation (2K lines). This difference is largely because Ebb automatically handles synchronization, parallel reductions, data movement, and selection of block sizes for CUDA kernels. While the reference CUDA implementations explicitly include block reduction code needed to compute a minimum time step, Ebb generates that code automatically. Additionally, Ebb's built-in library for modeling hexahedral grids simplifies mesh construction, while automatic memory management simplifies the application code, resulting in a further reduction of lines of code, compared to the reference serial code.

7.2 Seam

Evaluation of the correctness checking aspects of Seam are outside of the scope of this thesis. Please see the original paper[PBS+17] for those experiments and details if you are interested.

7.2.1 End-to-End Application Execution Measurements

I implemented a version of the *Enright Test* [BB09] using a combination of Seam and Terra code. This test uses a geometric flow to severely distort and then un-distort a sphere over the course of 300 computed timesteps of simulation. Remeshing is performed after every timestep. Additionally, I modified the Enright Test implementation in Brochu's ElTopo codebase [BB09], and implemented a version of the test using the remeshing code from my old TopTop codebase [BW13]. Both ElTopo and TopTop are written in C++. All three versions were written or modified to remove all remeshing/improvement operations outside of edge-split and edge-collapse, which was the common denominator supported by all three. The scheduling of these operations differs between the three systems: ElTopo priority sorts an array and makes multiple passes until no more operations need to be applied; TopTop uses a priority queue; thes Seam code simply makes a single pass over the edges that exists at the outset of a remeshing pass. To reduce the effect of this variance, I restricted all 3 systems to make only one pass over the edges (per timestep), in arbitrary order.

All code was run on a 2015 Macbook with an Intel Core M-5Y71 processor clocked up to 1.3 GHz and more than enough RAM. Total wall-clock times for the three variations were: 4.5 sec (Seam), 10.8 sec (TopTop), 1145 sec (ElTopo). Peak memory usage for the three variations was: 4.9 MB (Seam), 3.7 MB (TopTop), and 6.8 MB (ElTopo). Profiling revealed that TopTop spent around 45% of its time generating a `TopoCache` data structure at the start of every remeshing pass. In terms of Seam, this is analogous to regenerating the views and indices. This data structure is then "committed" and discarded at the end of a remeshing pass. Profiling ElTopo reveals that it spends around 90% of its time in a `defrag` pass over the data at the end of every remeshing pass. Because all references to an element must be updated whenever an element is moved, this re-arrangement pass can become quite expensive. By contrast, Seam uses free-lists and strictly incremental maintenance of data structures—the data structure is fully valid after the completion of any individual remeshing operation.

The entire Seam application, including the test harness, is about 800 lines of code, 135 of which are Seam code. TopTop's data structures and application code take about 3000 lines, after slicing out unused code and support libraries. ElTopo—which is less obvious how to slice—is a library of 25,000 lines of code. We can also look specifically at the code required to write edge split and collapse in the various systems. Seam takes 22 and 33 lines respectively. TopTop takes 212 and 300 respectively. ElTopo takes 225 and 344 respectively. This comparison suggests that using Seam results in an approximately 10 times reduction in the amount of code required to write local edit operations, even if we ignore other necessary support code.

What should we deduce from these comparisons? First, and most importantly, existing code for dealing with complex pointer data structures tends to not be heavily performance tuned. A Seam back-end that I know contains significant performance flaws can achieve better performance than hand-written C++ code. Second, in the absence of more automated memory and indexing management, programmers are forced to make trade-offs between performance and comprehensibility in this kind of complex remeshing code. Third, this memory management and index maintenance code leaks into higher-level algorithms rather than remaining encapsulated; as a result, it becomes disproportionately difficult to change memory management or indexing strategies in pre-existing code. Fourth, and finally, handwritten code is an order of magnitude more verbose than equivalent code written in Seam.

7.3 Gong

The primary goal of Gong was to demonstrate that the relational model of Ebb could be extended to handle collision detection problems. More specifically, I wanted to show that Gong could retain high-performance across CPU and GPU, and allow for exploitation of problem-specific characteristics, all while modeling a range of different collision detection problems. To demonstrate as much, I selected three problems: oriented-box collision, sphere-sphere collision, and arithmetically exact edge-triangle intersection.

To provide a challenging test of performance, I ran a set of stacked-box rigid body simulations using Bullet[Cou15] and Gong. As an open-source rigid body simulator used in dozens of films and games, Bullet represents state-of-the-art collision detection software. However, in practice many researchers and developers simply implement their own acceleration structures, as Jonathan Leaf did in his yarn-level GPU simulation work. Arguably doing so allows for exploiting problem-specific details in ways that libraries like Bullet cannot. I compared his sphere-sphere collision detection on the GPU to a Gong re-implementation, showing both that Gong allows for exploiting problem specific details and is able to achieve higher performance—in part by switching techniques depending on the problem size. Lastly, I show that exact edge-triangle collision detection can be implemented in Gong by re-implementing that part of my CSG library Cork. This last comparison shows that Gong can express a wide range of collision problems, by providing (to my knowledge) the first-ever port of extended-precision arithmetic predicates for intersection onto the GPU. Making code that requires specialized expertise more portable underscores my basic thesis: that separation of concerns enables more progress on specialized techniques.

All code was compiled for and executed on a machine with an Intel i7-i7-6700K CPU running at 4GHz and an Nvidia Titan X, Pascal architecture GPU. Terra was compiled against LLVM version 3.8.1, and CUDA 6.1. CUB-1.8.0 was used for sorting routines.

7.3.1 Bullet Boxes

Bullet is a rigid-body dynamics library that handles collision detection as one important sub-computation. I reimplemented Bullet's oriented-box collision sub-routine in Gong, and re-implemented its Sequential Impulse (projected Gauss-Seidel) solver in accompanying Terra code, replicating behavior as closely as possible. (Because the solver is order-sensitive, this was not perfectly possible) I used the Gong `primary_key` sorting functionality to ensure exactly-consistent behavior[2] of the Gong code with different acceleration structures and target platforms.

This comparison to Bullet is important because Bullet has been signficantly more heavily optimized than most research code. Reproducing close to its performance supports the claim that the abstractions introduced by Gong have no intrinsic reason to be slower than highly tuned general purpose collision detection systems.

Secondarily, Bullet provides us with an opportunity to look at the complications that arose with a real port of existing industrial code: Erwin Coumans previously ported Bullet onto OpenCL[Cou13]. However, in doing this, all of the acceleration structures, and algorithms were rewritten. Both for this reason, and because of the GPU memory model, the entire library API had to change, resulting in the version 3 revision of Bullet. Principal among these API changes was internalizing memory management of Bullet objects, rather than letting the library-user allocate the objects themselves. In the process, any hope of reproducing (on the GPU) the exact contact

[2]because of fused-multiply-add operations when compiling to GPUs this was still not strictly the case.

	600 boxes	1200 boxes	2400 boxes
Bullet Physics	1.115	2.135	4.673
Gong			
CPU Scan	2.266	7.030	23.567
CPU BVH	1.198	2.443	4.888
CPU Hash	2.626	5.184	11.665
GPU Scan	0.638	1.094	2.516
GPU BVH	4.752	5.101	6.789

Figure 7.4: Round Box Tower Timings. average collision time in ms.

point sampling algorithm from Bullet 2's CPU collision algorithm appears to have been abandoned. By contrast, the Gong implementation ports the exact method from Bullet 2 onto the GPU. By ensuring that code can be ported without changes to behavior or interface, algorithmic changes can be more cleanly separated from hardware ports.

The full Gong implementation of the join was around 1000 lines of code. It is difficult to isolate and count the collision code in Bullet, but the box-box collision function alone takes 750 lines of code, which is about the same as in Gong. This is unsurprising since I attempted to replicate the idiosyncracies of the `dBoxBox2` collision algorithm exactly.

I ran Gong on a series of tower-collapse simulations using 600, 1200, and 2400 boxes, each one initiated by tossing a block into the stacked boxes. The initial conditions were replicated in Bullet exactly. I verified general consistency of behavior visually, and additionally verified that the number of collisions per frame stayed within 5%. Measurements are presented in Figure 7.4.

I measured every possible variation of CPU/GPU and acceleration structure choice for the Gong version of the experiment. Given the relatively small number of objects, I found that a brute-force scan on the GPU could still run approxiamately twice as fast as the best CPU alternatives. Applying the GPU BVH to this small problem had at least three short-comings: (a) there is not enough work to saturate the machine with only $600-2400$ threads; (b) the computation is highly load-imbalanced, especially the large box representing the ground-plane, for which all tests are serialized in a single GPU thread; (c) the BVH build overhead is more significant on a small problem like this one.

On the CPU, the Hashing strategy did not work well for this problem. The oriented bounding box intersection test is a very compute and data-heavy operation. As a result, acceleration structures that are more precise (i.e. culling more potential tests earlier) are more advantageous.

The CPU BVH, despite being a less carefully tuned implementation than the Bullet acceleration structure ran no slower than 1.15× Bullet at the worst. I did not do anything clever to try to get better performance, so this is predictable.

More importantly than small differences in performance cost was the ability to reproduce the behavior of Bullet exactly. The collision detection behavior for Bullet is extremely idiosyncratic. So this serves as strong confirmation of the practical expressivity of Gong. In this vein, the `merge` feature was especially important to implement this example. Without it, this close correspondence of behavior would not have been possible to reproduce.

7.3.2 Cloth-Yarn Simulation

The cloth-yarn simulation I studied[LWS$^+$18] was based on a code-base originally in Java[KJM08] using BVHs for collision detection acceleration. The specific collision problem being modeled involved a large set of spheres sampled from the yarns, each of identical radius. This problem is amenable to spatial hashing, which Jonathan Leaf[LWS$^+$18] implemented on the GPU. In the original work, Jonathan Kaldor[KJM08] used a CPU BVH. While alternative approaches (spatial hash on CPU, or BVH on GPU) could have been tried, each additional approach would have required a substantial investment of time to implement and optimize. Alternatively, a standard library could have been used (such as Bullet[Cou15] or FCL[PCM12]). However, neither of these libraries implement spatial hashing, because it doesn't work well *in the general case.* Spatial Hashing requires the simulation-programmer to choose appropriate grid-size parameters based on query/problem-specific knowledge.

The Yarn simulation problem was interesting from a number of perspectives. First, it was research code with a GPU implementation. I had talked with the author about ways to further accelerate their specific problem, so I am sure that non-trivial attention was paid to performance. This can be interpreted as an optimistic view of the performance many researchers are likely to get by implementing their own custom collision detection.

Second, the yarn collision problem is very arithmetically simple compared to the other two queries (Oriented Bounding Boxes and Exact edge-triangle Intersection predicates) that I looked at. As such, this query stresses the efficiency of acceleration structures more severely.

Third, as mentioned prior, this query is a near-optimal situation for spatial hashing techniques. Therefore, this test can justify the benefit of supporting multiple acceleration structure choices in Gong, and the benefit of specialization to the specific query. I used a specific dilation strategy to compute this join asymmetrically. One copy of the spheres was hashed using `abs_point` on the sphere centers, and then the other copy probed the table using double radii spheres via `abs_range`.

I relaxed three knit-patterns of varying size using Jonathan Leaf's 'Yarnsim' (Figure 7.5). Timings were taken in situ for Yarnsim. Lists of sphere positions were also dumped to file and loaded into a Gong test harness (for expediency). This test harness was used to execute four variations. Since CPU Scans were embarassingly slow, their measurements were omitted.

Observe three essential phenomena in these timings. First, when looking at the Gong CPU times alone, we see that spatial hashing enjoys a consistent advantage. This tenatively supports the value in supplying special purpose acceleration structures. Second, note that Gong's GPU implementation outperforms Yarnsim. This supports the overall thesis in this dissertation that separation of concerns ought to allow performance improvements. Lastly, observe that at small enough sizes the GPU occupancy of the BVH becomes so poor that a brute-force scan is able to better utilize the GPU.

The Gong implementation uses very little memory and hardly any code. This is largely an artifact of it being divorced from the full simulation. Rather than try to

	Yarn Patterns		
	Seersucker	Alternating Diag.	Stockinette & Garter
# spheres	$3,420$	$10,932$	$31,860$
Yarnsim (ms)	1.95	2.73	4.67
Gong (ms)			
CPU BVH	2.44	7.96	30.53
CPU Hash	1.20	3.82	11.05
GPU Scan	$\boxed{0.73}$	7.82	61.58
GPU BVH	1.12	$\boxed{1.57}$	$\boxed{3.00}$

Figure 7.5: Yarn Spheres. average collision time in ms.

provide a meaningless comparison of memory usage or code size, I omit those here.

7.3.3 CSG: Cork

The most expensive part of computing polyhedral Booleans (aka Constructive Solid Geometry) is determining the intersections between the two (or more) surfaces. Specifically, one must find all of the edge-triangle intersections. In the Cork library[Ber14], the two surfaces are aggregated into a single triangle-mesh, on which edge-triangle self-intersections are computed. Doing this robustly is a historically difficult problem, requiring exact, often filtered predicates[She96]. My library Cork implements the exact-fallback for these predicates in big-integer arithmetic, specialized to custom bit-bounds, using some underlying routines from GMP[GT15]. These bit-bounds are computed using `C++` template-metaprogramming parameters. Therefore, in order to port this computation to Gong, I had to re-implement this metaprogramming using Lua; and provide a new implementation of the low-level limb addition/multiplication routines from GMP. While my generic (non-processor specialized) versions of the GMP routines are likely slower, they are also size-specialized in ways that the GMP library could not be. The result appears to have been largely a wash—though I did not attempt to precisely quantify the effect.

While CSG is not strictly a collision detection problem, similar exact-predicate methods have been used by Brochu et al.[BEB12] for continuous self-collision detection in cloth simulation.

A collection of 13 models were sampled from the Thingi10k[ZJ16] dataset. Each model was rotated a small amount around a random axis and unioned with itself. The resulting number of edges and triangles fed to intersection, as well as the timings to compute are shown in Figure 7.7. Because of the size of the problems and non-general applicability of hashing, only BVH variations were tested. The baseline is my old `C++` implementation of the intersection test inside of Cork, which the Gong library was compiled into for variation timings. (demonstrating interoperability with existing software) Timings are only of the intersection sub-computation, which dominates the runtime of CSG operations.

	#edges	#triangles
HexPentaPot	11532	7688
FourInARowV2	35292	23528
pcb_vise_v2-Swivel	36468	24312
Wizard_Hat	44502	29668
Shamrock_Shot	57870	38580
octopus	61026	40684
stylo2TOM	77370	51580
soapPumpRotor	78108	52072
dual-dodec	104400	69600
inverted-bracelet	179520	119680
metatron	415140	276760
GOYLE_LOW	589824	393216
TestForms01a-Final01b	2264064	1509376

Figure 7.6: union with self rotated; problem sizes.

In every case except for the pcb_vise_v2-Swivel object, GPU execution sped up intersection time by at least a factor of 2. Examining the exceptional case revealed that the model contains large, long and skinny triangles tesselating large flat planes in the object being modeled. These triangles and edges almost certainly caused bad load balancing for the naive BVH traversal algorithm I relied on.

This routine is a fairly large collision program. The Gong implementation took around 1725 lines of code, of which 825 were the metaprogrammed exact-arithmetic library. The original Cork implementation took around 1375 lines of code.

This is (to the best of my knowledge) the first implementation of exact predicates for polyhedral Booleans running on a GPU. While this is notable, of particular interest is the way that different specializations are de-coupled by Gong's design. Developing exact predicates and exact arithmetic is a highly-specialized skill, as is GPU programming and high-quality acceleration structure design. While I can hardly claim to have done any one of these tasks optimally, the decomposition proposed by Gong allows for different specialists to separately do their part of this work. Looking at a large scale evaluation done by Zhou et al.[ZGZJ16], we can see that Cork is already one of the fastest available libraries for this computation. As a result, this new GPU

	Cork	Gong $\begin{bmatrix}\text{CPU}\\\text{BVH}\end{bmatrix}$	Gong $\begin{bmatrix}\text{GPU}\\\text{BVH}\end{bmatrix}$	Speedup
HexPentaPot	117	118	65	1.8×
FourInARowV2	142	146	54	2.63×
pcb_vise_v2-Swivel	109	106	184	0.59×
Wizard_Hat	144	152	88	1.64×
Shamrock_Shot	181	180	56	3.23×
octopus	115	118	33	3.48×
stylo2TOM	261	227	49	5.33×
soapPumpRotor	153	143	48	3.19×
dual-dodec	222	190	23	9.65×
inverted-bracelet	457	436	159	2.87×
metatron	878	783	198	4.43×
GOYLE_LOW	961	917	172	5.59×
TestForms01a-Final01b	4021	3914	675	5.96×

Figure 7.7: union with self rotated; timings in ms.

implementation is probably the fastest robust boolean implementation yet developed.

7.4 Summary

With the experimental evaluation of Ebb, I showed four comparisons, illustrating how Ebb was able to interoperate with existing code, model a range of different geometric data structures, and achieve performance competitive with hand-tuned GPU code.

With the experimental evaluation of Seam, I showed more modestly that order of magnitude reductions in the amount of code required to write edge-based triangle remeshing operations was possible—while simultaneously improving performance.

With the experimental evaluation of Gong, I showed that a range of different collision detection problems can be expressed as spatial joins, enabling de-coupled exploration of both hardware target and acceleration structure strategies.

Because all three of these prototype compilers were designed with interoperable data models, these results together show the viability of relational data modeling to address the problem of performance portability for physical simulators.

Chapter 8

Related Work

8.1 Parallelism Mechanisms

The entire scope of parallel computing cannot be adequately represented by a brief discussion. However, a number of basic mechanisms deserve discussion. One of the earlier forms is symmetric-multi-processors (SMP) that place multiple CPUs on a single shared memory. Multi-core processors (where the multiple CPUs share a single package/die of silicon) use the same model, programmable via libraries such as pthreads. Processors were also developed for finer-grain parallelism via SIMD (Single-Instruction, Multiple-Data) or *vector* processors. Such processors have a fixed width (such as 4 on many Intel CPUs, or 32 on GPUs) so that every instruction executes on that many consecutive data registers at once. This model tends to rely on special machine instructions, and has not really been standardized in the way that thread libraries have.

In the 90s, Guy Blelloch developed NESL[Ble90] (NESted parallel Language) to map tree-structured computations down to vector machines. Some of the key inventions of this line of work were segmented-vector operations as a target for flattening, and the importance of parallel-prefix-sum primitives for data-parallel algorithms. Around the same time, Cilk[FLR98] approached similar kinds of irregular tree-structured parallel programs (fork-join parallelism) with the goal of mapping down to thread models. Its main invention lay in the use of work-stealing queues to efficiently manage load-balancing. This approach found its way into an Intel implementation, and Apple's Grand Central Dispatch[Nah11] among other parallel runtimes.

Some of the ideas of NESL were repurposed by Ian Buck et al. in Brook[BFH$^+$04] to make GPUs programmable. This model was developed into CUDA which combines vector programming ideas with a hierarchy of shared memories in an idiosyncratic way. OpenCL[MGM$^+$11] was proposed as a cross-manufacturer standard for GPU programming, but recently collapsed with the withdrawl of Apple in favor of their own Metal GPU programming language.

At the level of multiple machines connected over a network, MPI[For94] (Message-Passing-Interface) has remained the dominant API for accessing network capabilities to move data around. There is some interest in global addressing as an alternate view, as in UPC[EGCSY03]. However, this model is proposed largely as a programmer simplification and not as a reflection of actual distributed machine architectures—which all have node/processor-local memories.

In this thesis, I focused on GPU implementations because they require attention to fine-grained parallelism and are often overlooked by supposedly general parallel programming models. The most significant shortcoming of such a research approach is a lack of attention to data partitioning complications that arise at the scale of distributed machines, such as for supercomputers.

One important theme to take away from this discussion is the relative lack of standardization and stability for many basic lower-level parallelism APIs. While some of this might be attributable to market competition between various hardware manufacturers—i.e. the desire to produce platform lock-in among developers—there are equally good technical reasons for API instability. Successive generations of hardware frequently add new valuable parallelism features which require breaks with existing APIs.

The large degree of disagreement in shared memory consistency models for SMPs and GPUs is an excellent example of such pressures. Weaker memory models often allow for more efficient parallel architectures/programs. However, these models are often widely misunderstood. NVidia's CUDA model was shown to have several such shortcomings[ABD+15], which along with future hardware changes helped prompt revisions to the CUDA standard. I was caught by some of these changes while developing Gong and had to change the implementation to become compliant. (§6.4.2)

Regardless of the source, this turbulence in programming model causes software maintenance problems, such as the recent deprecation of OpenCL support by Apple. Client code must therefore maintain some degree of indirection in order to buffer code-bases and teams from these changes. This motivates the thesis that abstractions closer to the application domain are necessary. The fact that these changes occur to instruction sets and other detailed aspects of code generation serve as motivation for the thesis that this abstraction must necessarily be a language.

8.2 Safe Parallelism and Effects

One basic idea for analyzing the safety of parallel operations is data-race freedom, via safe commutation of memory-effecting operations[Ber66]. That is, the equivalence of sequentially executing two operations A and B in either order AB or BA constitutes a kind of formal proxy for the claim that A and B may be executed concurrently/in-parallel. Depending on the particular formalism, this “commutative” concept may surface as other properties like associativity of binary operators to much the same purpose.

In and of itself, the commutativity of operations (in the sense above) does not trivially guarantee safe parallel execution. Safe atomicity or interaction must be provided for. For instance, different sum-reductions into a particular memory address commute, but are only implemented safely if performed using a special atomic reduction instruction. In this thesis, *effects* bind together the abstract-formal and practical-implementation perspectives.

By contrast, many other approaches rely on locks, which can produce signficant overheads. (Software) Transactional Memory[CBM+08] attempts to mediate parallelism safely by ensuring atomicity of code blocks in a composable fashion. Targeting the GPU immediately ruled out this approach for me. For the sake of efficiency, any parallel conflicts on the GPU must be mediated by low-overhead atomic operations (e.g. increment) or amortizing bulk operations, as in the use of sorting to maintain indices after a join. Locks (outside of perhaps small-scale compare-and-swap spin-waiting) are not an option for GPU programs.

As Rinard notes[RD97], most approaches to parallelism in compilers rely on dependence analysis, which imposes stricter notions of causation than commutativity allows for. The approach here is closer in spirit to Kawaguchi et al.'s type-and-effect system[KRBJ12], which is built on refinement types. My effect-checking analyses in Ebb and Gong are much simpler and naive by comparison.

Work on the Galois programming system[PNK$^+$11] previously argued that parallel programming languages should more explicitly treat higher level data structures in order to make compiler analyses tractable. My work holds to the same philosophy with a more specific proposal about the nature of those data structures. Namely, I explore the benefit of leveraging relational data models from databases.

8.3 Database Concepts

Many authors from Codd[Cod70], to Stonebraker[HS05], to Helland[Hel16] have argued for the merits of relational algebra models (eventually in the form of SQL) for databases.

One of the essential properties argued by Codd is the separation of physical and logical data models. This meant specifically moving away from the CODASYL model that required database client-code to specify the path to access data in a hierarchy, similar to a filesystem. The languages proposed here occupy a range of intermediate positions on this hierarchical vs. relational access distinction. On the one hand, all of the languages require the programmer to chase references around and express a specific loop-nesting order. That is, Ebb/Seam/Gong code has a particular join-plan hardcoded into it. On the other hand, the specific representation of data indexing is not hard-coded in, and in some cases (§5) the code can be effectively re-written as if it were simply a relational query.

I chose this position on the issue largely out of a need to be familiar to programmers used to imperative programming languages. However, the particular space explored suggests an alternate philosophy to the traditional one for relational databases. A simulation-programmer might be required to both (a) write code in such a way that the compiler is able to fully abstract it to the relational level and (b) write code in such a way that the compiler can rely on a default query-plan. In particular, the locality constraints enforced through query-loops function in this way to demystify expected performance for simulation-programmers. Such performance concerns are too often abstracted away from programmers writing database queries.

Pat Helland makes the crucial point that because SQL/relational-algebra is built around sets, operations on those sets will tend to be parallelizable. Connecting to the prior discussion, this is because operations on sets tend to instrinsically commute with respect to the order of the data. Sets have no order on the data. In terms from the introduction to this thesis, this is a kind of negative expressiveness of the set abstraction from the client perspective. It reserves the flexibility for the implementation.

A number of specific ideas from databases were also used in this thesis.

Christoph Koch's work[Koc10] on treating incremental view maintenance via algebraic derivatives of relational multi-set algebra (ring of databases) was the basis for my view maintenance approach (§5).

Joe Hellerstein's work on GIST (Generalized Index Search Trees)[HNP95] served as a key source of inspiration for the BVH abstraction in Gong (§3.4.1).

Peter Hawkins' work on relational models for data-structure synthesis[HAF$^+$11] served as a general inspiration for applying relational ideas to more traditional programs.

Finally, conversations with Chris Aberger and insights from his Emptyheaded and Levelheaded systems[ATOR16, ALOR17] helped refine the specific properties of the query-loop. Dylan Hutchison[HHS17] and Fredrik Kjolstad[KKC$^+$17] have also explored the idea of convergences between databases and linear algebra, concurrent to this work.

8.4 Supercomputing Abstractions

One important popular alternative to creating new languages in supercomputing (and elsewhere) is to extend existing popular, low-level langauges (`C`, `C++`, Fortran) via pre-processor directives and other kinds of annotations. OpenMP[DM98] is a good representative of this approach. On the positive side, this strategy allows incremental acceleration of existing code. On the negative side, these directives have no particular safety guarantees, making it easy for a simulation-programmer to introduce subtle race conditions.

Automatic parallelization can be seen as an attempt for compilers to automatically perform transformations based on analyses of the code. This approach is inhibited by even moderately complicated data structure indirections in accessing memory. The philosophy of Galois[PNK$^+$11] can be understood in part as a response to these shortcomings.

However, the main focus of most parallel languages (and language extensions) for supercomputing has been distributed (multi-machine, no shared memory) execution since at least the 90s if not earlier. High-Performance-Fortran[KKZ07] attempted to standardize a variety of data-parallel Fortran extensions. Notable directives include the `FORALL`-loop and `INDEPENDENT` annotation, which function similarly to OpenMP parallel loop directives. The `DISTRIBUTE` and `ALIGN` directives allowed for expressing data partitioning across nodes of a machine. Further work on a similar approach was done in UPC[EGCSY03], focusing on a single global address space across the entire machine.

Later work under the heading of DARPA's High Productivity Computing Systems (HPCS) project attempted to merge some of these distributed and data parallel ideas with software engineering advances in object oriented languages: these include IBM's X10, Cray's Chapel, and Sun's Fortress. Of particular interest, many of these languages leaned on variations of *regions*, a kind of memory-management discipline integrated into typed low-level programming languages like Cyclone[JMG$^+$02].

Regions in this sense can be related to effect-analyses, which must say "where" something is happening.

The Legion Programming System[BTSA12] (whose name is an ammalgram of "Logical rEGION") expanded on the region idea for partitioning data. Notably it was one of a set of task-parallel models which attempt to introduce more flexibility into the *mapping* between compute/memory resources and tasks/data. This is understood as a physical/logical distinction borrowing from the tradition of terminology derived from Codd's arguments about database models. However, it is important to note that the meaning of the distinction in Legion is considerably more low-level. Ebb was developed with the intention of targetting Legion as a parallel runtime. For this reason, a number of ideas from this line of thought were influential if not determinative of the abstractions I describe in this thesis.

While the ZPL[Cha01] programming language is usually understood as being a "super-computing language," (e.g. it was influential on Cray's Chapel) I will discuss it alongside other simulation-specific programming languages instead.

8.5 Graph and Tensor Languages

While only tangentially related, it's worth mentioning a few machine learning and graph-processing frameworks/languages that have likewise attempted to provide some degree of parallel portability via more structured data models.

Graphlab[LGK$^+$14] and Ligra[SB13] provide models where data is stored on a graph and local stencil computations can be written on it. Naiad[MMI$^+$13] (technically a dataflow processing language) showed significant performance improvements over Graphlab and other popular systems for scaling to large datasets simply by paying more attention to lower level performance issues[MIM15]. In the context of this thesis, note that the simulations I compare against are usually more highly tuned; this is a distinct property of simulations remaining the representative application of high-performance-computing.

Graph languages are also notable for supporting interesting models of sparse computation that I do not explore in this thesis. In particular, breadth-first traversals, and other kinds of computations on dynamic subsets of the data have been a subject of particular interest, as has computing on graphs with vertex-degrees that are power-law distributed. A number of practical implementation choices that I made in this thesis will produce unacceptable performance if used to compute on such data.

The recent wave of interest in Deep Learning has largely been enabled by frameworks like Theano[The16], Tensorflow[ABC$^+$16], and PyTorch[PGC$^+$17]. All of these frameworks enable portability of code to execute on GPUs, which has made training on large datasets possible. These systems also benefit programmers by automating the work of computing derivatives. While this is necessary for back-propagation in neural network training algorithms, it also represents an instance of my general argument about the trend of specialization via domain-specific-langauges to enable use of parallel hardware.

In the context of this thesis, these deep learning languages are notable for how they enforce a separation of concerns not just of specialists in parallel hardware and software, but also between specialists who develop training algorithms and users who develop new models.

8.6 Languages in Graphics

According to Alan Kay, we should look at the first computer graphics system (Iverson's Sketchpad) as also the first object-oriented programming language. From this point of view, programming languages have always been a part of graphics systems. However, one of the most important earlier systems to be explicitly acknowledged as a special purpose programming language was Cook's Shade Trees[Coo84]. It was the first shading language, leading to the influential Renderman[HL90] and later on modern shading languages for GPUs. These languages are notable for always being integrated with a larger system or engine for rendering, rather than being intended as a stand-alone language. This architecture was necessary to both ensure high-performance and flexibility to different applications.

Recently graphics languages (most notably Halide [RKAP$^+$12, RKBA$^+$13]) were developed to accelerate image processing. Darkroom [HBD$^+$14] and Rigel [HDD$^+$16] explore languages for image processing hardware specifically. Unlike shading languages, these are not tied to specific host-systems. Instead, they allow for the creation of hardware chips or libraries usable in larger applications. These languages are also notable relative to Ebb as exemplars of *grid stencil languages*.

This brief sketch leaves out many important graphics languages designed for GUIs, printing, 3d printing, diagram description and numerous other tasks. However, the preceding languages are representative of graphics languages that are targeted at accelerating high performance application domains.

From the perspective of this thesis, it is important to note that all of these languages leverage highly domain-specific data models (images, pixels, textures, rays, &c.). Ebb was largely an attempt to generalize beyond these narrower data models while retaining the benefits of abstracting away from low-level random-access-memory models.

8.7 Simulation Libraries and Frameworks

The most common kind of simulation program attempts to simulate a specific phenomenon, potentially with variations on a basic model and on how it is executed. For instance, the VEGA[SSB13] library supports a variety of finite-element discretizations and elasticity models. It does not support plasticity or fluid material behaviors, nor any form of topology change. However, it still supports at least 6 different elasticity models, 2 different kinds of mesh elements, and partial multi-threaded support.

Another approach to reusable simulation code is to try to aggregate simulators for disparate phenomena, and using disparate techniques into a single code-base where some degree of code-sharing can be achieved. Physbam[DHF$^+$11] is exemplary of this model. Multiple PhD theses have been accrued into the codebase, which is used at multiple special-effects houses. However, interoperability between these different library components is not guaranteed, and support for distributed execution, GPU execution, or multi-threading is inconsistent/non-uniform. Code-reuse and interoperability only happens when and where the opportunity to do so is relatively obvious.

Other approaches to simulation engines attempt to reduce all physical phenomena to a single data structure and physics modeling technique. Quite often this is some form of particle-simulation. Nucleus[Sta09] and the work of Macklin et al.[MMCK14] are examples of this approach. Particles in particular suffer from bad geometry for the reconstruction of fluid surfaces. (compared to level-set methods for Eulerian grids) The uniform use of implicit integrators for all phenomena (regardless of the underlying stiffness) is often computationally sub-optimal.

For instance, contact solvers in rigid-body libraries such as Bullet[Cou15] that are used for games tend to specifically rely on projected-Gauss-Seidel solvers because of their qualitative tendency to avoid jitter. These solves must be integrated into positional Verlet-integration schemes in specific ways. Yet if collisions are made elastic, this highly specific choice of techniques becomes less justifiable.

The tension between the need to specialize techniques while achieving code re-use is a principal motivation for the research constituting this thesis.

Lastly, the OpenVDB[MLJ$^+$13] system is of particular interest for two reasons. First, it conforms to the basic assertion that simulation code should be abstracted away from lower level system details by the use of higher-level data abstractions. Specifically, OpenVDB abstracts the management of data fields over sparse-occupancy volumes. Second, this specific kind of geometric domain (hierchical, adaptive grids) lies outside of the data models considered in this thesis. A more complete simulation language would ideally render this kind of data abstraction into interoperable relational terms.

8.8 Languages for Simulation

While many researchers remember Spacetime Constraints[WK88] as an algorithms paper, we can also read it (with a LISP-based system for managing constraints) as one of the earliest simulation programming languages in Computer Graphics. Of particular note is the use of automatic differentiation.

Another odd candidate might be ZPL[Cha01], which is more often discussed as a “parallel programming language.” It is perhaps one of the more important attempts to create a language for stencil-computations on grids. From a data-model perspective, it is related to Halide and other image-processing languages that work on grids.

OP2[MGT$^+$13] and Loci[Luk99, LG05] were idiosyncratic languages built around datalog. They are the earliest instance to my knowledge of recognizing that relational database models could be applied to describing simulations on unstructured meshes. They restricted themselves to static topologies on these meshes. Ebb expands on them by providing more explicit performance guarantees, a more familiar imperative-ish syntax and by bridging the interaction with grid-structured data.

Liszt[DJP$^+$11] was the immediate predecessor to Ebb, which began development as Liszt 2.0. The original Liszt forced all programs to describe data via a single unstrucutred mesh data structure. Ebb's relational model was primarily motivated by the desire to loosen this restriction. The constraints of the PSAAP2 center at Stanford further required support for computing on grids interacting with particles, strongly influencing thinking about data-structure heterogeneity.

Concurrently to Ebb, Fredrik Kjolstad and others developed Simit[KKRK$^+$16]. We published a joint position paper[BK16] outlining some of the differences and common perspectives. Compared to Ebb, Simit focuses only on un-structured data domains via a hyper-graph data model. However, it also includes a linear-algebra sub-language which makes much of the numeric code written in Simit much shorter than the equivalent Ebb code.

8.9 Collision Detection as a Join

The idea that collision detection (and rendering visibility computations) are *spatial joins* is well-known folk wisdom. Evidence for this claim is the regular reference to “collision queries,” “shadow queries,” and the like, as well as the sharing of terminology (e.g. *k*d-tree) between acceleration structures and database indices. Equally sound is the claim that the connection is not obvious, as evidenced by divergent terminology (BVH vs. the variations of R-trees) and the lack of a strong distinction between queries and traversal algorithms. Regardless, I am not aware of any systems prior to Gong which make the connection formally explicit in the system design.

The Flexibile-Collision-Library [PCM12] was developed out of the UNC GAMMA group's research for application in conjunction with Willow Garage's robotics research. It occupies this in-between position of partially recognizing the spatial join perspective. The library provides a fixed set of queries, including continuous-time collision detection, penetration-depth estimation and distance computation. These queries are presented for a fixed set of "collision shapes" (Sphere, Box, Plane, Triangle Mesh, Ray, &c.) similar to Bullet[Cou15] and ODE[Smi06]. As a narrow articulation of these limits, consider that this fixed vocabulary of shapes and queries does not account for Kenny Erleben's taxonomy of 7 distinct ways to compute contact points and normals between different shapes[Erl18]. As a more expansive articulation, arithmetically robust methods[BEB12] for continuous collision detection are excluded, which includes my demonstrated application of Gong to CSG problems.

Besides generalizing the class of queries, Gong provides an important articulation of what effects can be performed in response to collisions—which lie outside of the strict relational-algebraic concept of a query. For instance, while SQL database systems agree on how a query-set is defined, there remain points of divergence over what kinds of "verbs" are allowed to be applied to the sets described by queries. Terminological divergence over the "update-or-insert" kind of operations serves as some evidence of this point.

Gong was especially influenced by ideas from Warren Hunt's thesis[Hun08], which argued for a spectrum of different rendering queries between the traditional Z-buffer and raytracing. By treating the Z-buffer as a special case (eye-rays, single-point-of-projection, uniform distribution) of raytracing, the specific computational implications of each assumption could be teased apart. The observation in Gong's design that `argmin` reductions are equivalent to specific speculative execution properties of Z-buffers follows this mode of thought.

I also drew inspiration from Ivan Sutherland's "A Characterization of Ten Hidden-Surface Algorithms"[SSS74]. In this work, Sutherland, Sproull, and Schumacker claimed that different primary visibility algorithms can be taxonomized by (and hence reduced to) the way they sort the inputs. Notoriously, Ivan Sutherland described this paper as the reason he lost interest in the rendering problem. From the point of view of Gong, I contend that rendering is a spatial-join problem—not a sorting problem. Sorting is just one possible way to compute non-spatial equi-joins, and there is no obvious complexity-theoretic reduction between the two, much less an equivalence.

Once non-worst-case methods of complexity analysis are introduced, the claim becomes even murkier. The further expansion out of the realm of equi-joins and back into a range of different spatial intersection predicates only exacerbates the problem of making the claim more precise. I would suggest we keep the spirit of trying to chart the space of every way to compute these joins, but drop the formally dubious claim that they are "just sorting."

At SIGGRAPH 2018, NVidia unveiled their Turing-architecture GPUs[nvi18], which included a special ray-tracing unit. This unit streams through frustum-bounded packets of rays, which are tested against a triangle-mesh stored in an axis-aligned bounding-box. In the context of Gong, this raises interesting questions about whether collision detection can simply be offloaded to this new piece of hardware. I hope that by charting out the space of collision queries, Gong can help inform further hardware support. For instance rendering queries tend to be characterized by exclusive use of reduction-effects (to use the Gong terminology). Limited "depth complexity" (i.e. the number of intersections along any given ray) also allows for architecting to special assumptions about potential load imbalances that may not be true for more general queries. If some "rays" (or whichever table is streamed) exhibit skew by hitting an outsized number of items on the other side of the join, then a streaming architectures could easily stall and exhibit pathological performance.

Chapter 9

Conclusion

As we run into more limitations in hardware technology scaling, we will necessarily rely on specialization (of algorithms, hardware, programs, &c.) to increase performance of simulation programs. However this same trend towards specialization immediately imposes its own limits by making unreasonable demands on programmers to become super-experts. As I have argued, further progress will depend on the ability to find new ways of organizing simulation code that achieves better separation of concerns between distinct disciplinary specialties.

In this thesis, I have argued that relational data models (which have been primarily of interest in resilient data storage systems) can be fruitfully applied to restructure simulation programs. With Ebb, I showed that structured and unstructured stencil computations can be integrated together via a single uniform and extensible data model. Then, with Seam and Gong I showed that this data model and related database concepts need not be confined to settings with fixed, static topology. Remeshing and Collision Detection computations that change the topology and "structure" of simulation programs also benefit from sharing a common higher-level data abstraction.

As we saw with the different indexing data structures backing query-loops, incorporating topology-changing computations forces compiler-programmers to reconsider how data-structures are managed. Different parts of a simulation program may benefit from different trade-offs in how data is stored.

This situation is analogous to the situation relational database administrators face when choosing how data in a database should be indexed so as to best account for a set of different applications/queries all accessing the same database. Different ratios, patterns, and predictability of reads and writes make different storage strategies more or less attractive. For instance, analytic queries are often run on a snapshot copy of data kept in a transaction-optimized database. Should we expect similar relationships between operations like re-meshing and force-computation/integration in simulation?

9.1 Limitations and Open Problems

One serious shortcoming of Ebb was the rudimentary support for subset representation and management. This was possible because the feature was narrowly needed for managing boundary conditions. However, graph languages[SB13] designed to compute breadth-first search, shortest paths, and/or connected components use more sophisticated encodings of *active sets* to implement algorithms that are otherwise not possible to express in Ebb. Sub-computations like connected-components are used inside systems like Bullet[Cou15] to partition contact solves and more quickly solve close-to-static sub-problems. What implications (if any) does this class of problems have for shared data representations?

The Ebb data model is far from effectively universal. For instance, hierchical grids (adaptive or not) such as those from OpenVDB[MLJ$^+$13] are not supported. On the other hand, I heard concerns from other researchers that Ebb does not support tree-data, like BVHs. While both data-structures are spatial trees, their purposes (discretizing fields on space vs. accelerating joins) are widely divergent. Incorporating other data structures into simulation languages using relational data models requires us to make important distinctions between what kinds of data-structuring decisions are modeling choices, vs. purely algorithmic/computational decisions.

From a very different perspective, symmetric matrices (or anti-symmetric matrices) have no obvious encoding in Ebb. In graph terms, these would correspond to undirected edges. I ran into more elaborate versions of these issues in Seam, when one wants to define triangles or tetrahedra independently of their vertex orderings.

Unless the "orientation" of the triangle or tetrahedron is important, in which case only *even* permutations of the vertices should be allowed. This kind of distinction (which is related to anti-symmetric functions) is rarely treated in a systematically satsifactory way. However, some form of these constructs (at least symmetry) appear to be important for any language focusing on linear-algebraic abstractions, as Simit[KKRK$^+$16] does.

My initial analysis of the need for separation of concerns refers to a potential range of disciplines, including physicists, numerical analysts, and geometers. However, the specific abstractions explored in the thesis have far more to do with the specific separation of parallel systems/computing specialists from these other disciplines in an undifferentiated way. A different set of interesting questions arises from trying to get separation between these disciplines. For instance, how can numerical analysts develop re-usable (non-)linear system solvers and integration algorithms that are not tied to a specific physical or geometric model? The Opt[DMZ$^+$17] language offers some ideas on separating modeling from solving, including use of auto-diff. In the simulation setting, this might allow easier application of variational physics formulations. Another division might be explored by abstracting geometric domains using standard operators such as the Laplacian.

Seam has some specific interesting shortcomings. Removing them could make the language far more applicable to tricky data-structure problems—even outside of simulation problems like remeshing. First, some way to safely represent nullable references in Seam is important for encoding lists, trees, and other traditional data structures. Second, the syntactic constraints of `new` statements prevent certain basic kinds of operations. While working on the Seam paper, I spoke with lab-mates managing circuit IRs (which use graph-like data structures). A simple local operation is to "inline" a module definition by making a copy of some graph and wiring it in, in place of a box. However, it soon became clear that there was no way in Seam as it exists to copy a set of nodes, a set of edges, and correctly wire all the new nodes to the new edges. Each new node must be defined inside of a loop body in a scope where no other new node is defined. As such there is no way to pull together two new nodes in a scope where a new edge can be defined. Perhaps a more careful investigation of how to encode the full range of query-execution strategies would help.

The loose interpretation of effect-checking is also a significant shortcoming of this thesis. On the one hand, it appears that a formalism would be almost trivial. However, that may not be true when the full range of desired effects are taken into account.

Compared to Halide[RKBA+13] and TACO[KKC+17], the languages I describe in this thesis lack sophisticated compilation techniques for generating optimized code. Those projects point the way towards techniques applicable to languages like Ebb. However, important questions remain around the re-use and representation of data which neither project has fully explored.

Lastly, an investigation of how to port these languages to distributed (i.e. multi-process/thread executions without shared memory) settings would go a long way towards reinforcing the performance portability claims in a broader setting.

Much work remains to be done.

Bibliography

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[ABD$^+$15] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. Gpu concurrency: Weak behaviours and programming assumptions. *SIGPLAN Not.*, 50(4):577–591, March 2015.

[ALOR17] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Levelheaded: Making worst-case optimal joins work in the common case. *CoRR*, abs/1708.07859, 2017.

[ATOR16] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446, New York, NY, USA, 2016. ACM.

[BB09] Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM J. Sci. Comput.*, 31(4):2472–2493, June 2009.

[BEB12] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph.*, 31(4):96:1–96:7, July 2012.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966.

[Ber14] Gilbert L. Bernstein. *Cork: Polyhedral Boolean Operation Library.* 2014.

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BK16] Gilbert Louis Bernstein and Fredrik Kjolstad. Perspectives: Why new programming languages for simulation? *ACM Trans. Graph.*, 35(2):20e:1–20e:3, May 2016.

[Ble90] Guy E. Blelloch. *Vector Models for Data-parallel Computing.* MIT Press, Cambridge, MA, USA, 1990.

[BTSA12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[BW13] Gilbert Louis Bernstein and Chris Wojtan. Putting holes in holey geometry: Topology change for arbitrary surfaces. *ACM Trans. Graph.*, 32(4):34:1–34:12, July 2013.

[CBM$^+$08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[Cha96] Bernard Chazelle. The computational geometry impact task force report: An executive summary. In *Selected Papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, FCRC '96/WACG '96, pages 59–65, London, UK, UK, 1996. Springer-Verlag.

[Cha01] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Coo84] Robert L. Cook. Shade trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 223–231, New York, NY, USA, 1984. ACM.

[Cou13] Erwin Coumans. Gpu rigid body simulation using opencl. `http://www.multithreadingandvfx.org/course_notes/GPU_rigidbody_using_OpenCL.pdf`, 2013. Accessed: 2019-04-15; part of 2013 SIGGRAPH Course: "Multi-Threading and VFX".

[Cou15] Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. ACM.

[DBG14] Fang Da, Christopher Batty, and Eitan Grinspun. Multimaterial mesh-based surface tracking. *ACM Trans. Graph.*, 33(4):112:1–112:11, July 2014.

[DHA$^+$13] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

[DHF+11] Pradeep Dubey, Pat Hanrahan, Ronald Fedkiw, Michael Lentine, and Craig Schroeder. Physbam: Physically based simulation. In *ACM SIGGRAPH 2011 Courses*, page 10. ACM, 2011.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[DMZ+17] Zachary DeVito, Michael Mara, Michael Zollöfer, Gilbert Bernstein, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Nießner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Transactions on Graphics 2017 (TOG)*, 2017.

[DRK+14] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of cvc4: How it works, and how to use it. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 4:7–4:7, Austin, TX, 2014. FMCAD Inc.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[EGCSY03] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming.* Wiley-Interscience, New York, NY, USA, 2003.

[Erl18] Kenny Erleben. Methodology for assessing mesh-based contact point methods. *ACM Trans. Graph.*, 37(3):39:1–39:30, July 2018.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[Goo07] Nolan Goodnight. Cuda/opengl fluid simulation. *NVIDIA Corporation*, 2007.

[GT15] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library.* Samurai Media Limited, United Kingdom, 2015.

[HAF$^+$11] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. *Data representation synthesis*, volume 46. ACM, 2011.

[HBD$^+$14] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[HDD$^+$16] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4):85:1–85:11, July 2016.

[Hec12] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[Hel16] Pat Helland. The singular success of sql. *Commun. ACM*, 59(8):38–41, July 2016.

[HHS17] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, pages 2:1–2:10, 2017.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 289–298, New York, NY, USA, 1990. ACM.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[HS05] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems: Fourth Edition*, chapter What Goes Around Comes Around. The MIT Press, 2005.

[Hun08] Warren Andrew Hunt. *Data Structures and Algorithms for Real-Time Ray Tracing*. PhD thesis, The University of Texas at Austin, 2008.

[JMG$^+$02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[Kar12] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37, Goslar Germany, Germany, 2012. Eurographics Association.

[KBC$^+$12] Ian Karlin, Abhinav. Bhatele, Bradford L.. Chamberlain, Jonathan. Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[KBK$^+$13] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[KJM08] Jonathan M. Kaldor, Doug L. James, and Steve Marschner. Simulating knitted cloth at the yarn level. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 65:1–65:9, New York, NY, USA, 2008. ACM.

[KKC$^+$17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[KKRK$^+$16] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph. Presented at SIGGRAPH 2016*, 35(2):20:1–20:21, March 2016.

[KKZ07] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[KLS13] A. Kuzmin, M. Luisier, and O. Schenk. Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 533–544. Springer Berlin Heidelberg, 2013.

[Koc10] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 87–98, New York, NY, USA, 2010. ACM.

[KRBJ12] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54, New York, NY, USA, 2012. ACM.

[LG] Yuan Lin and Vinod Grover. Using cuda warp-level primitives. `https://devblogs.nvidia.com/using-cuda-warp-level-primitives/`.

[LG05] Edward A. Luke and Thomas George. Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis. *J. Funct. Program.*, 15(3):477–502, May 2005.

[LGK$^+$14] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[Luk99] Edward A Luke. Loci: A deductive framework for graph-based algorithms. In *Computing in Object-Oriented Parallel Environments*, pages 142–153. Springer, 1999.

[LUL12] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254, 2012.

[LWS+18] Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug L. James, and Steve Marschner. Interactive design of periodic yarn-level cloth patterns. In *SIGGRAPH Asia 2018 Technical Papers*, SIGGRAPH Asia '18, pages 202:1–202:15, New York, NY, USA, 2018. ACM.

[MGM+11] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[MGT+13] G.R. Mudalige, M.B. Giles, J. Thiyagalingam, I.Z. Reguly, C. Bertolli, P.H.J. Kelly, and A.E. Trefethen. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Computing*, 39(11):669 – 692, 2013.

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[MLJ+13] Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. Openvdb: An open-source data structure and toolkit for high-resolution volumes. In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, pages 19:1–19:1, New York, NY, USA, 2013. ACM.

[MMCK14] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):104, 2014.

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul

Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[Nah11] Vandad Nahavandipoor. *Concurrent Programming in Mac OS X and iOS: Unleash Multicore Performance with Grand Central Dispatch.* O'Reilly Media, Inc., 2011.

[NPO13] Rahul Narain, Tobias Pfaff, and James F. O'Brien. Folding and crumpling adaptive sheets. *ACM Trans. Graph.*, 32(4):51:1–51:8, July 2013.

[NSO12] Rahul Narain, Armin Samii, and James F. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph.*, 31(6):152:1–152:10, November 2012.

[nvi18] Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018.

[Olu11] Kunle Olukotun. Taming heterogeneous parallelism with domain specific languages. `http://archive.dimacs.rutgers.edu/Workshops/Parallel/slides/olukotun.pdf`, 2011.

[PBS$^+$17] Manolis Papadakis, Gilbert Louis Bernstein, Rahul Sharma, Alex Aiken, and Pat Hanrahan. Seam: Provably safe local edits on graphs. *Proc. ACM Program. Lang.*, 1(OOPSLA):78:1–78:29, October 2017.

[PCM12] J. Pan, S. Chitta, and D. Manocha. Fcl: A general purpose library for collision and proximity queries. In *2012 IEEE International Conference on Robotics and Automation*, pages 3859–3866, May 2012.

[PGC$^+$17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[PMTH01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 159–170, New York, NY, USA, 2001. ACM.

[PNdJO14] Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. Adaptive tearing and cracking of thin sheets. *ACM Trans. Graph.*, 33(4):110:1–110:9, July 2014.

[PNK$^+$11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25. ACM, 2011.

[RD97] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, November 1997.

[RKAP$^+$12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

[RKBA$^+$13] 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. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[Rup19] Karl Rupp. 42 years of microprocessor trend data. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`, 2019. Accessed: 2019-04-15.

[SB13] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[She96] Johnathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, SCG '96, pages 141–150, New York, NY, USA, 1996. ACM.

[Smi06] Russell Smith. Open dynamics engine. `http://ode.org/ode-latest-userguide.html`, 2006.

[SSB13] F. S. Sin, D. Schroeder, and J. Barbič. Vega: Non-linear fem deformable object sizmulator. *Computer Graphics Forum*, 32(1):36–48, 2013.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[Sta09] Jos Stam. Nucleus: Towards a unified dynamics solver for computer graphics. In *Computer-Aided Design and Computer Graphics, 2009. CAD/Graphics' 09. 11th IEEE International Conference on*, pages 1–11. IEEE, 2009.

[TBS$^+$16] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 344–358, New York, NY, USA, 2016. ACM.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[UGMW02] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer D. Widom. *Database Systems: The Complete Book*, chapter 7. Prentice Hall, 2002.

[Wad98] Philip Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, November 1998.

[Wil13] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[WK88] Andrew Witkin and Michael Kass. Spacetime constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 159–168, New York, NY, USA, 1988. ACM.

[WTGT09] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Deforming meshes that split and merge. *ACM Trans. Graph.*, 28(3):76:1–76:10, July 2009.

[WTGT10] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Physics-inspired topology changes for thin fluid features. *ACM Trans. Graph.*, 29(4):50:1–50:8, July 2010.

[ZGZJ16] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. Mesh arrangements for solid geometry. *ACM Trans. Graph.*, 35(4):39:1–39:15, July 2016.

[ZJ16] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10, 000 3d-printing models. *CoRR*, abs/1605.04797, 2016.