MASTER'S THESIS

SIMD/MIMD Semantics of a Higher-Order Rank-Polymorphic Programming Language

Author:
Anfisa BOGDANENKO

Supervisor:
Prof. Olin SHIVERS III

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

in the

Khoury College of Computer Science
Northeastern University



August 16, 2026

Northeastern University
Khoury College of
Computer Sciences

MS Thesis Completion

Thesis Title: SIMD/MIMD Semantics of a Higher-Order Rank-Polymorphic Programming Language

Author: Anfisa Bogdanenko

MS Program: Master of Science in Computer Science

The thesis is approved to count toward the student's master's program.

Thesis Advisor (required)	2026/8/16 *Date*
Thesis Reader (required)	8/14/2026 *Date*
Robert Schenck *Thesis Reader (required)*	August 14, 2026 *Date*
Thesis Reader	*Date*
Thesis Reader	*Date*

KHOURY COLLEGE APPROVAL:

Program Director (required)	2026-08-18 *Date*
Program Director (if applicable)	*Date*
Assistant Dean of Master's Programs (required)	2026-08-18 *Date*

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Cailyn Ellard *Recipient's Signature*	8/18/2026 *Date*

Distribution: *Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Khoury College Proquest website - https://www.etdadmin.com/?siteId=282.*

Abstract

REMORA is a higher-order, rank-polymorphic array language whose semantics is parallel by construction: every function application lifts implicitly over the principal frame of its arguments. Like other high-level array languages, REMORA lets programmers express computation without worrying about low-level parallelism decisions, offloading that responsibility to the compiler. What sets it apart is being fully higher-order, which makes the compiler's job considerably harder. One such job is identifying *SIMD fragments*, the stretches of a program that can run in lockstep on parallel hardware. A fragment's extent is bounded by a *MIMD split*, the point at which execution is forced to diverge. Determining which call sites produce such spits is a control flow analysis problem.

This thesis contributes two artifacts on which such an analysis can be built. The first is CPS REMORA, a factored continuation-passing-style intermediate representation. The second, and the principal contribution, is the SIMD/MIMD Machine, an operational semantics for CPS REMORA. The machine is a set of SIMD evaluators, each of which can only execute in lockstep. Because MIMD splits are formalized as explicit machine transitions, the analysis can be derived directly from the presented semantics.

The thesis closes with a discussion of remaining work. It suggests a strategy for abstracting the machine into a statically computable approximation, and considers what the analysis should output to facilitate kernel extraction.

Contents

1 Introduction **1**
- 1.1 Background . . . 2
  - 1.1.1 MIMD, SIMD and Compute Elements . . . 2
  - 1.1.2 Kernels and Where to Find Them . . . 2
  - 1.1.3 REMORA: a Higher-Order Rank-Polymorphic Language . . . 4
- 1.2 Kernel Extraction in REMORA . . . 6
  - 1.2.1 MIMD Splits . . . 6
  - 1.2.2 SIMD/MIMD Analysis . . . 6
  - 1.2.3 The NSAS Framework . . . 7
  - 1.2.4 The Need For an Intermediate Representation . . . 8
- 1.3 Scope and Contributions . . . 9
  - 1.3.1 Roadmap . . . 9

2 CPS REMORA **10**
- 2.1 Notational Conventions . . . 10
- 2.2 CPS REMORA Syntax . . . 11
  - 2.2.1 Core CPS REMORA . . . 11
  - 2.2.2 REMORA Programs in CPS REMORA . . . 12
  - 2.2.3 Syntactic Sugar . . . 13
- 2.3 Assumptions and Limitations . . . 15
  - 2.3.1 Programs are Alphatized and Typechecked . . . 15
  - 2.3.2 No Higher-Order Primitives . . . 15
  - 2.3.3 Nonstandard Beta-Reduction . . . 15
- 2.4 Examples . . . 16
  - 2.4.1 Example 1: Lifting . . . 16
  - 2.4.2 Example 2: Parallel Binding and Continuation . . . 17
  - 2.4.3 Example 3: MIMD Split on If . . . 18

3 SIMD/MIMD Machine **19**
- 3.1 Notational Conventions II . . . 20
  - 3.1.1 Set Combinators and Tuples . . . 20
  - 3.1.2 Sequences . . . 20
  - 3.1.3 Sets . . . 21
  - 3.1.4 Maps . . . 21
- 3.2 Semantic Domains . . . 22
  - 3.2.1 Dimensions and Shapes . . . 22
  - 3.2.2 Indices and Domains . . . 22
  - 3.2.3 Index Maps . . . 23
  - 3.2.4 Collection Dimensions . . . 25
  - 3.2.5 Collections . . . 25
  - 3.2.6 Environments . . . 28
  - 3.2.7 Machine Values . . . 29
  - 3.2.8 Continuation Values . . . 30
- 3.3 SIMD/MIMD Machine . . . 31
  - 3.3.1 The Big Idea . . . 31
  - 3.3.2 Core Execution State . . . 33

3.3.3 Parallel Context and Domain . . . 34
3.3.4 Core Identity . . . 35
3.3.5 Core and How It Steps . . . 35
3.3.6 Machine and How It Steps . . . 37

4 Core Transitions **40**
4.1 Eval: Syntax to Semantic Values . . . 41
4.1.1 Transition Rule . . . 41
4.1.2 Syntax Evaluators . . . 42
4.2 Lift: Implicitly Replicating Data . . . 43
4.2.1 Transition Rule . . . 43
4.2.2 Lifting . . . 44
4.3 Split?: Deciding Between SIMD and MIMD . . . 48
4.3.1 Transition Rules . . . 48
4.3.2 Identifying MIMD Splits . . . 49
4.3.3 Initializing Child Cores . . . 51
4.4 Apply: Executing in SIMD Mode . . . 52
4.4.1 Transition Rules . . . 52
4.4.2 Executing Primops Inside Compute Elements . . . 53
4.4.3 Assembling Cells Into Collections . . . 55
4.5 CApply: Invoking Continuations . . . 56
4.5.1 Transition Rules . . . 56
4.5.2 Adjusting Frame Parallelism . . . 58
4.6 Wait: Monitoring Child Cores . . . 59
4.6.1 Transition Rules . . . 59
4.6.2 Talking to The Machine . . . 60
4.6.3 Assembling MIMD Results . . . 60
4.7 Done: Delivering Results . . . 61
4.7.1 Transition Rules . . . 61

5 Implementation Overview **62**
5.1 Execution Traces . . . 63

6 Further Work **64**
6.1 Higher-Order Primitives . . . 65
6.1.1 Delayed MIMD Splits . . . 65
6.1.2 Reducing Along the Main Axis . . . 66
6.1.3 Accumulating Data in Continuation Values . . . 67
6.2 SIMD/MIMD Analysis Output IR . . . 68
6.2.1 SIMD Call Graph Example . . . 68
6.2.2 SIMD Fragments . . . 69
6.3 Abstracting the SIMD/MIMD Machine . . . 70
6.3.1 Sound Approximations in SIMD/MIMD Analysis . . . 70
6.3.2 Simple Abstraction Is Good! . . . 70
6.3.3 Abstraction Targets . . . 71
6.4 Soundness Proofs . . . 72

7 Conclusion **73**

Chapter 1

Introduction

REMORA is a higher-order, rank-polymorphic array language whose semantics is parallel by construction: every function application lifts implicitly over the principal frame of its arguments. Like other high-level array languages, REMORA lets programmers express computation without worrying about low-level parallelism decisions, offloading that responsibility to the compiler. What sets it apart is being fully higher-order, which makes the compiler's job considerably harder—and more interesting for us.

One such job is *kernel extraction*, which seeks to identify program fragments that can run in lockstep on parallel hardware. This chapter explains what makes extraction difficult in REMORA, sketches out the static analysis proposed as a solution, and, importantly, identifies the scope of the work that this thesis carries out.

This chapter is organized as follows:

- Section 1.1 introduces parallel hardware, the SIMD and MIMD execution models, and existing approaches to programming for them;
- Section 1.2 connects kernel extraction to semantic MIMD splits in REMORA, frames their identification as a control flow problem, introduces the NSAS framework the project follows, and makes the case for a continuation-passing-style IR;
- Section 1.3 scopes this thesis against the overarching goal of kernel extraction and gives the roadmap for the remaining chapters.

1.1 Background

We live in a world of ever-increasing computational demands. Scientific computing, graphics, blockchain, finance, and most recently AI and machine learning all push for faster and more efficient computation. For a long time these demands could be met simply by waiting for the next generation of hardware, but ever since mid-2000s the slowing advancements in single-threaded performance could no longer keep up [8]. Instead, we now heavily rely on *parallelism*.

Much of computationally intensive work amounts to performing the same operations over massive data, which is naturally parallelizable. With specialized hardware, tens of thousands of computations can be performed simultaneously, which significantly increases the throughput and therefore the overall speed. As a result, nearly every modern device comes equipped with a GPU or some other parallel accelerator.

Despite the abundance of hardware, however, harnessing its power *well* proves to be difficult. Good utilization often means writing explicit, low-level code against a specific architecture, which in turn demands a lot of domain knowledge. This section gives an overview of various approaches to this issue, including the one favored by this thesis.

1.1.1 MIMD, SIMD and Compute Elements

Parallel execution is traditionally classified by how many instruction streams run over how many streams of data [5, 8]. Here, we are interested in two classes. Under *multiple-instruction multiple-data* (MIMD), independent instruction streams run over independent pieces of data. This describes, for example, a multicore CPU running different tasks on different cores. Under *single-instruction multiple-data* (SIMD), one instruction stream runs over many independent pieces of data. This is the computational model that parallel hardware is built to exploit.

A *compute element* (CE) is one component of a SIMD computation that has its own data but shares the instruction stream with the other CEs. Under SIMD, compute elements always stay in lockstep. In a GPU, they correspond to threads, though the exact terminology varies by architecture.

Throughout this thesis, SIMD and MIMD are claims about a *program*: whether a stretch of computation can proceed under one instruction stream, or whether it requires several.

1.1.2 Kernels and Where to Find Them

The massive throughput of parallel hardware is largely achieved by relying on SIMD execution, but SIMD breaks easily. Something as simple as a conditional would force the CEs to diverge, so how does one run a program that is not all SIMD?

The hardware itself absorbs a good deal of this, since it is not strictly SIMD to begin with. For example, NVIDIA describes its GPUs computational model as *single-instruction multiple-thread* (SIMT): threads in a warp share an instruction stream but may still take different branches [14, 16]. It handles divergent branches by executing each path sequentially while keeping the CEs not on that path switched off. This strategy decreases the overall throughput, so it is only worthwhile for minor divergences.

Sizeable divergence is a different matter. When a program genuinely calls for several instruction streams, the work has to be subdivided into separate *kernels*: units of work

that are assigned to parallel hardware. The kernels can then run in parallel, perhaps on several devices at once, while the divergent work and kernel launching are left to the CPU.

Programming parallel hardware well, then, amounts to finding the largest kernels and minimizing data communication costs. The question is who is responsible for this orchestration, and at what level of abstraction.

Architecture-specific kernel APIs

At the lowest (but still human-readable) end of abstraction, kernels are expressed in architecture-specific, mostly imperative APIs. Prominent examples are the CUDA platform for NVIDIA GPUs [15] and the OpenCL open standard for heterogeneous computing [26]. Code written in these APIs makes all the low-level decisions explicit: memory management, data layout, synchronization and, of course, the boundaries of the kernels themselves. Kernels are distinct syntactic units that rely on CPU code to launch and orchestrate them, so the language clearly separates code regions based on what computational model is expected to run them.

The programmer gets a lot of control in exchange for the mental burden of every low-level parallelism decision, and the resulting programs are verbose and error-prone. For demanding applications it is appropriate to assign parallel programming experts to write such code, but in most cases the complexity is prohibitive. We tend to think of these architecture-specific APIs as compilation targets for more developer-friendly tooling.

SPMD compilers

Sitting slightly above the kernel APIs is ISPC, a compiler for *single-program multiple-data* (SPMD) code, which it compiles to CPU vector units [18] and, in more recent versions, to Intel GPUs as well [10]. An ISPC program is written from the perspective of a single *program instance*, and the compiler maps a *gang* of instances onto SIMD lanes, handling lane divergence on the programmer's behalf. SPMD is a weaker claim than that of SIMD: instances share program text rather than an instruction stream, and the compiler is free to let them diverge and (possibly) reconverge. This removes much of the burden of writing explicitly vectorized code, but the kernel boundary is still drawn by the programmer.

ISPC also offers a precedent for the central idea of this thesis. A function pointer in it may be declared `varying`, meaning that different program instances may hold different ones. When handling a call through such a pointer, ISPC finds the distinct callees among the active instances and invokes each one with only the instances that held it [10]. This thesis will come to think about MIMD divergences with a similar mechanism (Sections 3.3.1 and 4.3.2).

Array languages

At a higher level of abstraction sit array languages, where programs are written functionally in terms of array combinators, leaving a lot of the low-level parallelism decisions to the compiler. Under imperative models the compiler has to *discover* the implicit parallelism, while the array model describes aggregate operations over sets of data,

so parallelism structure and data dependencies are encoded directly in the semantics [12]. Since the compiler does not have to work as hard to find them, it can make good optimization and codegen decisions, and programmers get good performance without much of the mental burden. As an added benefit, array language code is much terser and cleaner than kernel API code.

Many array languages in use today do not try to be general purpose. Instead, they offer to optimize the compute-intensive parts of an application that can be easily expressed within the array model. Examples include DSLs like PyTorch [17] and JAX [6] as well as full languages like Futhark [9]. These rely on a host language to orchestrate the overall system and on the programmer to decide the boundaries of compiled kernels.

Compiler-directed array languages

Other array languages, however, are able derive kernel boundaries themselves. For example, SaC parallelizes one architecture-agnostic source across multi-core CPUs, CUDA devices, and heterogeneous combinations of the two [7]. NESL supports *nested* data parallelism, where a parallel computation may itself be mapped over an array. A global *flattening* transformation rewrites such a program into data-parallel code suitable for SIMD execution [2, 1]. The Nessie compiler carries this model to the GPU, laying out kernels and fusing them in its intermediate representation [20].

Heterogeneous orchestration is therefore within reach of a high-level array language, but all these tools come with a notable limitation: they are first-order. Since functions do not flow through the program like values, which computation runs over which data is fixed by the program text. Kernel boundaries then become a structural property.

Higher-order array languages

Now suppose an array language were higher-order, with functions as ordinary values that a program can compute and then apply. Such a language could express general-purpose computation, divergence included, without forgoing the benefits of array programming. The compiler gains the freedom to extract kernels, optimize across them, and schedule for the system as a whole. The programmer, in turn, gets a powerful parallel programming tool.

1.1.3 Remora: a Higher-Order Rank-Polymorphic Language

Remora is such a language. It is a higher-order rank-polymorphic array language with a static type system that tracks the shapes of arrays [24, 25, 23]. This thesis assumes familiarity with core concepts of the language [21].

At the core of Remora's computational model is *rank polymorphism*: every function application is replicated implicitly over the principal frame of its arguments, so parallelism is entirely data-driven. The idea comes from Iverson's APL [11], which was later refined by its successor J [13]. Both these languages are general-purpose, but neither was built with a compiler for modern parallel hardware in mind: they are dynamically typed, they lack a formal semantics to ground a compiler analysis, and they are not fully higher-order. Remora aims to address these.

That last point is what gives the language its account of divergence. REMORA's formal dynamic semantics generalizes the array computational model from SIMD to MIMD precisely by allowing functions themselves to be lifted to arrays [24], so divergent computations are expressed as MIMD computations.

This view of divergence is what makes REMORA suitable for the compilation strategy described above, and also what makes it difficult. Since MIMD computation is expressible in the object language as an ordinary value, execution can diverge wherever that value is called, and nothing in the syntax indicates it. The notion of divergence is now *semantic* rather than syntactic, and identifying these divergences with a static analysis is the central goal of this project.

1.2 Kernel Extraction in REMORA

For REMORA to deliver on its promises, its compiler has to identify kernels that can be scheduled to run on parallel hardware. The bigger the kernels it finds, the better the overall throughput. We call this task *kernel extraction*.

Every language (or its IR) in Section 1.1.2 has a syntactic notion of where one kernel ends and the next begins, but higher-order REMORA does not. Instead, kernel extraction has to start somewhere else: with the question of which stretches of a program run in lockstep. We call such a stretch a *SIMD fragment*.

Fragments and kernels are related but distinct. A fragment is a property of the program, that is, some region of the computation that needs only one instruction stream. On the other hand, a kernel is a decision about scheduling, that is, some region worth launching on its own given available hardware. They need not line up. A fragment too small to justify a launch is better fused with its neighbors, and a small enough divergence can be left inside a kernel rather than cutting one kernel into three.

Before kernels can be formed, SIMD fragments must be identified, which is the central motivation for this thesis. How the identified fragments can then be used to extract kernels is briefly revisited in Section 6.2.

1.2.1 MIMD Splits

SIMD fragments are bounded what we call a *MIMD split*. It is the point in execution at which the compute elements can no longer stay in lockstep and previously SIMD computation is forced to diverge. Identifying fragments therefore amounts to locating splits, so it is worth highlighting how one arises in the first place.

Being higher-order separates REMORA from the array languages it descends from. Here, every value is an array, and an application expects a value in the function position. An array of *different* functions can then easily appear in the function position, which is the most straightforward way MIMD computation can be expressed. Here is an example:

$$([+\ -]\ \ [2\,3]\ \ [4\,5]) \quad \text{computes} \quad [(+\ 2\ 4)\ \ (-\ 3\ 5)] \ = \ [6\ \ \text{-}2]$$

When the array in the function position is applied, it launches two computations that diverge: the first one adds while the other subtracts. Note that the divergence follows from the ordinary semantics of rank polymorphism in REMORA and not any special MIMD construct. In operational terms, then, a MIMD split is a call site at which more than one distinct procedure is applied at once.

While the MIMD split is syntactically obvious in this example, it is not always so. The higher-order nature of REMORA allows us to write the value in the function position as a variable reference. In this scenario, the only way to decide whether the call site results in a MIMD split is to run a code analysis.

1.2.2 SIMD/MIMD Analysis

What the compiler needs, then, is a static *SIMD/MIMD analysis*: a procedure that determines which parts of a REMORA program can be executed in SIMD mode and which require MIMD. To be a viable part of the compilation pipeline it has to identify MIMD

splits without ever running the program, and use that information downstream to extract kernels. The standard way to compute such an answer is abstract interpretation, which approximates concrete program behavior [3]. The approximation will be imperfect, but with good abstraction it is close enough to decide the property of interest.

Such an analysis is a large undertaking; the role of this thesis is to lay the necessary semantic groundwork for it.

SIMD/MIMD analysis as a control flow problem

The MIMD split example from the beginning of the section already hints at the kind of problem that SIMD/MIMD analysis is. Since REMORA is higher-order, procedure arrays can flow to a call site through a variable reference, so the set of procedures that reaches a certain call site cannot be determined syntactically. This requires a *control flow analysis* (CFA).

This work follows in the footsteps of traditional control flow analysis for lambda-based languages, developed by Shivers [22]. Applied to higher-order programs, that approach yields substantial results, as 3CPS demonstrates [19].

The array-as-a-value view adds a twist, however. A traditional CFA asks which procedures can reach a call site, and the answer is a set of procedures. With REMORA, we introduce an extra level of structure: what arrives at a call site is not a single procedure but an array. The answer is therefore a *set of sets* of procedures. Proper SIMD/MIMD analysis must be able to distinguish (1) a call site that applies only single-procedure arrays, even if the procedure differs on each call, from (2) a call site that applies one array holding several distinct procedures. The former maintains SIMD every time, while the latter identifies a MIMD split.

1.2.3 The NSAS Framework

This project follows the framework laid out in Shivers' dissertation [22], called the *Non-Standard Abstract Semantics* (NSAS) framework. It builds a static analysis out of a language's semantics in three stages:

1. a *standard* semantics, which gives a precise account of what programs in the language mean;

2. a *nonstandard* semantics, which still describes concrete execution but is instrumented to also accumulate whatever information the analysis is after. It can then produce a "perfect" answer for the analysis, but it is not generally computable at compile time;

3. an *abstract* semantics, which collapses the parts of the nonstandard semantics that can grow without bound. It gives up some exactness in exchange for being statically computable.

Running a program under abstract semantics is what we call abstract interpretation. Where Shivers works denotationally, this thesis takes the standard semantics to be an abstract machine, which abstracts into a sound analysis of itself by bounding infinite parts of its state space [27]. Section 6.3 discusses in more detail how this applies to REMORA's SIMD/MIMD analysis, but it is worth highlighting what is considered a safe

approximation. When the collapsed states are unsure whether a certain call site results in a MIMD split, the safe answer is to always pick MIMD. This is because we can think of SIMD computations as trivial MIMD computations that only contain one procedure. The inverse is, of course, not true.

Which standard semantics is appropriate for NSAS?

The first stage calls for a standard semantics, and REMORA already has one [23], but deriving the SIMD/MIMD analysis directly from it is not viable because it is too abstract to observe and record MIMD splits the way the analysis demands. What is required instead is a semantics that makes the property the analysis cares about an explicit part of the operational state. At the very least, it must have a formal notion of a MIMD split.

1.2.4 The Need For an Intermediate Representation

When dealing with complex program analysis, it is ordinary compiler practice to define an *intermediate representation* (IR): a simpler, lower-level form of the surface language, backed by a meaning-preserving conversion. An IR whittles the operators and syntactic forms of the surface language down to a smaller, more primitive set. The result is harder for a human to read, but far easier to reason about algorithmically.

The SIMD/MIMD analysis is a higher-order control flow problem, and the traditional choice in this case is a *continuation-passing style* (CPS) IR [22]. This IR choice makes all transfers of control into explicit calls that the semantics has to handle, which is exactly what we need from our NSAS standard semantics.

This thesis defines such an IR for REMORA, called CPS REMORA. Chapter 2 presents it in full.

1.3 Scope and Contributions

Of the three NSAS stages described in the previous section, this thesis takes on the first. It defines a standard semantics that makes explicit where and how MIMD splits happen during execution. We call it the *SIMD/MIMD Machine*. The second and third stages are considered as well, and they inform many of the design decisions behind the standard semantics. The thesis concludes by laying out the expected further work.

Robert Schenck, PhD, collaborated on the first draft of the mathematical model presented here, which has since been substantially revised. All writing in this document is my own.

1.3.1 Roadmap

The thesis makes the following contributions, organized by chapter.

Chapter 2 presents CPS REMORA, a factored continuation-passing-style IR for a rank-polymorphic language. Section 2.2 gives its grammar and Section 2.3 states the assumptions about CPS REMORA programs that the rest of the thesis relies on. Section 2.4 works through a set of examples, building intuition for frames, cells, and lifting in a CPS setting.

Chapters 3 and 4 define the SIMD/MIMD Machine, an operational semantics for CPS REMORA, which is the primary contribution of this work. In this semantics, the shape of parallelism, the compute elements available at any given point, and the exact location of a MIMD split are all explicit. Most importantly, it gives an operational account of what MIMD actually means in a rank-polymorphic language. The machine is best understood as a collection of SIMD evaluators, which we call *cores*: a core step is SIMD computation, while a machine step advances an arbitrary subset of cores at once and is therefore MIMD. The semantics is also parameterized over a core scheduler function, which is how the machine models execution under bounded compute resources.

The formalization is split across two chapters. Chapter 3 establishes notation (Section 3.1) and semantic domains (Section 3.2), then presents the machine state, its transitions, and its initial and final states (Section 3.3). It also defines core state, but the individual core transitions are formalized in Chapter 4. Sections 4.1 to 4.7 give the rules for core transitions in each of the seven execution states.

Chapter 5 gives an overview of the Haskell implementation of the SIMD/MIMD Machine. A notable feature is execution trace printing, which makes it easy to examine how the machine steps through a given CPS REMORA program.

Chapter 6 surveys what remains to be done before kernel extraction from REMORA programs is possible. Section 6.1 describes how the SIMD/MIMD Machine can be extended to support higher-order primitives, including a new source of divergence that they would introduce. Section 6.2 considers what a useful SIMD/MIMD analysis output should look like. Section 6.3 outlines which parts of the semantics an abstraction needs to collapse to become computable. Section 6.4 identifies the proof obligations that will ensure the soundness of the SIMD/MIMD analysis.

Chapter 2

CPS REMORA

This chapter introduces CPS REMORA, a continuation-passing-style intermediate representation (IR) well-suited for the SIMD/MIMD analysis. Its formal semantics is given in the following chapter.

This chapter is organized as follows:

- Section 2.1 introduces the definitional conventions for this thesis;
- Section 2.2 gives CPS REMORA's core grammar, syntactic sugar, and how a full REMORA program is represented as a single top-level term;
- Section 2.3 states the syntactic and type-soundness assumptions the rest of this thesis relies on, along with the limitations of the current model;
- Section 2.4 works through a few small CPS REMORA programs, building intuition for frames, cells, lifting, and MIMD splits.

2.1 Notational Conventions

Throughout this thesis, mathematical objects get introduced in one of the following ways:

- Definitional assignment, written with $:=$, is used to give a name and a metavariable to a set that can be easily constructed by combining previously defined sets. For example, $d \in \mathit{Dim} := \mathbb{N}$ says a dimension just is a natural number.
- Grammar, written with $::=$, instead defines a set by cases, listing every syntactic variant that can be used to build one of its elements. For example, Figure 2.1 in the following section defines the core grammar of CPS REMORA this way.
- Functions are introduced by a type signature, $f : A \to B$, followed by one or more defining equations for f, often given by cases on the shape of its argument.

See also Section 3.1 for more notational conventions that are relevant specifically to the operational semantics of Chapter 3.

2.2 CPS REMORA Syntax

CPS REMORA is the intermediate representation (IR) that the SIMD/MIMD Machine executes. It is a continuation-passing-style (CPS) IR, which views every transfer of control as an explicit one-way function call. Figure 2.1 gives the core grammar of the IR.

2.2.1 Core CPS REMORA

CPS REMORA (CORE)

$b \in \mathit{Basic}$	$::=$	$z \mid \texttt{\#t} \mid \texttt{\#f} \mid \cdots$	basic values ($z \in \mathbb{Z}$)
$r \in \mathit{Rank}$	$::=$	$n \mid \texttt{all}$	arg. rank ($n \in \mathbb{N}$)
$l \in \mathit{Lab}$			labels
$x \in \mathit{Var}$			variables
$x_k \in \mathit{CVar}$			cont. variables
$\mathit{lam} \in \mathit{Lam}$	$::=$	$(\texttt{fn}_l\ [(x\ r)\ldots]\ x_k\ .\ \mathit{call})$	lambda
$\mathit{prim} \in \mathit{Prim}$	$::=$	$+ \mid - \mid \times \mid \texttt{id} \mid \texttt{select} \mid \texttt{zero?} \mid \cdots$	primop
$a \in \mathit{Atom}$	$::=$	$b \mid \mathit{lam} \mid \mathit{prim}$	array atoms
$f, e \in \mathit{Exp}$	$::=$	$x \mid (\texttt{array}\ (n\ldots)\ a\ldots)$	expression
$\mathit{clam} \in \mathit{CLam}$	$::=$	$(\texttt{cont}_l\ x\ .\ \mathit{call})$	cont. lambda
$k \in \mathit{CExp}$	$::=$	$x_k \mid \mathit{clam}$	cont. expression
$\mathit{call} \in \mathit{Call}$	$::=$	$(f\ e\ldots\ k)$	call

FIGURE 2.1: Core syntax for CPS REMORA

A CPS REMORA program consists of a series of nested *function calls*, each built from a function expression, an n-ary sequence of argument expressions, and a continuation expression. In CPS, the meaning of a function call is to apply the function expression to the arguments and then invoke the continuation expression with the result.

CPS REMORA is a *factored* CPS, meaning that we separate standard *"user" expressions* and *continuation expressions* into two disjoint syntactic sets [4]. Additionally, the variables that bind them are separated into *Var* and *CVar*, respectively. As a result, a lambda parameter can never be confused for a continuation parameter.

"User" expressions are either "user" variable references or array literals. An *array literal* consists of a sequence of natural numbers, representing the shape of the array, and a sequence of atoms listed in row-major order. An *array atom* can be either a basic value (here, integers or booleans), a "user" lambda, or a primitive operation. The sets of basic values and primitives can easily be extended, although the presented semantics comes with certain limitations (Section 2.3). Continuation expressions are either continuation variable references or continuation lambdas.

Both *"user" lambdas* and *continuation lambdas* consist of binding parameters, a unique syntactic label l drawn from *Lab*, and a *call* body. A standard lambda *lam* binds a possibly empty sequence of parameters $(x\ r)$ and a single continuation parameter x_k, which is the continuation for that lambda. A continuation lambda *clam* binds a single parameter x, the "return" value for that call.

Each parameter in a standard lambda $(x\ r)$ is annotated with rank r. An *argument rank* is a natural number representing the expected number of dimensions for that parameter. If a lambda application tries to bind an array of a higher rank than the corresponding parameter expects, the extra dimensions become the source of parallelism for this application in a process known as *lifting*. The continuation parameter x_{k} does not have an expected rank annotation because continuation expressions are not arrays.

Notice also that a continuation lambda's only parameter x does not have an associated expected rank. This is because, in direct-style REMORA, lifting only takes place on function application and there is no way to express expected rank for a return site. For generality, however, one can think of x as always expecting rank `all` for the return value.

On continuation primitives

In traditional factored CPS, continuation expressions include continuation primitives and, in particular, a `halt` primitive that stops program evaluation with a result. While the grammar of CPS REMORA does not support this construct at the syntactic level, it still has a notion of continuation primitives as semantic values.

This design decision stems from the complex evaluation state that CPS REMORA semantics has to navigate (Section 3.3). Whenever a program executes a MIMD computation, it will have multiple active continuations. The proposed semantics addresses this by operating multiple independent SIMD evaluators, each with its own continuation. In this setup, the traditional `halt` does not quite make sense. Still, there is a need for some continuation primitive to support MIMD splits. These primitives need to be aware of the surrounding evaluation state. Since the surface syntax does not encode the multiple-evaluator structure that arises during execution, these dynamically generated continuation primitives are not expressible as syntax.

See Section 3.2.8 for the description of the continuation primitive that the proposed semantics relies on.

2.2.2 REMORA Programs in CPS REMORA

Given a full, direct-style REMORA program, a CPS converter will output a CPS REMORA top-level thunk $lam_{\texttt{TOP}}$, as defined below.

$$lam_{\texttt{TOP}} := (\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ .\ call_{\texttt{TOP}})$$

FIGURE 2.2: CPS REMORA entrypoint.

Here, $call_{\texttt{TOP}}$ is the CPS-converted call body representing the entire program and $k_{\texttt{TOP}}$ is a top-level continuation parameter. To compute the result of the converted program, the thunk expression is called with no arguments and a semantic continuation value $\kappa_{\texttt{HALT}}$ (Section 3.3.6).

2.2.3 Syntactic Sugar

The core grammar in Figure 2.1 is deliberately small, but it can be inconvenient when writing example programs. The following syntax extensions are macro-expandable, but they significantly improve the readability of CPS REMORA programs.

CPS REMORA (SUGAR) extends CPS REMORA (CORE)

$$
\begin{array}{rcll}
\mathit{call} \in \mathit{Call} & ::= & \cdots & \\
& | & (\texttt{ret}\ k\ e) & \text{return} \\
\\
a \in \mathit{Atom} & ::= & \cdots & \\
& | & (\texttt{if}\ x\ x_{\mathrm{k}}\ \mathit{call}_{\mathrm{t}}\ \mathit{call}_{\mathrm{e}}) & \text{if-then-else} \\
\\
f, e \in \mathit{Exp} & ::= & \cdots & \\
& | & \mathit{narr} & \text{nested array} \\
\mathit{narr} \in \mathit{NestedArr} & ::= & a \mid [\mathit{narr} \ldots] &
\end{array}
$$

FIGURE 2.3: Sugar for CPS REMORA

Figure 2.4 gives the macro expansions for each of these forms.

A continuation expression can be called directly with $(\texttt{ret}\ k\ e)$, which means "return to continuation k with value e." An array literal can be written as a nested bracketed array, where the base case is just a scalar atom. This is a much more natural notation since the shape is implicit. Nested arrays must always be regular, so [[1 2] [3]] is not valid. Finally, and perhaps most surprisingly, we can introduce conditional `if` simply as syntactic sugar.

Notice that the given `if` form differs from a typical CPS conditional. First of all, it is not an expression but an array atom. Since all expressions in REMORA must be arrays, it follows that even conditional splits might end up running in parallel. Another notable detail is that the `if` form includes a continuation parameter x_{k}. This is because in a parallel setting, a conditional will always act like a synchronization barrier. After the branches diverge, we want them to get back in sync before the SIMD computation can continue. Invoking a conditional `if`, then, looks exactly like calling a lambda that only binds a continuation parameter, which will be the *join point* of a branch split. To get a specific example of how this conditional operates, see Section 2.4.3.

Desugaring

Non-Terminal	**Syntax**	**Desugars Into**
Call	(`ret` k e)	(`id` e k)
Exp	*narr*	flattened shape & atoms, below
Atom	(`if` x_{grd} k_{jp} $call_{\text{t}}$ $call_{\text{e}}$)	(`fn`$_{\text{if}}$ [] k_{jp} . (`select` x_{grd} (`fn`$_{\text{t}}$ [] k_{t} . $call_{\text{t}}$) (`fn`$_{\text{e}}$ [] k_{e} . $call_{\text{e}}$) (`cont`$_{\text{if}}$ x_{res} . (x_{res} k_{jp}))))

FIGURE 2.4: Desugaring patterns.

The first is a direct translation. The nested array case and the `if` case are both worth elaborating on.

A nested array literal is built out of *narr* ∈ *NestedArr* (Figure 2.3), a helper nonterminal that is either a scalar atom or a bracketed sequence of nested arrays. Desugaring a nested array form just means constructing an equivalent array literal form. The atoms in the nested array structure already appear in row-major order, exactly as the `array` form expects. A few examples make the pattern clear:

1	desugars to	(`array` () 1)
[1 2 3]	desugars to	(`array` (3) 1 2 3)
[[1 2 3] [4 5 6]]	desugars to	(`array` (2 3) 1 2 3 4 5 6)

As stated above, `if` acts like a lambda that takes only a continuation k_{jp}. Each of the branch calls gets wrapped into a zero-argument thunk, labeled "t" for "then-branch" and "e" for "else-branch". Both thunks, along with the guard variable x_{grd}, get passed to `select`, which is a core conditional primitive that expects scalars for each of its arguments. If the x_{grd} variable is bound to a non-scalar array, then the application will get lifted to match its shape and the overall result will be an array of branch thunks. That result is then bound to x_{res} and immediately called with no arguments and the join point continuation k_{jp}.

2.3 Assumptions and Limitations

2.3.1 Programs are Alphatized and Typechecked

We assume that every CPS REMORA program is *alphatized*, that is, each variable name is unique and therefore bound by exactly one lambda. We also assume that every program is *closed*, so no variable reference will ever be unbound.

Finally, we assume that every CPS REMORA program is a type-erased, CPS-converted REMORA program that *typechecks* [23]. Given REMORA's type soundness, this rules out shape-mismatch errors, ill-typed primitive arguments, and primitive arity errors. As a result, the only kind of error that might arise during execution of a CPS REMORA program is a runtime primop error, such as a divide-by-zero.

Runtime Errors

The primitives given in Figure 2.1 intentionally avoid any runtime errors, so that the operational semantics presented in the following chapter never needs to encode an error state. Formalizing the notion of stuck states will be necessary for the formal proofs discussed in Section 6.4. Additionally, this concrete semantics is only a stepping stone toward an abstraction that will not consider runtime errors at all, so their absence here is not a big concern.

2.3.2 No Higher-Order Primitives

Every defined primitive is first order: none of them takes a function as an argument, the way `reduce` or `scan` would. This is because the presented operational semantics does not model higher-order primitives to keep the discussion focused on the rank-polymorphic machinery. Extending the model to support such primitives is relatively straightforward, but not trivial; Section 6.1 demonstrates how.

2.3.3 Nonstandard Beta-Reduction

Ordinarily, one of the benefits of using a CPS IR is that beta-reduction is immediately available as a sound, purely syntactic rewrite. That, unfortunately, does not hold for CPS REMORA.

As will become clear in the following chapter, continuation values carry more than the syntax encodes. In particular, when a continuation lambda is evaluated, the current parallel context gets captured as a part of the value. This is because to properly resume the continuation, it needs to reestablish the surrounding parallel context, which is a dynamic property. Direct syntactic substitution breaks this control structure, since now the same expression appears in a different surrounding context. Beta-reducing an ordinary expression has a related problem: naive substitution bypasses lifting, and that breaks the semantics of rank polymorphism.

Presently, a collaborator on the REMORA project is developing a calculus for direct-style REMORA, which would formalize a beta-reduction for non-continuation expressions. This work is the necessary foundation for supporting beta-reductions in CPS REMORA as well.

2.4 Examples

This section walks through a few CPS REMORA programs, giving an intuition for its semantics. The discussion assumes familiarity with rank-polymorphic semantics of direct-style REMORA [21]. The goal here is to translate the same ideas into a CPS setting.

2.4.1 Example 1: Lifting

Consider the following program, which applies the primitive $+$ to a rank-1 array and a scalar:

$$(\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ .\\ \qquad (+\ [4\ 1\ 3]\ 6\ \ k_{\texttt{TOP}}))$$

The $+$ primop expects two scalar arguments. However, in the given call the first argument is a vector, meaning that it has a rank of 1 instead of the expected rank of 0. This extra dimension is what we call this argument's *frame*, while the expected scalars are this argument's *cells*. Put differently, $[4\ 1\ 3]$ has a frame with shape (3) where each cell is a scalar. The second argument 6 has an empty frame and a single scalar cell.

The two arguments therefore disagree on their frames, and $+$ cannot consume $[4\ 1\ 3]$ as it stands. CPS REMORA resolves this by implicitly replicating 6 until it matches the larger frame, so it becomes $[6\ 6\ 6]$. This process is known as *lifting*. The call then runs once per index of the shared frame (3), all three applications in parallel:

$$\begin{bmatrix}(+\ 4\ 6)\\(+\ 1\ 6)\\(+\ 3\ 6)\end{bmatrix} = \begin{bmatrix}10\\7\\9\end{bmatrix}$$

The whole call returns $[10\ 7\ 9]$ to $k_{\texttt{TOP}}$.

In general, an argument with a nonempty frame means that the function does not handle all the data available at the call site. By the semantics of rank polymorphism, the arguments are replicated until they all share the same frame, known as the *principal frame*: the longest argument frame of the application. The function is then applied to the corresponding argument cells at each index of the principal frame, in parallel, so all the data that did not fit into an individual cell gets distributed across different function applications. The overall result is a frame of result cells. Notice that since the principal frame is determined by the available argument data, the parallelism model is entirely *data-driven*.

Simplest MIMD

Recall that the expression in the function position of a call is still an array. With a slight modification, we can turn this example into a very simple MIMD computation:

$$(\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ .\\ \qquad ([+\ -\ \times]\ [4\ 1\ 3]\ 6\ \ k_{\texttt{TOP}}))$$

Following the same lifting and application procedure, the result of this program would be $[10\ {-5}\ 18]$. A closer look at CPS REMORA's MIMD semantics is given in Section 2.4.3.

2.4.2 Example 2: Parallel Binding and Continuation

To get an intuition for how continuations work in CPS REMORA, consider the following direct-style REMORA program:

$$(\times\ 4\ ((\lambda\ (x\ 0).\ (-\ x\ 1))\ [6\ 1\ 2]))$$

In CPS REMORA an equivalent program looks as follows:

$$\begin{array}{l}(\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ . \\ \quad ((\texttt{fn}_{\texttt{L1}}\ [(x\ 0)]\ k_1\ .\ (-\ x\ 1\ k_1)) \\ \quad\quad [6\ 1\ 2] \\ \quad\quad (\texttt{cont}_{\texttt{L2}}\ y\ .\ (\times\ 4\ y\ \ k_{\texttt{TOP}}))))\end{array}$$

Continuations offer a way for CPS programs to sequence basic computations. When dealing with primitive applications, like $(-\ x\ 1\ k_1)$, the call's continuation argument gets invoked immediately once the result has been computed. In lambda applications, the situation is slightly different. A lambda application is essentially just a binding procedure, and that includes binding a continuation parameter that will get invoked at some later point.

Importantly, the continuation parameter still represents the notion of "what to do next" for a lambda. Here, applying $\texttt{fn}_{\texttt{L1}}$ to $[6\ 1\ 2]$ and continuation $\texttt{cont}_{\texttt{L2}}$ plays exactly this role: whatever $\texttt{fn}_{\texttt{L1}}$ computes will get bound to y in $\texttt{cont}_{\texttt{L2}}$'s body before proceeding with the computation.

Similar to the previous example, the outer call lifts because $\texttt{fn}_{\texttt{L1}}$'s parameter x expects a rank-0 argument. The principal frame of the application is (3), contributed by $[6\ 1\ 2]$. Since $\texttt{fn}_{\texttt{L1}}$ is applied to each of the three scalars making up $[6\ 1\ 2]$, a valid interpretation of this scenario is that each application creates a separate lexical closure. However, CPS REMORA instead binds x to the full argument array and only creates one closure. As long as we record that x is bound to a frame of three scalar cells, we can resolve each reference to x based on the frame index. Each of the primop calls inside $\texttt{fn}_{\texttt{L1}}$ then computes:

$$\begin{bmatrix}(-\ 6\ 1) \\ (-\ 1\ 1) \\ (-\ 2\ 1)\end{bmatrix} = \begin{bmatrix}5 \\ 0 \\ 1\end{bmatrix}$$

giving $y = [5\ 0\ 1]$. $\texttt{cont}_{\texttt{L2}}$'s own call to $\times$ lifts again, this time broadcasting the scalar 4 across y's frame (3):

$$\begin{bmatrix}(\times\ 4\ 5) \\ (\times\ 4\ 0) \\ (\times\ 4\ 1)\end{bmatrix} = \begin{bmatrix}20 \\ 0 \\ 4\end{bmatrix}$$

so the whole program returns $[20\ 0\ 4]$ to $k_{\texttt{TOP}}$.

2.4.3 Example 3: MIMD Split on If

Finally, we consider `if`, which is how MIMD computation commonly manifests.

$$
\begin{array}{l}
(\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ . \\
\quad ((\texttt{fn}_{\texttt{L1}}\ [(grd\ 0)]\ k_1\ . \\
\qquad ((\texttt{if}\ grd\ k_{\text{jp}}\ . \\
\qquad\quad (\texttt{ret}\ \ k_{\text{jp}}\ \ [3\ 1\ 9]) \\
\qquad\quad (\texttt{ret}\ \ k_{\text{jp}}\ \ [1\ 6\ 1])) \\
\qquad\ k_1)) \\
\quad [\texttt{\#t}\ \texttt{\#f}] \\
\quad k_{\texttt{TOP}}))
\end{array}
$$

We begin with lifting an application of $\texttt{fn}_{\texttt{L1}}$ to a principal frame of (2). Since *grd* binds scalar cells but is applied to $[\texttt{\#t}\ \texttt{\#f}]$, $\texttt{fn}_{\texttt{L1}}$'s body will run two times in parallel, once for each element of $[\texttt{\#t}\ \texttt{\#f}]$.

grd is used as a guard of an `if` form, and since the two booleans differ, each of the calls to `if` will take a different branch:

$$
\begin{array}{l}
grd = \texttt{\#t} \to [3\ 1\ 9] \\
grd = \texttt{\#f} \to [1\ 6\ 1]
\end{array}
$$

This is what we call a MIMD split: rather than one instruction being applied uniformly across a frame, the frame's two positions diverge. Only after both branches are done can their results be reassembled back into a single array. In this sense, MIMD splits enforce a synchronization barrier. The value handed back to $k_{\texttt{TOP}}$ will be $[[3\ 1\ 9]\ [1\ 6\ 1]]$.

MIMD computation as multiple SIMD computations

A helpful way to think of MIMD computation is to understand it as multiple independently running SIMD computations, which we will call *SIMD subcomputations*. In the example above, that means *grd*'s frame of two positions splits into two SIMD subcomputations: one runs the `#t` branch, the other runs the `#f` branch. In this case, both branches just return an array literal, but nothing requires them to do a comparable amount of work after the split. The only constraint is that each branch must eventually produce a result cell with the same shape. Only after every SIMD subcomputation has computed its *result fragment* may they be assembled back together as the result of the original MIMD computation.

Chapter 3

SIMD/MIMD Machine

This chapter and Chapter 4 formalize the operational semantics of CPS REMORA in the form of the *SIMD/MIMD Machine*. This semantics makes precise the intuitions established by Section 2.4 and brings out interesting consequences of rank polymorphism in a CPS setting.

The discussion is split across two chapters. This chapter establishes notation, introduces relevant semantic domains and presents the high-level machine structure. It also formalizes machine state, machine transitions, and defines initial and final states. The SIMD evaluators that make up the machine, called cores, are defined, but the discussion of individual core transitions is saved for Chapter 4.

This chapter is organized as follows:

- Section 3.1 introduces notation for sets, sequences, tuples, and maps;
- Section 3.2 defines the semantic domains (i.e. parallel context, index domains, collections, cells, environments, values) that the machine operates over;
- Section 3.3 introduces the machine structure and the general parallelism model, defines SIMD cores and their states, defines the machine state and its stepping relation, and describes how the machine is run.

3.1 Notational Conventions II

This section introduces the notational conventions for standard mathematical structures that are used in the presentation of the SIMD/MIMD Machine.

3.1.1 Set Combinators and Tuples

$\times$ and $+$ are the two set combinators used to produce more structurally complicated sets out of already defined ones. Set products $\alpha \times \beta$ define a set of pairs, with one element of α and one of β. Set disjoint unions $\alpha + \beta$ describe a set of variants: either an element of α or of β. Both generalize to n-ary combinators.

Product type elements, also known as *tuples*, see a lot of use in the semantics. They get their own angle-bracket notation:

$$A^0 \times \cdots \times A^n \quad ::= \quad \langle \alpha^0, \ldots, \alpha^n \rangle \qquad \text{where } \alpha^0 \in A^0, \ldots$$

FIGURE 3.1: Tuples.

3.1.2 Sequences

$$\alpha^\star \in Seq(A) \quad ::= \quad \bullet \mid [\alpha_0\ \alpha_1 \ldots] \qquad \text{where } \alpha \in A$$

FIGURE 3.2: Sequences.

A *sequence* $\alpha^\star$ is either the empty sequence $\bullet$ or a nonempty list of αs, written out in brackets like $[\alpha_0\ \alpha_1 \ldots]$. When clear, the brackets may be omitted.

Common operations

- Two sequences can be concatenated with $+\!\!+$:

$$\alpha_1^\star +\!\!+ \alpha_2^\star$$

- Sequence comprehension builds a new sequence out of an old one, applying some f to every element in order. Unlike a set comprehension, this preserves order.

$$[f(\alpha) \mid \alpha \in \alpha^\star]$$

- Finally, sequences also come with the usual family of prefix/suffix operations, defined implicitly. The relevant operations are:

$$\texttt{take} : \mathbb{N} \to Seq(A) \to Seq(A)$$
$$\texttt{drop} : \mathbb{N} \to Seq(A) \to Seq(A)$$
$$\texttt{splitRight} : \mathbb{N} \to Seq(A) \to (Seq(A) \times Seq(A))$$

  Here, `take` $n\ \alpha^\star$ and `drop` $n\ \alpha^\star$ keep and discard the first n elements of $\alpha^\star$ respectively. `splitRight` $n\ \alpha^\star$ splits up a sequence at the nth element from the right.

3.1.3 Sets

$$\alpha^{\#} \in \mathit{Set}(A) \quad ::= \quad \varnothing \mid \{\alpha_0,\ \alpha_1,\ \ldots\} \qquad \text{where } \alpha \in A$$

FIGURE 3.3: Sets.

A *set* $\alpha^{\#}$ is an unordered, duplicate-free collection of αs. We list elements directly when convenient, and otherwise lean on set-builder notation.

Common operations

- Two sets can be unioned with $\cup$; i-indexed sets can be unioned with $\bigcup$:

$$\alpha_1^{\#} \cup \alpha_2^{\#} \qquad\qquad \bigcup_i \alpha_i^{\#}$$

- Set comprehension collects $f(x)$ for every x drawn from some S that satisfies an optional predicate P:

$$\{f(x) \mid x \in S,\ P(x)\}$$

3.1.4 Maps

$$mp \in A \rightharpoonup B \quad ::= \quad \varnothing \mid [\alpha_0 \mapsto \beta_0,\ \alpha_1 \mapsto \beta_1,\ \ldots] \qquad \text{where } \alpha \in A,\ \beta \in B$$

FIGURE 3.4: Partial maps.

A *(partial) map* associates some, but not necessarily all, of α with a β: the empty map $\varnothing$ associates nothing, and a nonempty one is written out as a list of bindings.

Common operations

- A map is applied to a key a to retrieve its binding, which exists only when a is in the map's domain:

$$mp\ a$$

- Maps can be unioned provided their domains are disjoint. Two maps are unioned with $\uplus$; i-indexed are unioned with $\biguplus$. There is also sugar to extend a map with a single new binding:

$$mp_1 \uplus mp_2 \qquad\qquad \biguplus_i mp_i \qquad\qquad mp[a \mapsto b]$$

- Map comprehension builds a binding for each element from a given iterator:

$$[a \mapsto f(a) \mid a \in S]$$

3.2 Semantic Domains

3.2.1 Dimensions and Shapes

$$\begin{aligned} d \in \mathit{Dim} &:= \mathbb{N} \\ \sigma \in \mathit{Shape} &:= \mathit{Seq}(\mathit{Dim}) \\ \Sigma \in \mathit{ParCtx} &:= \mathit{Shape} \end{aligned}$$

FIGURE 3.5: Dimensions and Shapes.

A *dimension* d is just a natural number describing the size of an array along one of its axes. A *shape* σ is a sequence of dimensions, one per axis, so its length is the rank of an array. It is also often referred to as an *iteration space*.

A *parallel context* Σ is just a shape, but one used in a specific context. It describes the upper-bound iteration space of a given SIMD evaluator, whose purpose is explained in detail in Section 3.3.3. It gets its own set name and symbol, Σ, because it plays a significant role in the execution of the SIMD/MIMD Machine and we want it to be visually distinct.

Common operations

In the context of the SIMD/MIMD Machine, sequences are heavily used to describe shapes and indices (see below). As a result, it is appropriate to refer to the length of such sequences as their rank. The rank function is defined as:

$$\begin{aligned} &\texttt{rank} : \mathit{Seq}(A) \to \mathbb{N} \\ &\texttt{rank}\ [\alpha_0\ \ldots\ \alpha_{n-1}] = n \end{aligned}$$

3.2.2 Indices and Domains

$$\begin{aligned} i \in \mathit{DimIdx} &:= \mathbb{N} \\ \mathit{idx} \in \mathit{Index} &:= \mathit{Seq}(\mathit{DimIdx}) \\ D \in \mathit{Domain}\ r &:= \{\mathit{idx} \in \mathit{Index} \mid \texttt{rank}\ \mathit{idx} = r\} \end{aligned}$$

FIGURE 3.6: Indices and Domains.

A *dimension index* i is just a non-negative zero-indexed address within a dimension. A dimension index i is valid for a dimension d when $0 \leq i < d$. An *index* idx is a sequence of dimension indices that addresses a full shape, one dimension index per axis. For an index to be a valid shape address, both their ranks must agree and each dimension index must be valid for the corresponding dimension.

The rank-0 case is worth highlighting, as it comes up throughout the chapter as the base case. A scalar shape has no axes, so the only index that addresses it is the *empty index* $[\bullet]$.

A *domain* D is a set of indices that address some shape. A shape has a fixed rank, so a valid domain will always consist of indices with the same rank r. This is made explicit in the definition by parametrizing *Domain* over rank r. However, later sections will drop the parameter whenever the rank is clear from context or does not matter to the discussion at hand.

Common operations

A basic domain operation is taking a Cartesian product of two domains, given by $\bowtie$. This is how a domain is extended to address a higher-dimensional shape.

$$\bowtie : \mathit{Domain}\ r_1 \to \mathit{Domain}\ r_2 \to \mathit{Domain}\ (r_1 + r_2)$$
$$D \bowtie D' := \{idx +\!\!+ idx' \mid \forall idx \in D,\ \ idx' \in D'\}$$

Another common operation is generating a domain that contains every valid index of an iteration space specified by shape. Concretely, it is the Cartesian product of each dimension's index range.

$$\texttt{dom} : \mathit{Shape} \to \mathit{Domain}$$
$$\texttt{dom}\ \bullet = \{[\bullet]\}$$
$$\texttt{dom}\ [d_0\ d_1 \ldots] = (\texttt{rangeSet}\ d_0) \bowtie (\texttt{dom}\ [d_1 \ldots])$$

where `rangeSet` is given as:

$$\texttt{rangeSet} : \mathit{Dim} \to \mathit{Set}(\mathit{DimIdx})$$
$$\texttt{rangeSet}\ d = \{0, \ldots, d-1\}$$

Note that for a sequence of r dimensions, every element in the output set will be an index of length r which belongs to *Domain* r.

The base case follows from the empty index: a scalar iteration space contains the single position $[\bullet]$. This also makes $\texttt{dom}\ \bullet$ the unit of $\bowtie$, so that $\texttt{dom}\ [d]$ comes out as $\texttt{rangeSet}\ d$ and not as the empty set.

Finally, while domains are unordered sets, sometimes it is useful to organize indices in a *row-major order* to line up with syntactic data layout in array literals (Sections 2.2.1 and 4.1). `domList` works like `dom`, except the output is a sorted sequence of indices in row-major order. The signature for the function is simply:

$$\texttt{domList} : \mathit{Shape} \to \mathit{Seq}(\mathit{Index})$$

3.2.3 Index Maps

$$\begin{aligned} m^\alpha \in \mathit{IdxMap}\ \alpha \quad ::= \quad & \texttt{IdxMap}\ (i \rightharpoonup m^\alpha) \\ \mid \quad & \texttt{Elem}\ \alpha \end{aligned}$$

FIGURE 3.7: Index maps.

An *index map* is either a `IdxMap` partial map from dimension indices to index maps or a leaf element `Elem` α parameterized by value α.

For convenience, index maps are built from a set of index-value pairs with an auxiliary constructor `makeIdxMap`. Given one pair per full-rank index, it constructs the corresponding nested `IdxMap` structure, placing `Elem` α at each leaf.

$$\texttt{makeIdxMap} : \mathit{Set}(\mathit{Index} \times \alpha) \to \mathit{IdxMap}\ \alpha$$

We use syntactic sugar for index maps to elide `IdxMap` and `Elem` constructors. For example, the 2×3 array literal

$$\begin{bmatrix} 4 & 1 & 3 \\ 6 & 1 & 2 \end{bmatrix}$$

is represented as the index map

$$\begin{pmatrix} 0 \mapsto \begin{pmatrix} 0 \mapsto 4 \\ 1 \mapsto 1 \\ 2 \mapsto 3 \end{pmatrix} \\ 1 \mapsto \begin{pmatrix} 0 \mapsto 6 \\ 1 \mapsto 1 \\ 2 \mapsto 2 \end{pmatrix} \end{pmatrix}$$

The example above shows how regular array data can be encoded, but notice that index maps are flexible enough to support irregular data. For example, this too is a valid index map:

$$\begin{pmatrix} 0 \mapsto \begin{pmatrix} 0 \mapsto 4 \\ 2 \mapsto 3 \end{pmatrix} \\ 1 \mapsto \begin{pmatrix} 0 \mapsto 6 \end{pmatrix} \end{pmatrix}$$

This is useful because our model often has to deal with partial data describing MIMD result fragments (Section 4.6.3). Additionally, this representation enables implicit data replication within collections, which is demonstrated in Section 3.2.5.

Common operations

An important operation for detecting MIMD splits is extracting a set of *distinct* values present in an index map (Section 4.3.2). `getElemSet` is a function that does exactly that, unioning every leaf value into a set *up to value equality*. That is, every two leaves holding equal values contribute only one element to the resulting set.

$$\texttt{getElemSet} : \mathit{IdxMap}\ \alpha \to \mathit{Set}(\alpha)$$

3.2.4 Collection Dimensions

The fundamental semantic value of CPS REMORA is a *collection*, defined in the following subsection. It relies on encoding information about data parallelism as a special shape, which consists of annotated dimensions.

$$\begin{aligned} \delta \in \mathit{CollDim} &::= d \mid \underline{d} \\ \psi, \omega \in \mathit{CollDims} &::= \delta^{\star} \end{aligned}$$

FIGURE 3.8: Collection dimensions.

A *collection dimension* δ is a dimension that might be underlined. A non-underlined, *real dimension* is just a natural number that describes the size of a dimension. This is exactly like a regular array dimension. On the other hand, the underlined, *replicated dimension* denotes that the collection is replicated along that dimension. It implies that the collection has been expanded as a result of lifting, but the underlying value data has not yet been copied over to back up this replication. It is fine to keep it simply as an annotation because when the replicated data is needed, indexing will take care of it.

ψ and ω are (possibly empty) sequences of *collection dimensions*. They are exactly the components making up collection shapes. Notice that any shape—and thus any parallel context—is a valid sequence of collection dimensions in which every dimension is real. More formally, we will rely on the fact that $\mathit{ParCtx} \subset \mathit{CollDims}$ when computing principal frames during lifting (Section 4.2.2).

Common operations

When defining operations on collections, we often need to add or remove underlines on every dimension in a sequence of collection dimensions. The helpers `addUnderlines` and `removeUnderlines` do exactly that.

$$\begin{aligned} &\texttt{addUnderlines} : \mathit{CollDims} \to \mathit{CollDims} \\ &\texttt{removeUnderlines} : \mathit{CollDims} \to \mathit{CollDims} \end{aligned}$$

3.2.5 Collections

Collections are the primary way array data gets represented within the SIMD/MIMD Machine. They are flexible and parallelism-aware, which allows for "free" implicit data replication and arbitrary data fragmentation.

$$\begin{aligned} \phi \in \mathit{CollShape} &::= (\psi \mid \omega) \\ \boxed{\alpha} \in \mathit{Collect}\ \alpha &::= \ulcorner \phi \parallel m^{\alpha} \urcorner \end{aligned}$$

FIGURE 3.9: Collections.

A *collection* is an index map paired with a *collection shape*, written as $(\psi \mid \omega)$. Here, ψ is a sequence of *frame dimensions* and ω is a sequence of *cell dimensions*. The vertical

bar separating them can be thought of as a cursor that adjusts the granularity of parallelism: frame dimensions describe how many independent copies of a computation there are, while cell dimensions describe the data shape visible to each computation. This intuition will become clearer after Section 3.3 formalizes the notion of a "compute element".

As an example, consider collection shape $(2\ \underline{4} \mid 3)$. Here, $2\ \underline{4}$ are frame dimensions and 3 is a cell dimension. The shape suggests that the associated collection contains $2 \times 4 \times 3 = 24$ elements overall, which can also be interpreted as $2 \times 4 = 8$ instances of vectors of length 3, each accessed independently. Notice also that $\underline{4}$ is a replicated dimension, which means that the collection does not vary along that axis. For a fixed index in the preceding real dimension 2, every valid index in $\underline{4}$ will address the exact same length-3 vector. Put differently, the collection behaves as if a single vector had been copied 4 times to fill out that dimension. So, to back this collection with data, it is sufficient to store only $2 \times 3 = 6$ values; everything else is just implicit copies. This is precisely what collections do. An example collection with this shape is:

$$\lceil\!\lfloor (2\ \underline{4} \mid 3) \Vert \begin{pmatrix} 0 \mapsto \begin{pmatrix} 0 \mapsto 4 \\ 1 \mapsto 1 \\ 2 \mapsto 3 \end{pmatrix} \\ 1 \mapsto \begin{pmatrix} 0 \mapsto 6 \\ 1 \mapsto 1 \\ 2 \mapsto 2 \end{pmatrix} \end{pmatrix} \rceil\!\rfloor$$

Implicit replication happens a lot due to rank-polymorphic lifting, so this mechanism turns out to be very useful. From the moment a syntactic array literal is converted into a collection, its underlying index map stays unchanged for the entire lifetime of the value. If the collection gets expanded due to lifting, the only necessary change to reflect that is adding underlined dimensions to the shape (Section 4.2.2).

As another example, consider the following collection where every dimension is replicated, including its cell dimension. This is useful when representing SIMD collections of procedures:

$$\lceil\!\lfloor (\underline{5}\ \underline{4} \mid \underline{3}) \parallel + \rceil\!\rfloor$$

Cells

A computationally significant kind of collection is one that has an empty sequence of frame dimensions. This means the "shape of parallelism" is a scalar, so we are looking just at cell data. Only in this form can a compute element perform primitive operations on the data (Section 4.4.2).

Usually, frame dimension sequences are not empty. Since the frame shape describes how many independent copies of a computation there are, with each seeing its own cell data, one can think of collections as *frames of cells*. Each cell, then, implicitly has an empty frame and only describes the underlying cell data. This is exactly the representation compute elements operate on.

Cells get their own notation.

$$\boxed{\boxed{\alpha}} \in \mathit{Cell}\ \alpha \quad ::= \quad \boxed{d^\star \parallel m^\alpha}$$

FIGURE 3.10: Cells.

Notice that the definition of a cell requires the dimensions describing the shape of a cell to be real, non-underlined dimensions. This is because when CEs execute a primop, they expect all data to be reified and available.

`makeCell` is the constructor that builds cells from collection components. Specifically, given a sequence of cell dimensions ω and the index map they describe, it reifies any replicated axes and packages the result as a cell.

$$\texttt{makeCell} : \mathit{CollDims} \to \mathit{IdxMap}\ \alpha \to \mathit{Cell}\ \alpha$$

Indexing collections

From a frame-of-cells point of view, a collection index is a frame address that points to a cell. More precisely, a valid collection index *idx* must be a valid index of the collection's frame shape ψ. The indexing operator @ is defined as follows:

$$\begin{aligned}
&@ : \mathit{Collect}\ \alpha \to \mathit{Index} \to \mathit{Cell}\ \alpha \\
&\lceil\!\lfloor (\bullet \mid \omega) \parallel m^\alpha \rceil\!\rfloor\ @\ [\bullet] = \texttt{makeCell}\ \omega\ m^\alpha \\
&\lceil\!\lfloor (d_0\ \delta_1 \ldots \mid \omega) \parallel m^\alpha \rceil\!\rfloor\ @\ [i_0\ i_1 \ldots] = \lceil\!\lfloor (\delta_1 \ldots \mid \omega) \parallel m^\alpha\ i_0 \rceil\!\rfloor\ @\ [i_1 \ldots] \qquad (0 \le i_1 < d_0) \\
&\lceil\!\lfloor (\underline{d_0}\ \delta_1 \ldots \mid \omega) \parallel m^\alpha \rceil\!\rfloor\ @\ [i_0\ i_1 \ldots] = \lceil\!\lfloor (\delta_1 \ldots \mid \omega) \parallel m^\alpha \rceil\!\rfloor\ @\ [i_1 \ldots] \qquad (0 \le i_1 < d_0)
\end{aligned}$$

The base case result is a cell, constructed with `makeCell`. This means that the output cell will always have its data reified and ready to be processed by a compute element.

The second case takes the leading dimension index i_0 and applies m^α to it. Since index maps are just a partial maps from indices to index maps, this lookup will select the nested index map corresponding to i_0.

In the last case, indexing into a replicated dimension $\underline{d_0}$ just causes the leading dimension index i_0 to be dropped. Regardless of the index used to address a replicated dimension, it always produces the same result.

Reassembling cells into index maps

If indexing takes collections apart, then `cellsToIdxMap` is for putting cells and their corresponding frame indices back together again. Given a set of index-cell pairs, this operation combines them into a new index map.

$$\begin{aligned}
&\texttt{cellsToIdxMap} : \mathit{Set}(\mathit{Index} \times \mathit{Cell}\ \alpha) \to \mathit{IdxMap}\ \alpha \\
&\texttt{cellsToIdxMap}\ S = \\
&\qquad \texttt{makeIdxMap} \left\{ \langle idx_\psi +\!\!+ idx_\omega,\ \alpha \rangle \;\middle|\; \langle idx_\psi,\ \boxed{_ \parallel m^\alpha} \rangle \in S,\ \langle idx_\omega,\ \alpha \rangle \in m^\alpha \right\}
\end{aligned}$$

where $\langle idx_\omega,\ \alpha \rangle \in m^\alpha$ ranges over every full-rank index of the cell's own index map together with the leaf found there.

Notice that `cellsToIdxMap` only rebuilds the index map, not a full collection. This is because the collection shape ϕ is not always recoverable from a set of index-cell pairs alone. For example, when dealing with collection fragments as a result of a MIMD split, only the caller has access to the overall expected parallel context of the collection.

Domain restriction

Recall that a domain is a set of indices. Domain restriction of a collection, then, is best understood as bulk indexing: instead of addressing one index to get back one cell, it addresses an entire domain at once and reassembles the resulting cells back into a collection. This operation is useful when dealing with partial domains, such as MIMD result fragments.

$\boxed{\alpha} \downarrow D$ restricts $\boxed{\alpha}$ down to domain D.

$$\downarrow : \mathit{Collect}\ \alpha \to \mathit{Domain} \to \mathit{Collect}\ \alpha$$
$$\ulcorner\!\!\lfloor\ \phi \parallel m^{\alpha}\ \rfloor\!\!\urcorner \downarrow D = \ulcorner\!\!\lfloor\ \phi' \parallel m^{\alpha}_{D}\ \rfloor\!\!\urcorner$$

where

$$\phi' = \texttt{removeUnderlines}\ \phi$$
$$m^{\alpha}_{D} = \texttt{cellsToIdxMap}\left\{ \langle idx,\ \ulcorner\!\!\lfloor\ \phi \parallel m^{\alpha}\ \rfloor\!\!\urcorner\ @\ idx \rangle \ \mid\ \forall idx \in D \right\}$$

This operation indexes $\boxed{\alpha}$ at every index of D and assembles the resulting cells back together with `cellsToIdxMap`. A scalar domain $\{[\bullet]\}$ needs no special treatment, since it simply indexes at $[\bullet]$ and produces a singleton map.

Because indexing produces cells, all the data in the underlying index maps will have been reified at that point, so domain restriction will always produce a reified collection. The resulting collection shape ϕ' is simply the original shape with all the underlines removed.

3.2.6 Environments

The SIMD/MIMD Machine describes an environment-based semantics. In preparation for future abstraction, the environment is factored into two parts [22].

$$\begin{array}{rlll} \mathit{cn} & \in \mathit{Contour} & & \\ \beta & \in \mathit{BEnv} & := & \mathit{Lab} \rightharpoonup \mathit{Contour} \\ \mathit{ve} & \in \mathit{VEnv} & := & (\mathit{Contour} \times \mathit{Var} \rightharpoonup \mathit{Val}) \ +\ (\mathit{Contour} \times \mathit{CVar} \rightharpoonup \mathit{CVal}) \end{array}$$

FIGURE 3.11: Contours and Environments.

The key idea here is *binding contours*. Each time a lambda application binds argument values to its parameters, a fresh contour is allocated out of the *Contour* set. The contour identifies this specific binding instance, so multiple applications of the same lambda can be differentiated. A *variable binding*, then, is a variable name paired up with a contour that was allocated during its binding. Both the contour set and the allocation mechanism are described later, in Section 3.3.6.

A *binding environment* (*BEnv*) is a (partial) map from lambda labels to contours. It is a lexical environment that identifies which contour is currently bound to each lambda's label at this particular point in evaluation; when paired with a lambda, it becomes a lexical closure. This is what allows us to resolve variable references (Section 4.1).

A *value environment* (*VEnv*), as the name suggests, stores values, which will be defined in the following subsection. Structurally, it is the disjoint union of (partial) maps from variable bindings to values, and from continuation variable bindings to continuation values. This disjointness is a downstream effect of factored CPS since *Var* and *CVar* are already disjoint. Unlike binding environments, this environment is global and every variable lookup consults it. Since CPS REMORA is pure and fresh contours are allocated on each application, the value environment grows monotonically.

Variable lookup

Given a variable x, `binder` x is the label of the lambda or continuation lambda that statically binds it, and since we assume that all CPS REMORA programs are alphatized, every variable has exactly one binder.

Given the global value environment ve, we can look up a variable x under some binding environment β as:

$$ve\ \langle x,\ \beta\ (\texttt{binder}\ x)\rangle$$

3.2.7 Machine Values

Similar to how every value in CPS REMORA is an array, SIMD/MIMD Machine (non-continuation) values are collections.

$$\begin{aligned} \varsigma \in \mathit{Clo} &:= \mathit{BEnv} \times \mathit{Lam} \\ \rho \in \mathit{Proc} &:= \mathit{Prim} + \mathit{Clo} \\ v \in \mathit{ValAtom} &:= \mathit{Basic} + \mathit{Proc} \\ \boxed{\rho}, \boxed{v} \in \mathit{Val} &:= \mathit{Collect}\ \mathit{ValAtom} \\ \boxed{\boxed{v}} \in \mathit{CellVal} &:= \mathit{Cell}\ \mathit{ValAtom} \end{aligned}$$

FIGURE 3.12: SIMD/MIMD Machine Values.

A machine value is defined over *value atoms*, which are either basic values (numbers and booleans, in the current instance) or procedures. A *procedure* ρ is either a primitive operation or a *closure* ς, which is a pair coupling a binding environment with a lambda.

A *value* and *cell value* are, respectively, a collection and a cell of value atoms. Strictly speaking, the notation for a collection of value atoms is $\lceil\!\lfloor \phi \parallel m^v \rfloor\!\rceil$, but for convenience we omit the v and simply write $\lceil\!\lfloor \phi \parallel m \rfloor\!\rceil$ from here on.

Collections of procedures will appear in the function position during calls. Since collections might contain arbitrary data, it is possible to have a collection with multiple distinct procedures. This is the basis of MIMD computation.

This instruction heterogeneity is bounded, however: the source program's type-checking guarantees that every procedure atom within a single procedure collection shares the same signature. Even when a MIMD split occurs, each of the diverging procedures still consumes the same number of arguments and expects the same ranks for

each (Section 4.2.2). This is what makes it safe to lift and dispatch a heterogeneous procedure collection using one shared notion of expected argument ranks.

3.2.8 Continuation Values

Since CPS REMORA is a continuation-passing style IR, we deal with continuations. Continuation values behave differently from data and require more information to facilitate control flow.

$$\begin{aligned} \varsigma_\kappa \in \mathit{CClo} &:= \mathit{BEnv} \times \mathit{CLam} \\ \rho_\kappa \in \mathit{CProc} &:= \mathit{CClo} + \mathit{CPrim} \\ \kappa \in \mathit{CVal} &:= \mathit{CProc} \times \mathit{ParCtx} \times \mathit{Domain} \end{aligned}$$

FIGURE 3.13: Continuation Values.

A *continuation procedure* ρ_κ is either a continuation primitive (defined below) or a closure. A *continuation closure* ς_κ is just a pair of a binding environment and a continuation lambda.

A *continuation value* κ is a 3-tuple: it combines a continuation procedure with the active parallel context Σ and domain D at the moment the continuation is captured. These are an important part of the dynamic state and it would be impossible to invoke a continuation without restoring them (Section 4.5).

The machine's own top-level continuation, $\kappa_{\texttt{HALT}}$, is an instantiation of a continuation value triple. Figure 3.22 in the following section gives its exact definition.

Continuation primitives

$$\mathit{cprim} \in \mathit{CPrim} \quad ::= \quad \texttt{yield}(\mathit{cid})$$

FIGURE 3.14: Continuation primitives.

A *continuation primitive* encodes a special control transfer. For now, we define only one such primitive, $\texttt{yield}(\mathit{cid})$, but as will be discussed in Section 6.1, it can be extended to support more interesting control flow.

$\texttt{yield}(\mathit{cid})$ indicates that a certain subcomputation has been completed and the result is ready to be communicated back to the MIMD split point that requested it. To identify the split point, the primitive carries a core id *cid*, which will be described in detail in Section 3.3.4.

3.3 SIMD/MIMD Machine

The concrete semantics of CPS REMORA is formalized in terms of the *SIMD/MIMD Machine*. The primary design goals were to make explicit (1) which computation is performed in SIMD mode and in what parallel context, (2) when precisely MIMD splits occur, and (3) how to perform the computation with potentially limited compute resources. In contrast, the semantics of REMORA is much more abstract [23]. It does not model the executing system as a whole or keep track of when SIMD computation diverges into MIMD.

Additionally, this semantics aims to maximize SIMD at a MIMD split. Recall that MIMD computation is just a number of SIMD computations running independently. A trivial split would be to treat every piece of data (and its corresponding instruction) as a SIMD computation. However, this semantics seeks to group all data executing with the same instruction into a single SIMD computation. It also means that a MIMD computation with a single distinct procedure can be easily converted into a SIMD computation, avoiding an unnecessary MIMD split.

3.3.1 The Big Idea

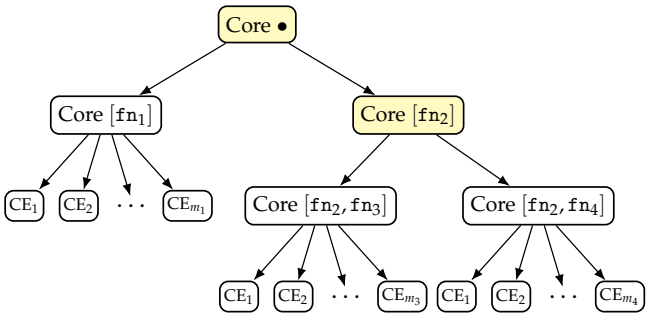

FIGURE 3.15: Example SIMD/MIMD Machine Snapshot. Here, two MIMD splits have taken place. First, the top level Core • encountered a two-way MIMD split between $\texttt{fn}_1$ and $\texttt{fn}_2$. Then, as Core $[\texttt{fn}_2]$ was executing, it encountered another two-way MIMD split between $\texttt{fn}_3$ and $\texttt{fn}_4$. Leaf cores with ids $[\texttt{fn}_1]$, $[\texttt{fn}_2, \texttt{fn}_3]$ and $[\texttt{fn}_2, \texttt{fn}_4]$ are presently eligible for execution since they are not waiting on results from any child cores. Each of the leaf cores can address a number of compute elements (CEs). All other cores are suspended, shown in yellow.

At a high level, the SIMD/MIMD Machine can be thought of as a set of *cores* (Section 3.3.5), each of which executes SIMD computations. A core itself is an abstraction over an arbitrarily large set of *compute elements* (CEs), all executing in lockstep. During execution, every CE within a core performs the same instruction over the data visible to the core. That is, cores are SIMD evaluators. The core is responsible for tracking current CE parallelism and iteration space (Section 3.3.3).

At every machine step, an arbitrary subset of cores will be selected to progress (Section 3.3.6). Since machine-level steps pick multiple cores to step at a time, a machine step can be seen as MIMD computation, while core-level steps are SIMD. The ability to pick

the number of cores to advance on every machine step is what allows the SIMD/MIMD Machine to model execution with limited resources. Specifically, if a system has n SIMD processing units, then at every machine step at most n cores can be picked to run.

When a core is selected to take a step, it will trivially execute SIMD in all cases except immediately before function application (Section 3.3.2, Section 3.3.5). This is where the machine will try to identify distinct procedures in the collection that appears in function position. If the machine determines that the application preserves SIMD execution, then the core will proceed as normal. However, if applying this collection of procedures would cause CEs to diverge, we have a MIMD computation. The core then will extract all the SIMD subcomputations and *spawn* a new core for each. We refer to the core that originally encountered a MIMD computation as the *parent core*, and all the cores that it spawns as *child cores*.

After the parent core spawns its children, it will enter a *Wait* state until all the children have completed their subcomputations. Once all the child cores reach a *Done* state, the parent assembles the result of the MIMD computation and resumes SIMD execution. The child cores will die after their parent moves on from the *Wait* state.

Notice that the wait-for-children-then-resume behavior of parent cores acts like a synchronization barrier. This is exactly what we expect when dealing with MIMD computations on SIMD-capable hardware.

Each core has a *core id* (Section 3.3.4) which never changes throughout the core's lifetime after initialization on MIMD split. The id for a new core is constructed by appending a procedure identifier of the current SIMD subcomputation to the parent core id. This way, the core id encodes the sequence of MIMD splits that led to this core's creation.

Since core ids capture ancestry, it is helpful to think about the machine as not just a set of cores, but a *tree* of cores. Then, only leaf nodes (i.e. most-recently-spawned child cores) will be actively performing computation, while the internal nodes will be waiting for child nodes to produce results. An example of core structure during execution can be seen in Figure 3.15.

The rest of this section formalizes core state and eventually machine state. Section 3.3.6 defines the machine stepping relation, while the core stepping relation is discussed in detail in Chapter 4.

3.3.2 Core Execution State

While the full core state includes auxiliary data, the primary component of core state is its *execution state*. As will be discussed in Section 3.3.5, this component deterministically specifies the stepping relation for cores. In other words, the core execution state captures the idea of what the core will do when it is run.

$$
\begin{aligned}
st \in \mathit{State} &:= \mathit{Eval} + \mathit{Lift} + \mathit{Split?} + \mathit{Apply} + \mathit{CApply} + \mathit{Wait} + \mathit{Done} \\
\\
\mathit{Eval} &:= \mathit{BEnv} \times \mathit{Call} \\
\mathit{Lift} &:= \mathit{Val} \times \mathit{Seq}(\mathit{Val}) \times \mathit{CVal} \\
\mathit{Split?} &:= \mathit{Val} \times \mathit{Seq}(\mathit{Val}) \times \mathit{CVal} \\
\mathit{Apply} &:= \mathit{Proc} \times \mathit{Seq}(\mathit{Val}) \times \mathit{CVal} \\
\mathit{CApply} &:= \mathit{CVal} \times \mathit{Val} \\
\mathit{Wait} &:= \mathit{Set}(\mathit{CoreId}) \times \mathit{CVal} \\
\mathit{Done} &:= \mathit{Val} \times \mathit{CoreId}
\end{aligned}
$$

FIGURE 3.16: Core execution state domains.

The execution state is one of the seven tuples given above. Additionally, since the tuple notation will make it difficult to tell at a glance which core state we are dealing with, each of the execution states comes with notational sugar.

The *Eval* state, written $\text{Eval}(\beta,\ \mathit{call})$, converts syntactic *Call* into semantic values that the rest of the machine understands. It also carries a binding environment *BEnv* to look up variables which might appear in the syntax.

The *Lift* state, written $\text{Lift}(\boxed{\rho},\ \boxed{v}^\star,\ \kappa)$, is responsible for preparing the core for a function application. Given a procedure collection, a sequence of argument collections and a continuation value, it negotiates the parallel context for the application in a process known as "lifting" (Section 4.2.2), growing the core's parallel context and domain to match. Once lifting is done, the core hands the lifted collections off to *Split*?.

The *Split*? state, written $\text{Split?}(\boxed{\rho},\ \boxed{v}^\star,\ \kappa)$, examines the collection in the function position to determine whether this application is a SIMD computation (Section 4.3.2). If yes, the core can apply the single procedure in the *Apply* state. If no, the core will spawn children and enter a *Wait* state.

The *Apply* state, written $\text{Apply}(\rho,\ \boxed{v}^\star,\ \kappa)$, takes care of SIMD function application. During it, every CE in the core performs the same procedure over lifted argument collections. As per CPS, the result of the application is passed to the continuation value.

The *CApply* state, written $\text{CApply}(\kappa,\ \boxed{v})$, is the continuation application state. Here the continuation value is invoked with some result collection value.

The *Wait* state, written $\text{Wait}(\mathit{cid}^\#,\ \kappa)$, arises only as a result of a core encountering a MIMD split. The state records a set of child core ids that this parent core is waiting on (Section 3.3.4) and a continuation value to invoke once the core is ready to resume SIMD execution. This continuation value acts as a synchronization barrier for the MIMD split.

Finally, the *Done* state, written $\text{Done}(\boxed{v},\ \mathit{cid})$, marks a core as finished, carrying its result and the core id of the parent waiting on it. A top-level core in this state indicates that the program execution is complete and the result is the result of the whole program.

3.3.3 Parallel Context and Domain

Another important component of a core's dynamic state is its notion of CE iteration space. The SIMD/MIMD Machine cores encode it with two values: a parallel context Σ (Section 3.2.1) and a domain of indices D (Section 3.2.2) that addresses the available CEs.

A *parallel context* is a sequence of dimensions, representing the upper bound shape of the frame this core is currently computing across. Put differently, the iteration space spanned by those dimensions is the upper bound on the current SIMD parallelism within the core.

During a MIMD split, the iteration space that was originally available to the parent core will get distributed among child cores in order to execute their SIMD subcomputations. The overall shape of machine parallelism, however, stays the same, and child cores need to be aware of it to perform lifting (Section 4.2.2). This is why the parallel context is an upper bound and also why we call it a *context* rather than just a frame shape.

In contrast, a *domain* is a set of indices that specify the precise iteration space of the core. It is also the set of indices that address all the CEs available to the core. Every index in the domain will always belong to the iteration space spanned by the parallel context dimensions. Figure 3.17 illustrates the distinction.

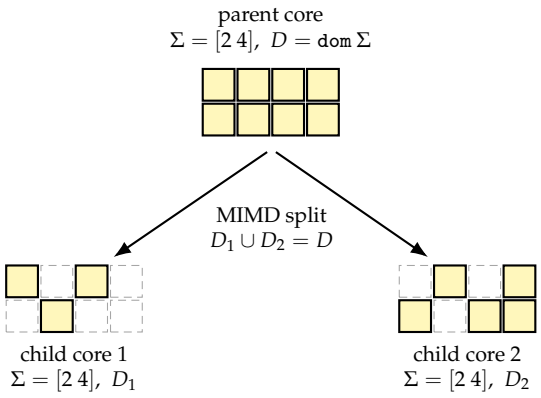

FIGURE 3.17: Parallel context versus domain across a two-way MIMD split. All three grids describe the iteration space of their corresponding cores. The parent passes down its parallel context Σ to its child cores as-is, but the domain D gets partitioned into D_1 and D_2. Each child addresses only a subset of the CEs that were available to its parent.

3.3.4 Core Identity

A core is identified by its core id, which stays fixed throughout the core's entire lifetime.

$$\begin{array}{rcl} id \in ProcId & := & Prim + (Lab \times BEnv) \\ cid \in CoreId & := & Seq(ProcId) \end{array}$$

FIGURE 3.18: Core identity.

A *procedure id* is either a primitive operator or a lambda label paired with a binding environment. Given a procedure, its identifier can be extracted with a `procId` helper. A *core id* is just a sequence of procedure ids. The top-level core id is an empty sequence $\bullet$.

When the machine encounters a MIMD split, it constructs child cores such that every child takes care of a SIMD subcomputation. The procedure assigned to a child during a split becomes a part of its identity: a child core id is constructed by appending the assigned procedure's id to the parent core's id.

This way, any standalone core id encodes its core's MIMD ancestry. Given two core ids, we can determine whether one core is an ancestor of the other by checking whether one is a prefix of the other.

Notice also that closure id includes a binding environment β. It is only necessary to distinguish sibling cores that got assigned different closures over the same lambda. When dealing with closures over different lambdas, β can be ignored.

Tree View on Cores

Strictly speaking, the id could be a random fresh string generated when a new core is spawned, but it is much more insightful to use *CoreId* as a condensed encoding of the core's MIMD ancestry. It enables us to think about the structural relationship between cores as a *prefix tree*, organized by core ids. An example of this view is given in Figure 3.15.

In this view, only the cores at the leaves of the tree can do work, while every interior node is suspended. This is because the most-recently-spawned child cores for a particular MIMD branch will always have the longest core ids and sit at the leaves. Every ancestor will be waiting for child cores to complete their subcomputations, and will only resume once all the child cores are done.

Another interesting consequence is that the prefix tree view on cores intuitively guarantees *CoreId* uniqueness. Since every new core id is constructed by appending a split-identifying procedure id to the parent core id, the new child core will always be inserted as a leaf in the prefix tree.

3.3.5 Core and How It Steps

$$core \in Core \quad := \quad State \times ParCtx \times Domain \times CoreId$$

FIGURE 3.19: Core state.

A *core* is a fundamental SIMD evaluator within the SIMD/MIMD Machine. It is represented as a 4-tuple; each of the components has been discussed in earlier subsections.

In short, *State* captures the current execution state of the core. *ParCtx* is the parallel context, which is the upper bound on the current SIMD parallelism, while *Domain* is the precise iteration domain of the core. Finally, *CoreId* is the fixed id of the core, which encodes the information about the core's MIMD ancestry.

We use notational sugar for cores: $\langle st,\ \Sigma,\ D \rangle^{cid}$. Here, the core id is written as a superscript, which is to show that the core's id is just an identifier and not a part of the core's dynamically changing state.

Core Step

Core execution states presented in Section 3.3.2 give rise to a transition system, where transitions are formalized with the $\overset{\text{Core}}{\Longrightarrow}$ stepping relation. It has the following signature:

$$M \vdash core \overset{\text{Core}}{\Longrightarrow} \langle core^{\#},\ ve_{\text{ext}} \rangle$$

This is a 3-way relation. Given machine state M (Section 3.3.6), a *core* picked from M steps to a pair. The first element of the pair is a set of cores $core^{\#}$ that contains the updated input core as well as, potentially, spawned child cores. The second element of the pair is a possibly empty value environment extension ve_{ext}.

The pair on the right-hand side of the arrow is set up specifically to communicate machine state updates. However, most of the core transitions do not result in any machine state updates, so the relation comes with minor notational sugar:

$$M \vdash core \overset{\text{Core}}{\Longrightarrow} core' \quad \text{desugars to} \quad M \vdash core \overset{\text{Core}}{\Longrightarrow} \langle \{core'\},\ \varnothing \rangle$$

Figure 3.20 depicts all the defined state transitions. Formalizing each will require a fair bit of math which deserves proper discussion, so Chapter 4 is devoted entirely to breaking down these state transitions. As will be demonstrated there, the relation is deterministic.

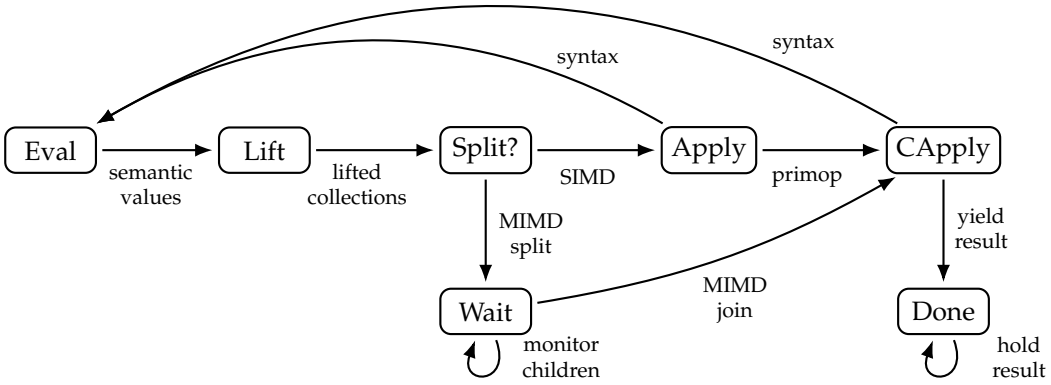

FIGURE 3.20: Core state transitions.

3.3.6 Machine and How It Steps

The machine state consists of a set of cores and a value environment *VEnv*.

$$M \in \mathit{Machine} \quad := \quad \mathit{Set}(\mathit{Core}) \times \mathit{VEnv}$$

FIGURE 3.21: Machine state.

The set of cores represents all the SIMD computations presently performed by the machine. The value environment *VEnv* holds all the bindings that any of the cores produced during their runtime. It is safe to keep the value environment as a part of global machine state because it only ever grows, as described in Section 3.2.6.

Initial and Final States

To talk about SIMD/MIMD Machine execution, it must have well-defined initial and final states.

$$\begin{aligned}
\varsigma_{\texttt{TOP}} \in \mathit{Proc} &::= \langle \varnothing,\ \mathit{lam}_{\texttt{TOP}} \rangle \\
\kappa_{\texttt{HALT}} \in \mathit{CVal} &::= \langle \texttt{yield}(\bullet),\ \bullet,\ \texttt{dom}\ \bullet \rangle \\
\mathit{core}_{\texttt{INI}} \in \mathit{Core} &::= \langle \mathrm{Apply}(\varsigma_{\texttt{TOP}},\ \bullet,\ \kappa_{\texttt{HALT}}),\ \bullet,\ \texttt{dom}\ \bullet \rangle^{\bullet} \\
M_{\texttt{INI}} \in \mathit{Machine} &::= \langle \{\mathit{core}_{\texttt{INI}}\},\ \varnothing \rangle
\end{aligned}$$

FIGURE 3.22: Initial top-level core and machine state.

As described in Section 2.2.2, the CPS-converted REMORA program will be a thunk $\mathit{lam}_{\texttt{TOP}}$. Its closure $\varsigma_{\texttt{TOP}}$ is initialized with an empty binding environment, since the top-level thunk does not close over anything. Since $\varsigma_{\texttt{TOP}}$ is a procedure, we know that calling it will always be SIMD, so the initial core $\mathit{core}_{\texttt{INI}}$ can be constructed in the *Apply* state. The supplied continuation value is $\kappa_{\texttt{HALT}}$, which is a `yield` continuation primitive tagged with the core's own id. Notice also that since $\mathit{core}_{\texttt{INI}}$ is the top-level core, its core id will be $\bullet$.

The initial parallel context is $\bullet$, since no lifting has taken place yet, and the initial domain is the domain of that context, `dom` $\bullet$ which is a singleton set containing the empty index $\{[\bullet]\}$.

The initial machine state $M_{\texttt{INI}}$, then, is just the $\mathit{core}_{\texttt{INI}}$ and an empty value environment.

$$\begin{aligned}
\mathit{core}_{\texttt{FIN}} \in \mathit{Core} &::= \langle \mathrm{Done}(\boxed{v},\ \bullet),\ \bullet,\ \texttt{dom}\ \bullet \rangle^{\bullet} \\
M_{\texttt{FIN}} \in \mathit{Machine} &::= \langle \{\mathit{core}_{\texttt{FIN}}\},\ \varnothing \rangle
\end{aligned}$$

FIGURE 3.23: Final machine state.

We know that the CPS REMORA program has completed its computation when the top-level core with id $\bullet$ has reached a *Done* state. The final machine state $M_{\texttt{FIN}}$, then, contains only the final core $\mathit{core}_{\texttt{FIN}}$ and the result of the machine is the result collection $\boxed{v}$.

Machine Step

A machine M can always take a step as defined by the $\xRightarrow{\text{Mach}}$ stepping relation.

MACHSTEP

$$\frac{\begin{array}{c} core_{\texttt{FIN}} \notin core^{\#}_{\texttt{A}} \qquad M = \langle core^{\#}_{\texttt{A}},\ ve\rangle \qquad \langle core^{\#}_{\texttt{I}},\ core^{\#}_{\texttt{R}}\rangle = \texttt{pickCores}\ core^{\#}_{\texttt{A}} \\ stepResults = \{\langle core^{\#}_i,\ ve_i\rangle \mid \forall core_i \in core^{\#}_{\texttt{R}},\ M \vdash core_i \xRightarrow{\text{Core}} \langle core^{\#}_i,\ ve_i\rangle\} \\ core^{\#}_{\texttt{S}} = \bigcup\{core^{\#}_i \mid \forall\langle core^{\#}_i,\ \_\rangle \in stepResults\} \\ ve_{\text{ext}} = \biguplus\{ve_i \mid \forall\langle \_,\ ve_i\rangle \in stepResults\} \end{array}}{\langle core^{\#}_{\texttt{A}},\ ve\rangle \xRightarrow{\text{Mach}} \langle core^{\#}_{\texttt{I}} \cup core^{\#}_{\texttt{S}},\ ve \uplus ve_{\text{ext}}\rangle}$$

FIGURE 3.24: Machine Step.

The MACHSTEP rule applies when the set of cores does not contain the final core $core_{\texttt{FIN}}$ defined above. Then, `pickCores` selects a subset of cores that will run during this step. Specifically, it partitions the set of all available cores $core^{\#}_{\texttt{A}}$ into two subsets: the idle cores $core^{\#}_{\texttt{I}}$ and running cores $core^{\#}_{\texttt{R}}$.

In a way, the SIMD/MIMD Machine semantics is parameterized by `pickCores`, since different implementations produce different scheduling behavior. Whatever the scheduler is, though, it must meet a fairness condition: child cores get priority up until they reach a *Done* state, which is then given to their immediate parent. This ensures that the active SIMD subcomputations make progress and then join up their results. A trivial unconstrained choice for `pickCores` is to pick every available core at each step. For the purposes of this presentation, we assume this implementation.

Every core in $core^{\#}_{\texttt{R}}$ will take a step in accordance with the $\xRightarrow{\text{Core}}$ stepping relation. See Section 3.3.5 for high-level discussion.

The next machine state is constructed by taking the union of idle cores $core^{\#}_{\texttt{I}}$ and stepped cores $core^{\#}_{\texttt{S}}$ and extending the global value environment with all the individual extensions.

MACHHALT

$$\frac{core_{\texttt{FIN}} \in core^{\#}}{\langle core^{\#},\ ve\rangle \xRightarrow{\text{Mach}} \langle\{core_{\texttt{FIN}}\},\ \varnothing\rangle}$$

FIGURE 3.25: Machine Halt.

The MACHHALT rule applies when the set of cores within the machine contains a final core $core_{\texttt{FIN}}$. This indicates that the top-level computation is done, so the machine transitions to a final machine state.

Allocating Fresh Contours

With the machine structure in place, we can define the set of binding contours that was introduced in Section 3.2.6. A contour is a natural number paired with the id of the core that allocated it:

$$\mathit{Contour} := \mathit{CoreId} \times \mathbb{N}$$

A contour is considered *spent* if it appears in the machine's value environment, so the set of contours that are no longer fresh can be determined from the machine state:

$$\begin{aligned} &\texttt{cns} : \mathit{Machine} \to \mathit{Set}(\mathit{Contour}) \\ &\texttt{cns}\ \langle \_,\ ve \rangle = \{cn \mid \langle cn,\ \_ \rangle \mapsto \_ \in ve\} \end{aligned}$$

A core performing a binding may only use contours outside this set, which is written as the freshness side condition $cn \notin \texttt{cns}\ M$, where the allocated contour is $cn = \langle cid,\ n \rangle$. Including the core id is necessary: during a machine step, multiple cores advance under the same M, so the freshness condition cannot prevent them from picking the same natural number n. Deduplication is achieved through the core id, which we know to be unique within the set of available cores. Put differently, a core needs to consult M only to avoid repeating *itself* when picking a contour.

Every rule that allocates a contour will use it to construct new bindings and communicate them to the machine through a value environment extension ve_{ext}. As a result, the chosen contours will enter the spent set in the next machine state.

Running the SIMD/MIMD Machine

Running the SIMD/MIMD Machine means starting it off in a well-defined initial state and repeatedly taking machine steps until it reaches a well-defined final one.

Although a formal proof is future work, the semantics of the SIMD/MIMD Machine presented here give us the following claim. For every top-level thunk $lam_{\texttt{TOP}}$ subject to CPS REMORA assumptions (Section 2.3), we can construct a corresponding initial core $core_{\texttt{INI}}$. It is then the case that

$$\langle \{core_{\texttt{INI}}\},\ \varnothing \rangle \xrightarrow{\text{Mach}}{}^{*}\ \langle \{core_{\texttt{FIN}}\},\ \varnothing \rangle$$

where $core_{\texttt{FIN}}$ is the final core in a $\text{Done}(\boxed{v},\ \bullet)$ state, and $\boxed{v}$ is the correct result of $lam_{\texttt{TOP}}$. Section 6.4 discusses how this claim could be formalized.

Chapter 4

Core Transitions

Chapter 3 left the core stepping relation $\xrightarrow{\text{Core}}$ specified only by Figure 3.20, which names the transitions a core can take from a given execution state. This chapter formalizes this stepping relation.

Every section tackles one execution state and the possible transitions that the core can take at that point. Together the transitions make the relation total: a core is always in exactly one of the seven states with just one applicable transition rule.

This chapter is organized as follows:

- Section 4.1 formalizes the *Eval* state transition, which converts syntactic expressions into the semantic values that the rest of the machine understands;
- Section 4.2 formalizes the *Lift* state transition, which lifts a procedure and its arguments to a shared principal frame;
- Section 4.3 formalizes the *Split*? state transitions, which decide whether a lifted application can proceed in SIMD mode on the current core or requires a MIMD split across newly spawned child cores;
- Section 4.4 formalizes the *Apply* state transitions, which execute a procedure application in SIMD mode;
- Section 4.5 formalizes the *CApply* state transitions, which invoke a continuation value with the result of an application;
- Section 4.6 formalizes the *Wait* state transitions, which synchronize a core with the children it spawned during a MIMD split;
- Section 4.7 formalizes the *Done* state transitions, which deliver a core's finished result back to its parent.

4.1 Eval: Syntax to Semantic Values

Recall that a core in the *Eval* state has the following shape:

$$\langle \text{Eval}(\beta,\ [\![(f\ e\ \ldots\ k)]\!]),\ \Sigma,\ D\rangle^{cid}$$

This state is tasked with converting syntactic expressions of a given program into semantic values that the rest of the machine understands. Specifically, every program in CPS REMORA is a function call of the form $(f\ e\ \ldots\ k)$, so *Eval* must produce (1) a procedure collection for function f, (2) a sequence of collections for argument expressions $[e \ldots]$, and (3) a continuation value that captures the core's current parallel context for continuation expression k.

A core in *Eval* state will always transition to a *Lift* state, where the core will perform rank-polymorphic lifting. Figure 4.1 gives a formal specification of the step.

4.1.1 Transition Rule

EVALCALL

$$\frac{\langle \_,\ ve\rangle = M \qquad \boxed{\rho} = \texttt{expEval}\ f\ \beta\ ve \qquad \boxed{v}^\star = [\texttt{expEval}\ e_a\ \beta\ ve \mid e_a \in e^\star] \qquad \kappa = \texttt{contEval}\ k\ \beta\ ve\ \Sigma\ D}{M \vdash \langle \text{Eval}(\beta,\ [\![(f\ e^\star\ k)]\!]),\ \Sigma,\ D\rangle^{cid} \overset{\text{Core}}{\Longrightarrow} \langle \text{Lift}(\boxed{\rho},\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D\rangle^{cid}}$$

FIGURE 4.1: Eval transition.

During the EVALCALL transition, function expression f, argument expressions $e^\star$ and a continuation expression k are evaluated with `expEval` and `contEval`. These syntactic evaluators are defined in the following subsection.

Both evaluators consume a binding environment β and the current value environment ve. These environments are necessary to resolve variable references that might occur in expressions. Note that the β comes directly from the *Eval* state tuple. One can think of it as evaluating a given program under some specific binding environment.

Also, notice that `contEval` consumes the current core's parallel context and domain. The produced continuation values are constructed with the evaluating core's dynamic state. This captures the idea that calling a continuation is the same as a function call return: the running core needs to revert its parallel context back to what it was when the continuation expression was first evaluated.

4.1.2 Syntax Evaluators

Function and Argument Expressions

Expression evaluator `expEval` is defined as:

$$\begin{aligned}
&\texttt{expEval} : \mathit{Exp} \to \mathit{BEnv} \to \mathit{VEnv} \to \mathit{Val} \\
&\texttt{expEval}\ [\![x]\!]\ \beta\ ve = ve\ \langle x,\ \beta\ (\texttt{binder}\ x)\rangle \\
&\texttt{expEval}\ [\![(\texttt{array}\ (d^\star)\ a^\star)]\!]\ _\ _ = \ulcorner\!\llcorner (\bullet \mid d^\star) \parallel m \urcorner\!\lrcorner
\end{aligned}$$

where

$$m = \texttt{makeIdxMap}\left\{\langle idx,\ \texttt{atomEval}\ a_{idx}\ \beta\rangle \ \mid\ \langle idx,\ a_{idx}\rangle \in \texttt{zip}\ (\texttt{domList}\ d^\star)\ a^\star\right\}$$

Let us break down the array literal case. Here, `domList` $d^\star$ is like `dom` $d^\star$, except that the output is not a set of indices spanning the iteration space specified by the dimension sequence, but instead it is a list, sorted in row-major order. Atoms in an array literal are also laid out in row-major order, so `zip` pairs up corresponding indices and atoms. For each index-atom pair, the atom is evaluated with `atomEval`, as given below:

$$\begin{aligned}
&\texttt{atomEval} : \mathit{Atom} \to \mathit{BEnv} \to \mathit{ValAtom} \\
&\texttt{atomEval}\ [\![\mathit{lam}]\!]\ \beta = \langle \beta,\ \mathit{lam}\rangle \\
&\texttt{atomEval}\ [\![\mathit{prim}]\!]\ _ = \mathit{prim} \\
&\texttt{atomEval}\ [\![b]\!]\ _ = b
\end{aligned}$$

Finally, `makeIdxMap` constructs an index map out of index-value pairs, and a new collection is assembled. The sequence of dimensions $d^\star$ in the array literal will become the new collection's cell shape and the sequence of frame dimensions will be empty. This is because a freshly evaluated array literal does not know anything about the parallel context in which it occurs. It is the job of the *Lift* state to resolve the parallel context for this collection.

Continuation Expressions

The corresponding evaluator `contEval` for continuation expressions requires the core's parallel context in addition to the binding environment β:

$$\begin{aligned}
&\texttt{contEval} : \mathit{CExp} \to \mathit{BEnv} \to \mathit{VEnv} \to \mathit{ParCtx} \to \mathit{Domain} \to \mathit{CVal} \\
&\texttt{contEval}\ [\![x_\mathrm{k}]\!]\ \beta\ ve\ _\ _ = ve\ \langle x_\mathrm{k},\ \beta\ (\texttt{binder}\ x_\mathrm{k})\rangle \\
&\texttt{contEval}\ [\![\mathit{clam}]\!]\ \beta\ _\ \Sigma\ D = \langle\langle \beta,\ \mathit{clam}\rangle,\ \Sigma,\ D\rangle
\end{aligned}$$

4.2 Lift: Implicitly Replicating Data

Recall that a core in the *Lift* state has the following shape:

$$\langle \text{Lift}(\boxed{\rho},\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D\rangle^{cid}$$

The *Lift* state prepares the machine for applying a procedure collection $\boxed{\rho}$ to a sequence of argument collections $\boxed{v}^\star$. The continuation value κ is simply passed along; a later state will invoke it with the result of this application once it has been computed.

This state performs lifting, which is the operation at the heart of REMORA's rank-polymorphic semantics (Section 4.2.2). Given the shape of the procedure collection, expected argument ranks and shapes of argument collections, the core will determine the principal frame of the application. Every collection participating in the call then gets implicitly replicated to match the shape of principal frame, and the core's dynamic state grows to match it too. Notice that the principal frame depends on available data, which is what makes REMORA's parallelism model data-driven.

After lifting, the core still has not determined whether this application can run in SIMD mode or requires a MIMD split. The core transitions to *Split*? state to examine the procedure collection and make that decision. Figure 4.2 gives the single rule that performs this transition.

4.2.1 Transition Rule

LIFT

$$\frac{\langle \boxed{\rho_{\text{L}}},\ \boxed{v_{\text{L}}}^\star,\ \sigma_{\text{pf}}\rangle = \texttt{lift}\ \boxed{\rho}\ \boxed{v}^\star\ \Sigma \qquad \Sigma' = \Sigma +\!\!+ \sigma_{\text{pf}} \qquad D' = D \ltimes (\texttt{dom}\ \sigma_{\text{pf}})}{M \vdash \langle \text{Lift}(\boxed{\rho},\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D\rangle^{cid} \xrightarrow{\text{Core}} \langle \text{Split?}(\boxed{\rho_{\text{L}}},\ \boxed{v_{\text{L}}}^\star,\ \kappa),\ \Sigma',\ D'\rangle^{cid}}$$

FIGURE 4.2: Lifting.

All collection values participating in the call get lifted with the `lift` function, as defined in Section 4.2.2. Specifically, $\boxed{\rho_{\text{L}}}$ is the lifted procedure collection and $\boxed{v_{\text{L}}}^\star$ is the sequence of lifted argument collections. The function also returns the principal frame σ_{pf} of the application, which is used to update the dynamic state of the executing core.

The updated parallel context Σ' is obtained by appending the principal frame σ_{pf} to the existing parallel context Σ. This is because data-driven lifting will only ever increase parallelism on application, if the principal frame is non-empty. The updated core domain D' is the cross product of the current domain D and of the iteration space specified by the sequence of dimensions in σ_{pf}. This domain extension corresponds to increasing the number of CEs available to the core.

The core then transitions to *Split*?, carrying the lifted procedure collection $\boxed{\rho_{\text{L}}}$ and lifted arguments $\boxed{v_{\text{L}}}^\star$ forward under the newly extended Σ' and D'. The continuation value κ is passed through unchanged.

4.2.2 Lifting

Lifting is the mechanism by which REMORA's rank-polymorphic semantics is realized. It describes how data gets replicated during procedure application.

In direct-style REMORA, the process goes as follows:

- **Identify frame and cell dimensions for the procedure and arguments.** The array in the function position can have an arbitrary frame but will always have scalar cells, so the dimensions making up the array shape will become its frame dimensions. For argument arrays, the frame and cell dimensions are identified based on the rank of the corresponding parameter in the procedure signature. The parameter's rank r specifies expected cell rank; then, we can find frame and cell dimensions of an argument by splitting its shape at the rth dimension counting from the right.
- **Compute the principal frame.** When lifting, the array with the longest frame will dictate the shape of the computation. More precisely, the maximum of all frames under prefix ordering (i.e. $[2\ 3] \sqsubseteq [2\ 3\ 4]$) will become this application's *principal frame*.
- **Lift the function and argument arrays.** The goal is to make everyone's frame match the principal frame of the application. To do so, every array with a frame shorter than the principal frame needs to replicate its cells. Notice that since the frame dimensions maintain prefix ordering, we only ever need to extend the frame dimensions to the *right*.

In CPS REMORA the process looks very similar, but with a few modifications. First, following the goal of delaying replication of data for as long as possible, lifting will never actually copy cells (Section 3.2.5). Instead, the frame dimensions that were introduced to the collection shape as the result of lifting will simply be annotated with an underline (Section 3.2.5). Second, since the parallel context of the core is tracked explicitly, the computed principal frame will also extend the core's parallel context.

The lifting operation, as described above, is performed by `lift`.

$$\texttt{lift} : \mathit{Val} \to \mathit{Seq}(\mathit{Val}) \to \mathit{ParCtx} \to \mathit{Val} \times \mathit{Seq}(\mathit{Val}) \times \mathit{ParCtx}$$
$$\texttt{lift}\ \boxed{\rho}\ \boxed{v}^\star\ \Sigma = \langle \boxed{\rho_{\mathrm{L}}}, \boxed{v_{\mathrm{L}}}^\star, \sigma_{\mathrm{pf}} \rangle$$

The function takes the collection in the function position, a sequence of argument collections and the core's current parallel context. It returns the input collections post-lifting and the principal frame of the current application. The rest of this subsection will describe, in detail, how the return values are constructed.

Core's view on lifting

First, collections are unpacked to get their original (superscripted with "o") frame and cell dimensions.

$$\lceil\!\lfloor (\psi_\rho^o \mid \omega_\rho^o) \,\|\!|\, m_\rho \,\rfloor\!\rceil = \boxed{\rho}$$
$$\lceil\!\lfloor (\psi_i^o \mid \omega_i^o) \parallel m_i \,\rfloor\!\rceil = \boxed{v}_i \quad \forall\, \boxed{v}_i \in \boxed{v}^\star$$

Starting here and until the end of the lifting subsection, we use "i" to refer to the ith argument and any intermediate value computed for it.

Since collection shapes capture the shape of the data at the current point in execution, frame dimensions ψ^o and cell dimensions ω^o describe the core's view of the values *before* any lifting for this function application took place. ψ^o describes the shape of SIMD compute elements executing in lockstep, each of which is performing this function application. Meanwhile, ω^o describes the shape of the data that a given compute element sees, which is precisely the data that participates in principal frame negotiation. In other words, principal frame negotiation will happen in terms of the shapes given by ω^o.

Expected argument ranks

Expected argument ranks are necessary to determine frame and cell dimensions during lifting. Since the transformed CPS REMORA typechecks, it is the case that every atom in $\boxed{\rho}$ is a procedure with the same signature. Even if the procedures differ, as would be the case during a MIMD split, each must still consume the same number of arguments and expect the same ranks.

Then, to compute a sequence of expected argument ranks $r^\star$, pick any procedure atom ρ_{any} from the procedure collection $\boxed{\rho}$ and extract the signature with `argRanks`, as defined below. The given definition states expected argument ranks for all defined primitive operators, but can be easily extended to support more operators.

$$
\begin{aligned}
&\texttt{argRanks} : \mathit{Proc} \to \mathit{Seq}(\mathit{Rank}) \\
&\texttt{argRanks}\ [\![(\texttt{fn}_l\ [(x_0\ r_0) \ldots (x_n\ r_n)]\ x_{\text{k}}\ .\ \mathit{call})]\!] = [r_0 \ldots r_n] \\
&\texttt{argRanks}\ [\![\texttt{select}]\!] = [0\ \texttt{all}\ \texttt{all}] \\
&\texttt{argRanks}\ [\![\texttt{zero?}]\!] = [0] \\
&\texttt{argRanks}\ [\![\texttt{id}]\!] = [\texttt{all}] \\
&\texttt{argRanks}\ [\![+]\!] = [0\ 0] \\
&\texttt{argRanks}\ [\![-]\!] = [0\ 0] \\
&\texttt{argRanks}\ [\![\times]\!] = [0\ 0]
\end{aligned}
$$

Note that the length of expected argument ranks $r^\star$ must be the same as the length of argument collections $\boxed{v}^\star$. We write r_i to refer to ith expected argument rank.

Identifying frames and cells

Given the procedure's expected argument ranks, we compute frames and cells of argument collections that will drive the principal frame negotiation. Recall that the procedure collection will always have a scalar cell. Let

$$
\begin{aligned}
&\langle \psi_\rho,\ \omega_\rho \rangle = \langle \omega^o_\rho,\ \bullet \rangle \\
&\langle \psi_i,\ \omega_i \rangle = \texttt{splitAtRank}\ r_i\ \omega^o_i
\end{aligned}
$$

where the `splitRight` is given as:

$$\begin{aligned}
&\texttt{splitAtRank} : \mathit{Rank} \to \mathit{CollDims} \to \mathit{CollDims} \times \mathit{CollDims} \\
&\texttt{splitAtRank all } \delta^\star = \langle \bullet,\ \delta^\star \rangle \\
&\texttt{splitAtRank } n\ \delta^\star = \texttt{splitRight } n\ \delta^\star
\end{aligned}$$

Computing principal frame

The next step is to compute the principal frame. The source program's typechecking guarantees that prefix ordering (i.e. $[2\ 3] \sqsubseteq [2\ 3\ 4]$) will hold for all the frames at this point, so we can simply perform a lattice join to get the principal frame σ_{pf}:

$$\sigma_{\text{pf}} = \left(\bigsqcup_i \psi_i \right) \sqcup \psi_\rho$$

Here $\sqcup$ is just a partial join on dimension sequences. Notably, only dimension sizes are comparable, so underlines of replicated dimensions will be ignored. As a result, σ_{pf} will be a sequence of *Dim*s and the longest frame of the application.

"Replicating" data

Now, the procedure and argument collections can be lifted to match the principal frame. By the semantics of rank polymorphism, any dimensions that a frame needs to extend to will be replicated.

Recall that a *CollDim* might be a *replicated dimension*, denoted with an underline, like $\underline{d}$. This is precisely the context when it is useful: we can annotate a dimension with an underline without actually replicating the data, so that the underlying index map remains unchanged.

$$\begin{aligned}
\psi_\rho^{\text{pf}} &= \psi_\rho +\!\!+ \texttt{addUnderlines}\ (\texttt{drop}\ (\texttt{rank}\ \psi_\rho)\ \sigma_{\text{pf}}) \\
\psi_i^{\text{pf}} &= \psi_i +\!\!+ \texttt{addUnderlines}\ (\texttt{drop}\ (\texttt{rank}\ \psi_i)\ \sigma_{\text{pf}})
\end{aligned}$$

Here, a helper function `addUnderlines` is used to underline every dimension in the sequence.

Handling pre-existing parallelism

In direct-style REMORA this would complete lifting. However, the pre-existing parallel context ψ^o still needs to be dealt with. It goes through a mostly identical process of determining a parallel context principal frame and then extending all the original frames to the same shape. The only difference is that this time the core's current parallel context Σ (which is itself just a sequence of dimensions) now participates in principal frame negotiation:

$$\sigma_{\text{pf}}^{\text{ctx}} = \left(\bigsqcup_i \psi_i^o \right) \sqcup \psi_\rho^o \sqcup \Sigma$$

The original frame dimensions get extended the same way.

$$\psi_\rho^{\text{ctx}} = \psi_\rho^o +\!\!+ \texttt{addUnderlines}\ (\texttt{drop}\ (\texttt{rank}\ \psi_\rho^o)\ \sigma_{\text{pf}}^{\text{ctx}})$$
$$\psi_i^{\text{ctx}} = \psi_i^o +\!\!+ \texttt{addUnderlines}\ (\texttt{drop}\ (\texttt{rank}\ \psi_i^o)\ \sigma_{\text{pf}}^{\text{ctx}})$$

Putting it all together

Finally, the lifted collections can be constructed as follows:

$$\psi'_\rho = \psi_\rho^{\text{ctx}} +\!\!+ \psi_\rho^{\text{pf}} \qquad \psi'_i = \psi_i^{\text{ctx}} +\!\!+ \psi_i^{\text{pf}}$$
$$\boxed{\rho_{\text{L}}} = \ulcorner\!\llcorner (\psi'_\rho \mid \bullet) \,\Big\|\, m_\rho \,\urcorner\!\lrcorner \qquad \boxed{v_{\text{L}}}_i = \ulcorner\!\llcorner (\psi'_i \mid \omega_i) \parallel m_i \,\urcorner\!\lrcorner$$

Notice that $\boxed{\rho_{\text{L}}}$ always ends up with empty cell dimensions: every one of the function collection's own dimensions has been folded into its frame.

An interesting thing to note is that lifting moves what were originally per-compute-element cell dimensions into frame dimensions. Every time a function consumes an argument cell, its resulting ω' rank will be at most the size of the original ω^o. That is, on application the overall parallelism of the computation will grow. This process is mirrored by function call returns, that is, when a continuation value gets invoked (Section 4.5).

4.3 Split?: Deciding Between SIMD and MIMD

Recall that a core in the *Split*? state has the following shape:

$$\langle \text{Split?}(\boxed{\rho},\ \boxed{v}^{\star},\ \kappa),\ \Sigma,\ D\rangle^{cid}$$

When a core reaches this state, *Lift* has already lifted $\boxed{\rho}$ and $\boxed{v}^{\star}$ to share the application's principal frame. However, to compute this procedure application the core needs to first decide whether it can run in SIMD mode.

Recall that cores only execute SIMD computations, so the procedure collection can only be applied element-wise if it keeps all the core's CEs in lockstep. The SIMD/MIMD Machine enforces this: the *Apply* state that actually executes procedure application takes a *procedure atom* and not a procedure collection. This exactly encodes the "single-instruction" part of SIMD. In other words, this core will only transition into *Apply* if every atom in $\boxed{\rho}$ is the same procedure (Section 4.3.2). Otherwise, if the collection contains more than one distinct procedure, the core has to extract each SIMD subcomputation, spawn child cores and transition into a *Wait* state.

These transitions are formalized in Figure 4.3.

4.3.1 Transition Rules

PROCEEDSIMD

$$\frac{\{\langle \rho,\ \_\rangle\} = \texttt{extractSIMD}\ \boxed{\rho}\ D}{M \vdash \langle \text{Split?}(\boxed{\rho},\ \boxed{v}^{\star},\ \kappa),\ \Sigma,\ D\rangle^{cid} \xRightarrow{\text{Core}} \langle \text{Apply}(\rho,\ \boxed{v}^{\star},\ \kappa),\ \Sigma,\ D\rangle^{cid}}$$

SPLITMIMD

$$\frac{\begin{gathered}\{s_0,\ s_1 \ldots\} = \texttt{extractSIMD}\ \boxed{\rho}\ D \\ core^{\#}_{\mathsf{c}} = \{\texttt{initChild}\ s\ \boxed{v}^{\star}\ \Sigma\ cid \mid \forall s \in \{s_0,\ s_1 \ldots\}\} \\ cid^{\#}_{\mathsf{c}} = \{cid_{\mathsf{c}} \mid \forall \langle \_,\ \_,\ \_\rangle^{cid_{\mathsf{c}}} \in core^{\star}_{\mathsf{c}}\}\end{gathered}}{M \vdash \langle \text{Split?}(\boxed{\rho},\ \boxed{v}^{\star},\ \kappa),\ \Sigma,\ D\rangle^{cid} \xRightarrow{\text{Core}} \langle\langle \text{Wait}(cid^{\#}_{\mathsf{c}},\ \kappa),\ \Sigma,\ D\rangle^{cid} \cup core^{\#}_{\mathsf{c}},\ \varnothing\rangle}$$

FIGURE 4.3: Core checks for a MIMD split.

PROCEEDSIMD and SPLITMIMD both start by calling `extractSIMD` on the already-lifted procedure collection $\boxed{\rho}$, as defined in Section 4.3.2. The function examines the atoms in the collection and partitions the core's domain D based on how many distinct procedures are present in $\boxed{\rho}$. The result is a set of procedure-subdomain pairs that `extractSIMD` was able to identify.

PROCEEDSIMD applies when the result of `extractSIMD` is a singleton procedure-subdomain pair. This means that every CE in the core executes the same procedure in

lockstep. As a result, the core can simply transition into the SIMD *Apply* state, carrying the already-lifted argument collections and the single procedure atom forward.

Otherwise, SPLITMIMD applies. To handle the MIMD split, the current core must spawn child cores to take care of each SIMD subcomputation s identified by `extractSIMD`.

Each SIMD subcomputation s becomes one child core via the `initChild` constructor (Section 4.3.3), giving the set of child cores $core_C^\#$ and their ids $cid_C^\#$, which together define the *Wait* state the parent core transitions to.

Finally, the right-hand side of the $\xRightarrow{\text{Core}}$ in SPLITMIMD is specified in its complete, desugared form (Section 3.3.5). It is a pair representing a machine state update. The first element of the pair is the set of cores that will replace the original core. Here, it is a set of child cores $core_C^\#$ unioned with the updated parent core, which will transition to a *Wait* state. The second element of the pair is the value environment extension, which in this case is empty.

4.3.2 Identifying MIMD Splits

After lifting, every collection participating in the application shares the same frame shape, that is, the application's principal frame. The lifted procedure collection will also have a scalar cell, with all of its original cell dimensions folded into the frame. Additionally, the core's domain now addresses all the compute elements introduced by principal frame lifting.

As a result, every index in the core's updated domain—equivalently, every CE in the core—corresponds to some procedure atom in $\boxed{\rho}$, which that CE will apply. The CEs stay in lockstep as long as all of those atoms are the same procedure, no matter how the argument data differs. Otherwise the application is MIMD, and the SIMD core can no longer carry it out.

Generally speaking, a SIMD computation can be seen as a trivial case of MIMD with just one SIMD subcomputation. As a result, we say that a *MIMD split* occurs when it is possible to extract more than one SIMD subcomputation from the current procedure collection application. Otherwise, the application can be executed by the current core in SIMD mode.

Distinct procedures

A lot of SIMD/MIMD intuition relies on the notion of executing procedures being the "same" or "different". In the presence of lambdas, it is a tricky question due to observational equivalence. For our purposes, however, it is sufficient to use a heuristic.

Two procedures are considered the same instruction only when they are equal as semantic values. Specifically, two primops are equal when they are the same primop; two closures are equal when they close over the same lambda under the same binding environment.

This same-environment requirement might appear too strict at first. A collection of closures seemingly closes over different data at each index, which would suggest that every index deserves its own binding environment. However, recall that in the SIMD/MIMD Machine a variable is always bound to an entire collection at once, not to one index's atom in isolation. So a closure's binding environment β only records *which*

collections it closes over, not what any single CE will read from them. Two closures that agree on β and *lam* therefore still count as the same instruction.

To identify all the distinct procedures within a collection, we define `distinctProcs` that simply extracts the set of elements present in the index map, up to our notion of procedure equality.

$$\texttt{distinctProcs} : \mathit{Collect\ Proc} \to \mathit{Set}(\mathit{Proc})$$
$$\texttt{distinctProcs}\ \ulcorner\!\llcorner \phi \parallel m \urcorner\!\lrcorner = \texttt{getElemSet}\ m$$

Extracting SIMD subcomputations

There are a number of valid ways to extract SIMD subcomputations from a possibly-MIMD procedure collection application. The naive answer is to treat every individual CE and its corresponding procedure as a standalone SIMD subcomputation running on its own core. This is suboptimal, of course, since all the hard-earned core-level SIMD parallelism gets translated into machine-level MIMD parallelism.

A better way is to group together all CEs executing the same procedure. Then, extracting all SIMD subcomputations will partition the current core's domain into subdomains, each corresponding to a distinct procedure atom present in the procedure collection.

We represent a SIMD subcomputation as a procedure atom paired with a subdomain that executes it. `extractSIMD` computes the complete set of procedure-subdomain pairs for a lifted procedure collection restricted to the core's domain:

$$\texttt{extractSIMD} : \mathit{Collect\ Proc} \to \mathit{Domain} \to \mathit{Set}(\mathit{Proc} \times \mathit{Domain})$$
$$\texttt{extractSIMD}\ \boxed{\rho}\ D = \{\langle \rho_i,\ D_i \rangle \mid \forall \rho_i \in \texttt{distinctProcs}\ (\boxed{\rho} \downarrow D)\}$$

where subdomain D_i contains all the indices of D that correspond to CEs executing with some ρ_i:

$$D_i = \{idx \mid \forall idx \in D,\ \boxed{\rho}\ @\ idx = \boxed{\bullet \parallel \rho_i}\}$$

Recall that $\boxed{\rho}$ must have a scalar cell. Then, every index of D maps to exactly one procedure atom in $\boxed{\rho}$ and every distinct procedure is used to extract a subdomain. It follows that `extractSIMD` indeed partitions D: every compute element will get assigned to exactly one subcomputation. Interestingly, ISPC dynamically arrives at an analogous partition through a varying function pointer (Section 1.1.2).

Notice also that if `distinctProcs` $\boxed{\rho}$ is a singleton, then we end up with the trivial, "exactly one SIMD subcomputation" case. The result of `extractSIMD` will also be a singleton pair containing the one procedure and the entire core's domain. This is the case when the PROCEEDSIMD rule applies and the core transitions into an *Apply* state.

4.3.3 Initializing Child Cores

Given a SIMD subcomputation of the form $\langle \rho,\ D \rangle$ produced by `extractSIMD`, the current core needs to initialize a new child to take care of it. `initChild` is a helper constructor that does exactly that:

$$\texttt{initChild} : (\mathit{Proc} \times \mathit{Domain}) \to \mathit{Seq}(\mathit{Val}) \to \mathit{ParCtx} \to \mathit{CoreId} \to \mathit{Core}$$
$$\texttt{initChild}\ \langle \rho,\ D \rangle\ \boxed{v}^{\star}\ \Sigma\ \mathit{cid} = \langle \mathrm{Apply}(\rho,\ \boxed{v}^{\star},\ \kappa),\ \Sigma,\ D \rangle^{\mathit{cid}_{\mathrm{C}}}$$

where

$$\kappa = \langle \texttt{yield}(\mathit{cid}),\ \Sigma,\ D \rangle$$
$$\mathit{cid}_{\mathrm{C}} = \mathit{cid} +\!\!+ \texttt{procId}\ \rho$$

The new core starts directly in *Apply* state, already carrying the lifted argument collections $\boxed{v}^{\star}$ and a single procedure ρ. Its dynamic state reuses the parent's lifted Σ, together with only the subdomain D of the subcomputation. Every sibling core gets this same Σ but a different, disjoint D, which is exactly what lets them run over disjoint pieces of the same principal frame.

The child's continuation value is a `yield` tagged with the current core's id *cid*. To the new child core, this is its *parent's* id. This is the communication link that allows the child to report its result fragment back up to the parent (Section 4.5).

Finally, the child's id $\mathit{cid}_{\mathrm{C}}$ is constructed by extending the parent's id with the id of its assigned procedure (Section 3.3.4).

4.4 Apply: Executing in SIMD Mode

Recall that a core in the *Apply* state has the following shape:

$$\langle \text{Apply}(\rho,\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D\rangle^{cid}$$

where ρ is either a closure or a primitive.

By the time a core reaches an *Apply* state, all the necessary setup for a CPS REMORA function application has been taken care of. In particular, every argument collection in $\boxed{v}^\star$ has been lifted to a shared principal frame and ρ has been identified as the instruction that every compute element addressed by D will be executing at this step. The result of the computation will always be passed along to the continuation value κ.

A core can end up in this state either because the original call was already a SIMD computation, or because this core is one of the children freshly spawned as a result of a MIMD split (Section 4.3.2). Either way, *Apply* can treat ρ as a single instruction executed simultaneously by every compute element that this core owns.

What "executing" ρ means depends on what kind of procedure it is. A closure ς carries a CPS REMORA program, so applying it just means binding its parameters to the already-lifted arguments and handing the body back to *Eval* to keep interpreting syntax. A primitive *prim* computes a result at every index of the domain and assembles those results into one collection. Figure 4.4 and Figure 4.5 formalize each of these transitions.

4.4.1 Transition Rules

APPLYCLO

$$\frac{\begin{array}{c}\langle \beta,\ [\![(\texttt{fn}_l\ [(x_0\ r_0)\ldots(x_n\ r_n)]\ x_\text{k}\ .\ call)]\!]\rangle = \varsigma \qquad x^\star = [x_0\ldots x_n] \\ cn = \langle cid,\ n\rangle \qquad cn \notin \texttt{cns}\ M \qquad \beta' = \beta[l \mapsto cn] \\ ve_\text{ext} = [\langle cn,\ x_\text{k}\rangle \mapsto \kappa] \uplus [\langle cn,\ x\rangle \mapsto \boxed{v} \mid \langle x,\ \boxed{v}\rangle \in \texttt{zip}\ x^\star\ \boxed{v}^\star]\end{array}}{M \vdash \langle \text{Apply}(\varsigma,\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D\rangle^{cid} \overset{\text{Core}}{\Longrightarrow} \langle \{\langle \text{Eval}(\beta',\ call),\ \Sigma,\ D\rangle^{cid}\},\ ve_\text{ext}\rangle}$$

FIGURE 4.4: Closure application.

The APPLYCLO rule runs when the procedure in the *Apply* state is a closure. Here, the only effect of executing this SIMD procedure is value binding.

First, a binding contour *cn* is picked. It is tagged with this core's id *cid* and must not already be spent in M, which is what makes it fresh. The closure's binding environment β gets extended with a new mapping from the lambda's label l to *cn*. The extended binding environment β' will be passed along to the *Eval* state that this core transitions to.

To extend the global value environment with bindings, this core step must communicate a value environment extension ve_ext back to the machine. In particular, the continuation parameter x_k is bound to the call's continuation value κ, and each of the parameters $[x_0\ldots x_n]$ is bound, in order, to corresponding lifted argument collections in $\boxed{v}^\star$.

The core steps to *Eval* state where it will evaluate the closure's body *call* under the extended binding environment β'.

APPLYPRIM

$$\frac{\boxed{res} = \texttt{assemble}\ \Sigma\ \left\{ \langle idx,\ \boxed{\boxed{res}_{idx}} \rangle \mid \forall idx \in D,\ \langle prim,\ \boxed{v}^\star,\ idx \rangle \overset{\text{CE}}{\Longrightarrow} \boxed{\boxed{res}_{idx}} \right\}}{M \vdash \langle \text{Apply}(prim,\ \boxed{v}^\star,\ \kappa),\ \Sigma,\ D \rangle^{cid} \overset{\text{Core}}{\Longrightarrow} \langle \text{CApply}(\kappa,\ \boxed{res}),\ \Sigma,\ D \rangle^{cid}}$$

EXECCE

$$\frac{\boxed{\boxed{res}_{idx}} = \texttt{execPrim}\ prim\ \left[\boxed{v}\ @\ idx \mid \boxed{v} \in \boxed{v}^\star \right]}{\langle prim,\ \boxed{v}^\star,\ idx \rangle \overset{\text{CE}}{\Longrightarrow} \boxed{\boxed{res}_{idx}}}$$

FIGURE 4.5: Primitive operation application.

The APPLYPRIM rule runs when the procedure in the *Apply* state is a primitive. This is the only place where compute elements actually perform work and produce result data.

For every index *idx* in the core's domain D, the corresponding compute element will execute the primitive operation *prim* over the data visible to it. The CE step is defined by the EXECCE rule, where the stepping relation is denoted with $\overset{\text{CE}}{\Longrightarrow}$.

A CE step relates a triple—the primitive being run, a sequence of argument collections $\boxed{v}^\star$, and the compute element's index *idx*—to a resulting cell value $\boxed{\boxed{res}_{idx}}$. During the step, every argument collection gets indexed by *idx* which produces a sequence of cells for each argument. Recall that a cell is *not* a collection (Section 3.2.5), as it only represents the data visible to this specific CE. The compute element then invokes `execPrim` (Section 4.4.2) to compute a result cell.

Back in the APPLYPRIM rule, the core collects all the per-CE result cells and passes them off to `assemble` (Section 4.4.3). This function is responsible for taking every individual result cell and putting it back together into a collection value $\boxed{res}$ that the core can continue with. The frame dimensions of the result will come from the core's current parallel context Σ.

The core then transitions to a *CApply* state, invoking the continuation value κ with the freshly assembled result.

4.4.2 Executing Primops Inside Compute Elements

The ρ that appears in $\text{Apply}(\rho,\ \boxed{v}^\star,\ \kappa)$ describes an instruction that every compute element inside of the core will perform in lockstep. From a core's point of view, this instruction is performed over entire argument *collections*, but this is just an abstraction. A more precise description is that every compute element will execute this instruction over whatever collection *cells* are visible to it.

Recall that a collection shape $(\psi \mid \omega)$ consists of frame dimensions ψ and cell dimensions ω. The frame describes the shape of parallelism (i.e. how many CEs can access data from this collection at the same time), while the cell tells us what each compute element sees. Indexing a collection, exactly as the EXECCE rule does, picks out a specific cell at the given index. All of the ψ dimensions will be used to resolve an index, leaving behind an ω-shaped piece of data, which is exactly what a cell value is.

Since `execPrim` is run by compute elements, it only operates on cell values and produces cell value results. For each of the defined primitive operators, the CEs do the following:

$$\texttt{execPrim} : \mathit{Prim} \to \mathit{Seq}(\mathit{CellVal}) \to \mathit{CellVal}$$

$$\texttt{execPrim}\ [\![\texttt{id}]\!]\ \left[\boxed{\boxed{v}}\right] = \boxed{\boxed{v}}$$

$$\texttt{execPrim}\ [\![\texttt{zero?}]\!]\ \left[\boxed{\bullet \parallel z}\right] = \boxed{\bullet \parallel \texttt{\#t}} \quad (z = 0)$$

$$\texttt{execPrim}\ [\![\texttt{zero?}]\!]\ \left[\boxed{\bullet \parallel z}\right] = \boxed{\bullet \parallel \texttt{\#f}} \quad (z \neq 0)$$

$$\texttt{execPrim}\ [\![\texttt{select}]\!]\ \left[\boxed{\bullet \parallel \texttt{\#t}}, \boxed{\boxed{v}}_{\mathrm{t}}, \boxed{\boxed{v}}_{\mathrm{f}}\right] = \boxed{\boxed{v}}_{\mathrm{t}}$$

$$\texttt{execPrim}\ [\![\texttt{select}]\!]\ \left[\boxed{\bullet \parallel \texttt{\#f}}, \boxed{\boxed{v}}_{\mathrm{t}}, \boxed{\boxed{v}}_{\mathrm{f}}\right] = \boxed{\boxed{v}}_{\mathrm{f}}$$

$$\texttt{execPrim}\ [\![+]\!]\ \left[\boxed{\bullet \parallel z_1}, \boxed{\bullet \parallel z_2}\right] = \boxed{\bullet \parallel z_1 + z_2}$$

$$\texttt{execPrim}\ [\![-]\!]\ \left[\boxed{\bullet \parallel z_1}, \boxed{\bullet \parallel z_2}\right] = \boxed{\bullet \parallel z_1 - z_2}$$

$$\texttt{execPrim}\ [\![\times]\!]\ \left[\boxed{\bullet \parallel z_1}, \boxed{\bullet \parallel z_2}\right] = \boxed{\bullet \parallel z_1 \times z_2}$$

Most of the primitives are self-explanatory: `id` is the identity, `zero?` is a zero predicate and $+$, $-$, $\times$ are arithmetic primitives. The interesting one is `select`, as it is a conditional primitive. If the guard is a scalar cell containing `#t`, then the output is $\boxed{\boxed{v}}_{\mathrm{t}}$; otherwise, it is $\boxed{\boxed{v}}_{\mathrm{f}}$. CPS REMORA uses this primop to encode if-else conditionals.

Notably, the function is only defined for well-formed argument values. Since we assume that the top-level program typechecks, it will never be the case that, for example, we try to add a boolean to a number. More interesting, however, is that these primitives also expect specific cell shapes. These are precisely the procedure signatures specified by `argRanks`, as discussed in Section 4.2.2. Since lifting has already happened, we know that argument collections will always have appropriate cell shapes at this point.

The mechanism described here works only for first-order primitives (Section 2.3). `execPrim` operates over cell values, so a compute element only ever handles passive data. A higher-order primitive like `reduce` cannot be handled in this way because it requires us to apply the operator argument over argument data. SIMD/MIMD Machine CEs do not model such transfers of control, so an alternative mechanism must be developed. Section 6.1 sketches out the necessary extensions to the machine.

4.4.3 Assembling Cells Into Collections

Once every compute element has produced its result cell, it is the responsibility of the core to assemble these pieces back into a collection. `assemble` takes a set of all per-index CE contributions and the core's parallel context to produce the overall result:

$$\texttt{assemble} : \mathit{CollDims} \to \mathit{Set}(\mathit{Index} \times \mathit{CellVal}) \to \mathit{Val}$$

$$\texttt{assemble}\ \psi\ \{\langle idx_i,\ \boxed{\boxed{v}_i}\rangle\} = \ulcorner\!\llcorner (\psi \mid \omega) \Big\| (\texttt{cellsToIdxMap}\ \{\langle idx_i,\ \boxed{\boxed{v}_i}\rangle\})\ \urcorner\!\lrcorner$$

where $\forall i, \boxed{\boxed{v}_i} = \boxed{\omega \parallel \_}$, i.e. each cell has the shape ω.

This is just a thin wrapper. The underlying index map for the output collection is constructed directly from the set of index-cell pairs. This is why index maps need to support partial data: if D of the current core was a subdomain as a result of a MIMD split, the set of index-cell pairs could never be assembled into a regular array.

The frame dimensions come from the core's parallel context Σ, since that dimension sequence is precisely what describes the shape of CE parallelism during this application. A sequence of dimensions making up the parallel context is a valid sequence of *CollDim*. The cell dimensions come from the result cells themselves: since every CE executes the same primitive, their output will share the same shape ω.

Notice also that no frame dimensions are underlined at this point. Since the primitive ran at every index of D and every CE computed real data, there is no implicit replication anymore.

4.5 CApply: Invoking Continuations

Recall that a core in the *CApply* state has the following shape:

$$\langle \text{CApply}(\langle \rho_\kappa,\ \Sigma_\kappa,\ D_\kappa \rangle,\ \boxed{v}),\ \Sigma,\ D \rangle^{cid}$$

where ρ_κ is either a continuation closure ς_κ or a `yield` primitive.

This state is for control transfers. Here, $\boxed{v}$ is the result some procedure just computed, and $\langle \rho_\kappa,\ \Sigma_\kappa,\ D_\kappa \rangle$ is the invoked continuation. The continuation value is a triple that contains a continuation procedure ρ_κ as well as a snapshot of the dynamic state of the core that first constructed this continuation. This is necessary because parallel context Σ_κ and domain D_κ are a crucial part of the core state, and it would be impossible to resume a continuation without reinstating them. This is also how the SIMD/MIMD Machine models function call returns in a parallel setting. Lifting implicitly replicates data on calls, which is how parallelism increases. This is why the *Lift* state only ever grows Σ and D. However, after a procedure application has computed a result, the parallelism will decrease on return. Reinstating previous dynamic state is exactly how the core will achieve this.

When ρ_κ is a continuation closure ς_κ, resuming it means binding its single parameter to the (rank-adjusted) result and handing its body to an *Eval* state. This is similar to a closure application, except that no lifting takes place. This transition is formalized in the ApplyCont rule in Figure 4.6.

ρ_κ can also be a $\texttt{yield}(cid_\text{P})$ primitive. This is a special continuation that indicates that a parent core with id cid_P is waiting on this result. Invoking this continuation means that the current core will prepare the value $\boxed{v}$ for a handover and transition into a *Done* state. This transition is formalized in the Yield rule in Figure 4.7.

4.5.1 Transition Rules

ApplyCont

$$\frac{\langle \beta_\kappa,\ [\![(\texttt{cont}_l\ x\ .\ call)]\!] \rangle = \varsigma_\kappa \qquad \boxed{v}' = \texttt{retFrameTo}\ \Sigma_\kappa\ \boxed{v} \\ cn = \langle cid,\ n \rangle \qquad cn \notin \texttt{cns}\ M \qquad \beta'_\kappa = \beta_\kappa[l \mapsto cn] \qquad ve_\text{ext} = [(cn, x) \mapsto \boxed{v}']}{M \vdash \langle \text{CApply}(\langle \varsigma_\kappa,\ \Sigma_\kappa,\ D_\kappa \rangle,\ \boxed{v}),\ \Sigma,\ D \rangle^{cid} \overset{\text{Core}}{\Longrightarrow} \langle \{ \langle \text{Eval}(\beta'_\kappa,\ call),\ \Sigma_\kappa,\ D_\kappa \rangle^{cid} \},\ ve_\text{ext} \rangle}$$

FIGURE 4.6: Continuation closure application.

The ApplyCont rule applies when ρ_κ is a continuation closure $\langle \beta_\kappa,\ [\![(\texttt{cont}_l\ x\ .\ call)]\!] \rangle$. First, the frame of $\boxed{v}$ is reverted back to the rank of Σ_κ with `retFrameTo` (Section 4.5.2), giving $\boxed{v}'$. That adjusted result is then bound to x under a fresh contour cn, chosen exactly as in ApplyClo.

This is similar to the ApplyClo rule (Section 4.4.1), except that there is no lifting. This is because lifting only takes place on application and there is not even a way to express expected rank for a return site. For generality, however, one can think of parameter x as if it expects rank `all`. Then the principal frame of the application is trivially empty.

The core steps to an *Eval* state where the continuation's body *call* will be evaluated under extended binding environment β'_κ. Crucially, the core does not keep its own Σ and D after the transition. Instead, it resumes execution under Σ_κ and D_κ, the parallel state captured back when this continuation value was constructed. In this sense, invoking a continuation really is a call return.

YIELD

$$\boxed{v}' = \texttt{retFrameTo}\ \Sigma_\kappa\ \boxed{v} \qquad \boxed{v}'' = \boxed{v}' \downarrow D_\kappa$$

$$M \vdash \langle \text{CApply}(\langle \texttt{yield}(cid_\text{P}),\ \Sigma_\kappa,\ D_\kappa \rangle,\ \boxed{v}),\ \Sigma,\ D \rangle^{cid} \xrightarrow{\text{Core}} \langle \text{Done}(\boxed{v}'',\ cid_\text{P}),\ \Sigma_\kappa,\ D_\kappa \rangle^{cid}$$

FIGURE 4.7: Yielding the result to parent core.

YIELD applies when ρ_κ is $\texttt{yield}(cid_\text{P})$. This primitive indicates that the child core has completed the assigned SIMD subcomputation and it now needs to report the result back to the parent core with id cid_P.

Just as before, the frame of the incoming result $\boxed{v}$ is reverted to the rank of Σ_κ with `retFrameTo`. This time though, Σ_κ is coming from the parent core and describes the parallel context within which this child core (and its siblings) were spawned. Each child, then, must produce a result that conforms to this frame shape.

D_κ indicates the original domain that this child core was spawned with, which is also the subdomain of the parent core that this child is responsible for. To prepare the result collection to be communicated to the parent, the core restricts the result to D_κ (Section 3.2.5). The produced $\boxed{v}''$ is a fully reified collection fragment that is ready to be assembled into a complete MIMD result by the parent core.

The core then transitions to a *Done* state. It will hold onto the processed collection $\boxed{v}''$ for as long as the parent core with id cid_P is in a *Wait* state. Once every sibling is done, the parent resumes and collects $\boxed{v}''$ from the children. After this, the child core will die.

Note also that the top-level $\kappa_{\texttt{HALT}}$ continuation is just a continuation value defined to be $\langle \texttt{yield}(\bullet),\ \bullet,\ \texttt{dom}\ \bullet \rangle$ (Figure 3.22). Then, invoking $\kappa_{\texttt{HALT}}$ will reset the frame of the result to empty, making the entire collection shape its cell shape. Restricting to the scalar domain $\texttt{dom}\ \bullet$ indexes the result at $[\bullet]$, which reifies the entire collection, so the machine will never have implicit replication in the final result.

4.5.2 Adjusting Frame Parallelism

`retFrameTo` specifies how to adjust the view on parallelism represented within a collection such that it conforms to some provided parallel context. Recall that a collection shape $(\psi \mid \omega)$ consists of frame dimensions ψ and cell dimensions ω. The frame describes the shape of parallelism (i.e. how many CEs can access data from this collection at the same time), while the cell tells us what each compute element sees.

The parallel context of a core must always be in sync with the frame dimensions of collections on which it operates. When a continuation gets invoked, the core's parallel context gets rolled back to whatever the continuation value captured, and the frame of the result collection must also be adjusted accordingly. Operationally, one can think of adjusting the parallelism within a collection as moving the vertical bar inside of $(\psi \mid \omega)$.

$$\texttt{retFrameTo} : \mathit{ParCtx} \to \mathit{Val} \to \mathit{Val}$$
$$\texttt{retFrameTo}\ \Sigma\ \ulcorner\!\llcorner (\psi \mid \omega) \parallel m \lrcorner\!\urcorner = \ulcorner\!\llcorner (\psi' \mid \omega') \parallel m \lrcorner\!\urcorner$$

where

$$\psi' = \texttt{take}\ (\texttt{rank}\ \Sigma)\ (\psi +\!\!+ \omega) \qquad \omega' = \texttt{drop}\ (\texttt{rank}\ \Sigma)\ (\psi +\!\!+ \omega)$$

4.6 Wait: Monitoring Child Cores

Recall that a core in the *Wait* state has the following shape:

$$\langle \text{Wait}(cid^{\#},\ \kappa),\ \Sigma,\ D \rangle^{cid}$$

This state's purpose is MIMD synchronization. The set $cid^{\#}$ contains the ids of all the child cores that *Split*? state spawned during a MIMD split, and κ is the continuation of the original call, carried over unchanged from before the split. A core sits in this state until every one of its children is done, at which point their results get assembled back into one collection and handed off to κ. In this way, *Wait* state serves as a synchronization barrier after a MIMD split. Figures 4.8 and 4.9 formalize this.

4.6.1 Transition Rules

WAIT

$$\frac{cid^{\#} \not\subseteq \texttt{doneCores}\ M}{M \vdash \langle \text{Wait}(cid^{\#},\ \kappa),\ \Sigma,\ D \rangle^{cid} \xrightarrow{\text{Core}} \langle \text{Wait}(cid^{\#},\ \kappa),\ \Sigma,\ D \rangle^{cid}}$$

FIGURE 4.8: Waiting for children after a MIMD split.

The WAIT rule applies for as long as some child in $cid^{\#}$ has not yet reached a *Done* state, that is, while $cid^{\#} \not\subseteq \texttt{doneCores}\ M$ (Section 4.6.2). This transition does nothing else and the core simply steps to itself. It will keep waiting until all of its children have computed their assigned SIMD subcomputations.

JOIN

$$\frac{cid^{\#} \subseteq \texttt{doneCores}\ M \qquad \boxed{res_C}^{\#} = \texttt{collectResults}\ M\ cid^{\#} \qquad \boxed{res} = \texttt{assemble}\ \Sigma \left\{ \langle idx,\ \boxed{\boxed{res_C}} \rangle \mid \forall idx \in D,\ \exists \boxed{res_C} \in \boxed{res_C}^{\#},\ \boxed{\boxed{res_C}} = \boxed{res_C}\ @\ idx \right\}}{M \vdash \langle \text{Wait}(cid^{\#},\ \kappa),\ \Sigma,\ D \rangle^{cid} \xrightarrow{\text{Core}} \langle \text{CApply}(\kappa,\ \boxed{res}),\ \Sigma,\ D \rangle^{cid}}$$

FIGURE 4.9: Join after a MIMD split.

JOIN applies once every child in $cid^{\#}$ is done, so the parent can join the results after a MIMD split and resume execution in SIMD mode. `collectResults` (Section 4.6.2) collects each child's result into a set $\boxed{res_C}^{\#}$, one collection per child. To put these back together, the core calls `assemble` with its own parallel context Σ, exactly as APPLYPRIM does after running a primitive (Figure 4.5). For every index idx of the core's domain D, we know that there exists a collection result in $\boxed{res_C}^{\#}$ that has computed that cell. This is because when child cores were originally spawned, D was partitioned among them and every index must have been assigned to exactly one child.

`assemble` takes the set of index-cell pairs and puts it all together in a result collection $\boxed{res}$, exactly as it would for a primitive result. Finally, the core transitions to

CApply(κ, $\boxed{res}$), invoking the original continuation with the joined value. At this point, the core has moved past its synchronization barrier.

4.6.2 Talking to The Machine

The WAIT and JOIN rules require the parent core to examine the state of other cores in the machine. These two helpers make it possible.

`doneCores` consults the current machine state and returns a set of core ids that are presently in a *Done* state. This is how a parent core determines whether all of its SIMD subcomputations have been completed.

$$\texttt{doneCores} : \mathit{Machine} \to \mathit{Set}(\mathit{CoreId})$$

`collectResults` goes over the set of cores in the *Done* state and pulls out the results those finished cores are holding onto. They will then be assembled into a MIMD result collection by the parent.

$$\texttt{collectResults} : \mathit{Machine} \to \mathit{Set}(\mathit{CoreId}) \to \mathit{Set}(\mathit{Val})$$

4.6.3 Assembling MIMD Results

Assembling the joined MIMD result uses the exact same `assemble` function defined for primitive operations (Section 4.4.3). This is because, from the perspective of a core, a MIMD split and a primitive application are processed in a very similar way. In both cases, the core has to wait for every one of its subordinate evaluators, be they CEs or child cores, to produce a fraction of the overall result. In fact, a child core is simply an abstraction over a group of CEs executing in lockstep, so a MIMD split can also be seen as a primop-like procedure executed by the parent core. Alternatively, *Apply* state over a primop can be seen as a trivial, first-order synchronization barrier, similar to how *Wait* state is a proper synchronization point for MIMD computation.

The similarity goes further than just reusing `assemble`. Every result cell that reaches `assemble` during a primitive application is guaranteed to be reified, since every CE in D executed and produced a result. The same is true here: every child core that is ready to yield its result fragment back to the parent will first restrict and reify the data within its assigned domain (Section 4.5). By the time `collectResults` collects $\boxed{res_C}^{\#}$, every collection in it will be fully realized.

4.7 Done: Delivering Results

Recall that a core in the *Done* state has the following shape:

$$\langle \text{Done}(\boxed{v},\ cid_{\text{P}}),\ \Sigma,\ D\rangle^{cid}$$

This is the final state within a core's lifetime. At this point, the core has executed all the assigned computation and prepared the result collection $\boxed{v}$ to be handed off to its parent with id cid_{P}. A core in this state will simply hold onto the result until the parent is ready to collect it, and then it will die. Figures 4.10 and 4.11 formalize this.

4.7.1 Transition Rules

DONEWAIT

$$\frac{\texttt{parentWaiting}\ M\ cid_{\text{P}}}{M \vdash \langle \text{Done}(\boxed{v},\ cid_{\text{P}}),\ \Sigma,\ D\rangle^{cid} \xrightarrow{\text{Core}} \langle \text{Done}(\boxed{v},\ cid_{\text{P}}),\ \Sigma,\ D\rangle^{cid}}$$

FIGURE 4.10: Holding onto a result.

The DONEWAIT rule applies while the parent core cid_{P} is still waiting on a MIMD split. `parentWaiting` is a predicate that checks exactly that. Recall that the parent core will be in the *Wait* state up until all of its children have completed their assigned computations (Section 4.6). Only after they are all in a *Done* state will the parent collect their result fragments and transition to *CApply* to resume execution in SIMD mode. As a result, child cores will wait in *Done* state at least until all of their siblings are done too.

DIE

$$\frac{\neg\ \texttt{parentWaiting}\ M\ cid_{\text{P}}}{M \vdash \langle \text{Done}(\boxed{v},\ cid_{\text{P}}),\ \Sigma,\ D\rangle^{cid} \xrightarrow{\text{Core}} \langle \varnothing,\ \varnothing\rangle}$$

FIGURE 4.11: Core self clean-out.

DIE applies once parent core cid_{P} is no longer waiting on the current core. Past this point, the current core has no reason to keep occupying space in the machine, so it requests to be replaced by the empty set of cores and simply disappears after the machine step completes.

Chapter 5

Implementation Overview

The SIMD/MIMD Machine formalized in the previous two chapters is implemented in Haskell, in roughly 3800 lines. The implementation was written alongside the semantics, which facilitated prototyping and helped identify minor inconsistencies in transition rules. A tested implementation also lends confidence in the model, even in the absence of formal soundness proofs.

The module structure follows the thesis presentation. The semantic domains of the SIMD/MIMD Machine each get their own module. These include shapes, domains, index maps, collections, contours and values. The core state and stepping relation are grouped in one module, where each of the core transition rules is implemented with a separate function. The machine stepping relation is defined in terms of `pickCores`, as described in Figure 3.24. It is currently instantiated with a basic scheduler that selects every available core.

Since the code is in Haskell, the implementation is pure: running a program is a function from CPS REMORA syntax to a result collection, with state threaded monadically. The machine runs in a state monad holding the cores, the global value environment and the contour supply. A core step runs in a monad transformer, carrying the core's id, parallel context and domain. The contour supply is how the implementation satisfies the freshness condition. At each machine step, the picked cores are traversed sequentially and the contour supply is threaded through. A plain numeric counter suffices to keep every allocation distinct.

The front end parses CPS REMORA directly, including the syntactic sugar of Section 2.2.3, and then alphatizes the program and assigns syntactic labels. It supports several modes. One parses a program and prints it back, which is the quickest way to see how a piece of sugar expands. Another constructs an initial core with the given top-level thunk and runs the machine until the final core reports the resulting collection. A third mode is for abstract interpretation, which is where the abstraction of Section 6.3 was prototyped. Input can be a file, a single top-level call passed as an argument, or standard input.

The implementation was tested against a suite of representative CPS REMORA programs, covering lifting, closures, conditionals and direct MIMD splits. Each test file specifies its expected result in a header comment, and the testing harness checks the machine against it.

5.1 Execution Traces

A notable feature of the implementation is the accumulation of the execution trace, which can be printed out at the end of a run. The machine carries a full history as a part of its state, appending to the log after every machine step. If a failure is encountered, we can examine the trace that led to it; this is what made the implementation useful when debugging the SIMD/MIMD Machine semantics.

To see an example of an execution trace, consider the following program: an array of two distinct primitives applied to a scalar and a vector.

$$(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ .\\ \qquad ([+\ -]\ \ 4\ \ [1\ 2]\ \ k_{\mathtt{TOP}}))$$

The application lifts to a principal frame of (2), computes $(+\,4\,1)$ and $(-\,4\,2)$ on separate cores, and returns $[5\ \ 2]$. When run on the SIMD/MIMD Machine with the default `pickCores` scheduler, it will take nine machine steps. Figure 5.1 shows the three steps in the middle, where the MIMD split happens.

step 3: core $[\,]$: <split?> $\ulcorner\!\llcorner (2 \mid \bullet) \parallel [0 \mapsto +,\ 1 \mapsto -] \lrcorner\!\urcorner$ $[\,a_1\ ,\ a_2\,]$ $\kappa_{\mathtt{TOP}}$

step 4: core $[\,]$: <wait> $\{[+], [-]\}$ $\kappa_{\mathtt{TOP}}$
core $[+]$: <apply> $+$ $[\,a_1\ ,\ a_2\,]$ $\langle \mathtt{yield}([\,]),\ (2),\ \{[0]\}\rangle$
core $[-]$: <apply> $-$ $[\,a_1\ ,\ a_2\,]$ $\langle \mathtt{yield}([\,]),\ (2),\ \{[1]\}\rangle$

step 5: core $[\,]$: <wait> $\{[+], [-]\}$ $\kappa_{\mathtt{TOP}}$
core $[+]$: <capply> $\langle \mathtt{yield}([\,]),\ (2),\ \{[0]\}\rangle$ $\ulcorner\!\llcorner (2 \mid \bullet) \parallel [0 \mapsto 5] \lrcorner\!\urcorner$
core $[-]$: <capply> $\langle \mathtt{yield}([\,]),\ (2),\ \{[1]\}\rangle$ $\ulcorner\!\llcorner (2 \mid \bullet) \parallel [1 \mapsto 2] \lrcorner\!\urcorner$

FIGURE 5.1: Steps 3 to 5 of a printed trace. The lifted arguments are abbreviated as $a_1 = \ulcorner\!\llcorner (2 \mid \bullet) \parallel 4 \lrcorner\!\urcorner$ and $a_2 = \ulcorner\!\llcorner (2 \mid \bullet) \parallel [0 \mapsto 1,\ 1 \mapsto 2] \lrcorner\!\urcorner$, and $\kappa_{\mathtt{TOP}}$ is the top-level $\kappa_{\mathtt{HALT}}$ continuation.

Each core is printed as its id followed by its execution state. The pretty printer uses the notation of Chapter 3, so a trace can be read against the transition rules presented in this thesis directly.

At step 3 the core finds two distinct procedures, so the SPLITMIMD rule applies. Step 4 shows the machine state after the split: the parent core is in *Wait* state, and the children are in *Apply* state, each applying a different procedure to the same arguments over disjoint domains. At step 5, both child cores are invoking the `yield` continuations they were spawned with.

Chapter 6

Further Work

The SIMD/MIMD Machine formalizes a concrete semantics that makes MIMD splits explicit, but that is not the full story. The eventual goal is a *static* SIMD/MIMD analysis: a compile-time procedure that identifies stretches of SIMD code in a CPS REMORA program so that they can be extracted into kernels. This chapter sketches what work remains to be done.

This chapter is organized as follows:

- Section 6.1 shows how the machine can be extended to support higher-order primitives such as `reduce`, which the current model deliberately leaves out;
- Section 6.2 discusses what a nonstandard semantics derived from the SIMD/MIMD Machine should accumulate, and what shape that output has to take for kernel extraction to be possible;
- Section 6.3 discusses how to abstract that nonstandard semantics into a computable approximation, and why a very coarse abstraction is likely to be good enough;
- Section 6.4 lays out the soundness obligations that the concrete, nonstandard and abstract semantics still owe.

6.1 Higher-Order Primitives

Higher-order primitives like `reduce` and `scan` are what allow REMORA programs to combine data along an axis without writing out accumulating loops. The SIMD/MIMD Machine implements the semantics of rank polymorphism, and, as a result, every function application comes with an implicit map, but the current model is not equipped for combining data *across* an axis.

Supporting such primitives is definitely possible but not trivial. At a basic level, the problem stems from how the machine views primop applications. The primop evaluator `execPrim` is understood to model actual computation that compute elements within a SIMD evaluator can perform. Mathematically, this means that `execPrim` consumes and produces cell values with passive data (Section 4.4.2).

A higher-order primitive like `reduce` has to bring together data sitting in different cells and combine them with an arbitrary operator. While it is possible to hardcode certain procedure combinations, like `reduce/plus` for a reduce over $+$, that is far from general. Besides, it would be a waste not to reuse the established SIMD/MIMD machinery when handling the arbitrary computation that sits inside a higher-order primitive.

Luckily, the modifications necessary to support this are not too bad; it is mostly a matter of letting these special primops transfer control to the operator argument. Figure 6.1 indicates the states and transitions such an extension would need to affect.

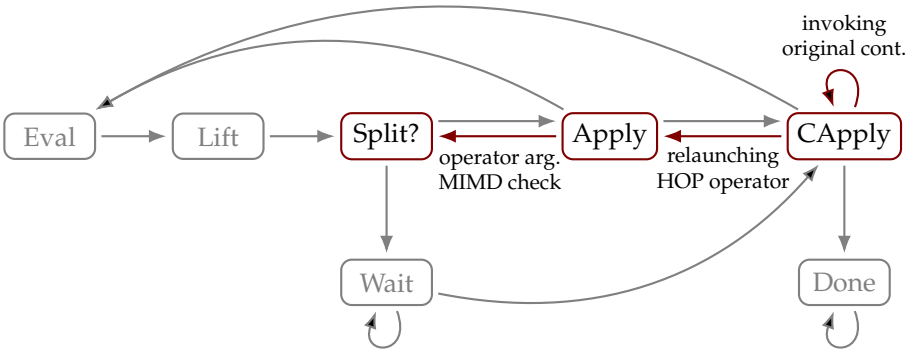

FIGURE 6.1: Core states and transitions that would need to be modified to support higher-order primitives are highlighted in red (*Split*?, *Apply*, *CApply*).

The rest of this section walks through one possible way to achieve this, using the `reduce` primitive as an example. In REMORA, its expected argument ranks are [0 `all`].

6.1.1 Delayed MIMD Splits

We start with the most glaring issue: a new source of MIMD splits.

Recall that a core decides whether an application can proceed in SIMD mode by examining the collection in *function* position. For a higher-order primitive, that collection holds a single primitive operator, so the check succeeds trivially. But the procedure that the primitive will actually run arrives as an *argument*, and arguments are lifted collections, *not* individual procedures. Nothing stops such a collection from holding more than one distinct procedure.

Consider, for example,

$$
\begin{array}{l}
(\texttt{fn}_{\texttt{TOP}}\ []\ k_{\texttt{TOP}}\ . \\
\quad ((\texttt{fn}_{\texttt{L1}}\ [(f\ 0)]\ k_1\ .\ (\texttt{reduce}\ f\ [4\ 1\ 3]\ k_1)) \\
\quad\quad [+\ -] \\
\quad\quad k_{\texttt{TOP}}))
\end{array}
$$

The application of $\texttt{fn}_{\texttt{L1}}$ lifts to a principal frame of (2) and f gets bound to a collection

$$
\ulcorner (2 \mid \bullet) \urcorner \ \Big\| \ \begin{pmatrix} 0 \mapsto + & \\ 1 \mapsto - & \end{pmatrix}
$$

If `reduce` were a first-order primop, applying it under a parallel context of (2) would be a SIMD computation, but not here. Once the two CEs begin evaluating the primop, their respective operators diverge. What we are looking at is a *delayed* MIMD split.

It might be possible to address this with a more complicated `extractSIMD` that looks past the function position and partitions the domain by the operator argument as well. There is, however, a more elegant solution if we handle it as a part of the *Apply* state. Then, with a special transition for higher-order primitives, the core can get kicked back into *Split*? to examine the argument that might result in a MIMD split. Here, that would be the collection bound to f.

The rest of the machinery follows naturally: `extractSIMD` partitions the domain, child cores get spawned, and the parent drops into *Wait* exactly as it does for regular MIMD splits. A nice consequence is that if a split is triggered, the parent waits specifically on the `reduce`'s MIMD computation and not on any other procedure.

6.1.2 Reducing Along the Main Axis

The next big issue is accumulating data, which is always at the core of a higher-order primitive. In the `reduce` case, this means combining along the outer axis of available cell data, until just one reduced cell remains. A reduce in CPS REMORA can of course run over an arbitrarily large array, but it will always operate along that one main axis, and its result will have collapsed that axis down to a scalar.

To get started, we need the sequence of cells that we will be reducing over, which a new helper `mainAxis` could produce:

$$\texttt{mainAxis} : \mathit{Val} \to \mathit{Seq}(\mathit{Val})$$

It takes a value and returns a sequence of values, sliced along the outermost cell dimension.

That sequence then needs to get launched, preferably while maximizing SIMD. To do so, we split it in two (with a possible odd element left over), combine each of the two subsequences into a new argument collection, and then the *Apply* state can run the reduce's operator over those collections. Every such pass halves the length of the sequence, maximizing the amount of work being done at each iteration.

6.1.3 Accumulating Data in Continuation Values

Because of the data dependencies inherent to reductions, it is of course impossible to compute the entire `reduce` in one go. After a pass is completed, there needs to be a mechanism to recombine the intermediate results (along with the possible odd main-axis element) and launch a new pass. This describes a special control transfer that relies on the dynamic state of the execution, which suggests that we need a new continuation primitive (Section 3.2.8).

A continuation primitive for `reduce` can look something like

$$\texttt{reduceCont}(\rho, \boxed{v}^\star, \kappa)$$

where ρ is the procedure this reduce is operating over, $\boxed{v}^\star$ is a possibly empty sequence of intermediate results not being processed, and κ is the continuation value that must be invoked once the `reduce` computation has been completed.

When this continuation primitive gets invoked with the results of a pass, `mainAxis` can extract the current accumulated sequence of intermediate results, combine it with $\boxed{v}^\star$, and then repeat the process of constructing two new argument collections for the next run of ρ. This is the *CApply* $\rightarrow$ *Apply* transition in Figure 6.1. We know that there is no more work to be done when the intermediate results contain just one value. That value is the answer, and κ gets invoked with it, which keeps the core in *CApply* for one more step.

6.2 SIMD/MIMD Analysis Output IR

Everything in this thesis builds toward the goal stated in Section 1.2: the SIMD/MIMD analysis. What we ultimately want is a procedure that can analyze an arbitrary REMORA program at compile time and identify where MIMD splits occur; everything in between two splits is then a SIMD fragment. Kernels intended for parallel hardware can then be built out of SIMD fragments.

This section and the next follow the stages of the NSAS framework (Section 1.2.3). The SIMD/MIMD Machine of Section 3.3 is the standard semantics, and it has been designed to make MIMD splits explicit. This section considers the second stage, namely how we would go about designing *nonstandard semantics* with an analysis output that is actually useful.

Given the design of the SIMD/MIMD Machine, coming up with some notion of nonstandard semantics is not difficult. One can imagine a nonstandard machine that carries a call graph payload alongside its cores and extends that graph every time a procedure is applied by one of its cores. A MIMD split happens precisely when a *Split*? state takes the SPLITMIMD transition, so a nonstandard machine can easily record that split in the graph.

A much harder question is deciding what that call graph should look like exactly. "The set of call sites that split" is a perfect answer to the question as posed, but it is not very usable when thinking about kernel extraction. What is a useful result that could allow us to extract maximal kernels out of it?

One possible strategy is to organize the output as an extended call graph whose nodes are *call sites* and annotated edges record what procedure was called from this call site, and where control went as a result.

Nodes are deliberately not procedures: this is a higher-order language, so a single call site can call many different procedures over the course of execution.

6.2.1 SIMD Call Graph Example

The program below demonstrates some of the design challenges.

$$
\begin{array}{l}
(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ . \\
\quad ((\mathtt{fn}_{\mathtt{L1}}\ \ [(f\ 0)]\ \ k_1\ . \\
\qquad\quad (f\ \ \times\ \ (\mathtt{cont}_{\mathtt{L3}}\ x\ .\ (f\ \ [+\ -]\ \ k_1)))) \\
\quad\ (\mathtt{fn}_{\mathtt{L2}}\ \ [(g\ 0)]\ \ k_2\ .\ (g\ \ 3\ \ 4\ \ k_2)) \\
\quad\ k_{\mathtt{TOP}}))
\end{array}
$$

Here $\mathtt{fn}_{\mathtt{L1}}$ is applied to $\mathtt{fn}_{\mathtt{L2}}$, so f is bound to a procedure. $\mathtt{fn}_{\mathtt{L1}}$'s body then calls f twice, first with $\times$ and then, from inside $\mathtt{cont}_{\mathtt{L3}}$, with $[+\ -]$. $\mathtt{fn}_{\mathtt{L2}}$ applies whatever it is handed to 3 and 4.

It will be useful to have names for the call sites. Let $c_{\mathtt{TOP}}$ be the call in $\mathtt{fn}_{\mathtt{TOP}}$'s body, c_1 the call in $\mathtt{fn}_{\mathtt{L1}}$'s body, c_2 the call in $\mathtt{fn}_{\mathtt{L2}}$'s body, c_2^{κ} the continuation position of c_2, and c_3 the call in $\mathtt{cont}_{\mathtt{L3}}$'s body. Figure 6.2 below proposes a possible SIMD call graph for this program.

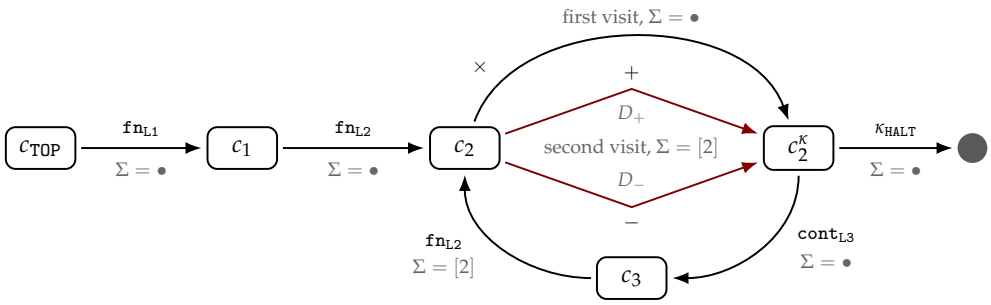

FIGURE 6.2: A sketch of the SIMD call graph for the example program above. The MIMD split is highlighted in red.

In this graph, each of the call sites is a node, and the edges represent transfers of control. Edges are also annotated with the procedure identifiers and the parallelism state of the executing core. A join point is just a call site that invokes a continuation after a MIMD split. The join point is a synchronization barrier, and in the graph it is the vertex that closes the diamond that a MIMD split opens.

Consider c_2, which corresponds to $(g\ 3\ 4\ k_2)$. It is just one call site, but it is reached twice. On the first visit g is bound to $\times$, one distinct procedure, and the application is plain SIMD. On the second visit g is bound to $[+\ -]$, and the very same call site is a MIMD split.

It follows that being SIMD or MIMD is not just a property a call site has; it is a dynamic property of a call site *reached in a particular way*. Perhaps it is worth exploring redefining graph nodes with more complex structure, but we can also utilize edge annotations. Then, the $\times$ edge and the $\{+, -\}$ edges fall into different call contexts and only the latter pair denotes a split. If there were more SIMD calls from the same call site, they would be grouped with the $\times$ edge.

6.2.2 SIMD Fragments

A *SIMD fragment* is a chain of SIMD edges. The graph in Figure 6.2 has four:

$$F_1 \ : \ c_{\text{TOP}} \xrightarrow{\text{fn}_{\text{L1}}} c_1 \xrightarrow{\text{fn}_{\text{L2}}} c_2 \xrightarrow{\times} c_2^\kappa \xrightarrow{\text{cont}_{\text{L3}}} c_3 \xrightarrow{\text{fn}_{\text{L2}}} c_2$$

$$F_2 \ : \ c_2 \xrightarrow{+} c_2^\kappa$$

$$F_3 \ : \ c_2 \xrightarrow{-} c_2^\kappa$$

$$F_4 \ : \ c_2^\kappa \xrightarrow{\kappa_{\text{HALT}}} \mathit{Done}$$

F_1 is the interesting one. It passes through c_2 twice, so a kernel extracted from it jumps to fn_{L2}'s body twice: once in full at parallel context $\bullet$, and once again up to the point where it splits at (2). A call site that eventually splits therefore does not have to end a fragment; only the visit that splits does. This is what makes paths a better output than a set of call sites labeled SIMD or MIMD.

In this framing, kernel extraction is the problem of identifying maximal SIMD chains, which is to say a graph optimization problem. It is also where fragments become kernels: the analysis must report every split, but a compiler need not make each one a kernel boundary. A split this small is a candidate for a target-dependent rewrite merging F_2 and F_3, since SIMT hardware absorbs minor branching on its own [14, 16].

6.3 Abstracting the SIMD/MIMD Machine

The nonstandard semantics discussed in the previous section computes a perfect answer to the SIMD/MIMD analysis, but it gets there by fully running the program. This is not feasible at compile time and therefore it cannot be utilized as a static analysis, so the third NSAS stage (Section 1.2.3) is to abstract the semantics.

Strictly speaking, the subset of REMORA presented in this thesis is simple enough that it does not run into some of the fundamental issues demanding abstraction. That said, REMORA is a general-purpose language, and for this analysis to be useful downstream, it needs to stay computable. Three issues are worth highlighting.

The first is recursion. The current instantiation does not support recursive scope like `letrec`, but it is a planned part of REMORA. Once non-termination is a possibility, we run into programs with a potentially infinite number of bindings. Recall that every application picks a fresh contour from the contour set (Section 3.2.6), and with the set of contours being natural numbers, there will be infinitely many distinct contours to pick from. This makes the set of environments infinite as well.

Another issue is user input. The data that will get loaded into the program will be finite, but the analysis does not have access to this information at compile time. For the analysis to hold over all possible programs it also needs to hold over all possible inputs.

The last issue is already present. Arrays in REMORA treat procedures as simple atoms up until a procedure collection flows to a function position. Then, the previously passive data suddenly becomes responsible for dynamic control flow—which could result in a MIMD split. If the static analysis tried to precisely identify the control flow up to domain splits, it would have to execute primitives over concrete data just to find out the frame indices of certain procedures. This is, of course, not feasible at compile time.

6.3.1 Sound Approximations in SIMD/MIMD Analysis

An abstract semantics achieves computability by merging states. One abstract state stands for many concrete ones, and a single abstract trace covers multiple concrete executions. A finite state space is precisely what allows for a computable analysis, but we lose precision in the process. After all, two concrete states represented by the same abstract state are indistinguishable to the analysis.

For the SIMD/MIMD analysis, the primary question is classifying call sites as SIMD or MIMD. Recall that a SIMD computation is just a trivial MIMD computation with exactly one subcomputation. Marking a call site as MIMD is therefore a safe overapproximation the analysis can always fall back to.

6.3.2 Simple Abstraction Is Good!

Normally, coming up with a good abstraction is a lot of delicate work, but in our case a simple abstraction is already bound to be very effective. This holds for higher-order control flow in general. Following a folk theorem, most higher-order code is really just first-order, so even an abstraction that collapses states significantly still yields precise results [19].

REMORA makes the deal even better, because Iverson's rank-polymorphic computational model exerts a lot of linguistic pressure on programmers to write SIMD code.

Lifting is implicit and comes for free, so the terse way of expressing something is usually also the data-parallel way of expressing it. An analogous folk theorem could be that core computational functions in general-purpose rank-polymorphic code are really just SIMD. The analysis does not need to worry about an abundance of MIMD splits.

6.3.3 Abstraction Targets

Two components drive the abstraction.

The first is binding contours. The size of *Contour* determines how many binding instances the analysis keeps distinct, and with $\mathit{Contour} := \mathit{CoreId} \times \mathbb{N}$ it is countably infinite. Following 0CFA [22], we can pick a singleton instead,

$$\mathit{Contour} := \mathit{CoreId} \times \mathbb{N} \qquad \xrightarrow{\alpha} \qquad \widehat{\mathit{Contour}} := \{\diamond\}$$

With only one contour to pick, the abstract rules drop the freshness side condition and bind under $\diamond$ every time. Every binding of a variable maps to a set that contains all abstract collections ever bound to it:

$$\langle cn_0,\ x\rangle \mapsto \boxed{v_0},\ \langle cn_1,\ x\rangle \mapsto \boxed{v_1},\ \ldots \qquad \xrightarrow{\alpha} \qquad \langle \diamond,\ x\rangle \mapsto \{\widehat{\boxed{v_0}},\ \widehat{\boxed{v_1}},\ \cdots\}$$

The second is collections. A collection holding more than one distinct procedure might lead to a MIMD split, so an abstract collection must preserve the set of procedures. The shape is inconsequential to the analysis, so it can be dropped.

$$\lceil\!\lfloor (2 \mid \bullet) \parallel [0 \mapsto +,\ 1 \mapsto -] \rfloor\!\rceil \qquad \xrightarrow{\alpha} \qquad \{+,\ -\}$$

Collections with basic atoms collapse entirely, since no procedure can flow out of them:

$$\lceil\!\lfloor (\bullet \mid 3) \parallel [0 \mapsto 4,\ 1 \mapsto 1,\ 2 \mapsto 3] \rfloor\!\rceil \qquad \xrightarrow{\alpha} \qquad \varnothing$$

Discarding passive data allows `execPrim` to approximate a computation instead of running it. For basic primitives this is trivial, as they take empty sets and return an empty set. For `id`, the function stays an identity, simply passing through the possibly empty sets. `select` is the interesting one, as it might affect the control flow. Since the abstracted guard argument is empty, `select` cannot tell which of the two branch thunks to return, so a safe overapproximation is to return both. The call site that applies the result then sees two distinct procedures, and the analysis reports a MIMD split. Any additional primitives should be safely abstracted in a similar fashion.

It is worth noting why collection shapes do not matter to the analysis. Lifting replicates data and grows frames, but it never introduces atoms to a collection that were not already there. Whether a given call site splits depends only on which procedures reach it, so the semantics of rank polymorphism has no bearing on the presence of MIMD splits. What is left after the abstraction is an ordinary control flow problem, except it deals with sets of sets of procedures rather than a flat set.

Abstract cores collapse significantly. With no index maps there is no need for a domain, and with no collection shapes there is no lifting or parallel context. The core id is worth keeping, since it still records the MIMD ancestry that led to a given abstract state.

$$\langle st,\ \Sigma,\ D\rangle^{cid} \qquad \xrightarrow{\alpha} \qquad \langle \widehat{st}\rangle^{cid}$$

6.4 Soundness Proofs

Every claim this thesis has made about the SIMD/MIMD Machine has been informal. Making them precise would take a sequence of proofs, roughly one per NSAS stage. Each leans on the one before it.

The reference point exists already: direct-style REMORA has a semantics whose type system is proven sound [23]. To go from this reference semantics to the SIMD/MIMD Machine—which is our standard semantics on the way to SIMD/MIMD analysis—we need to define a CPS converter. Formalizing it should not take much effort. Structurally, CPS REMORA is an ordinary factored CPS, and linear-time conversions for it are well understood [4].

With a converter in hand, the SIMD/MIMD Machine can be shown sound with respect to the converted program by simulation. The correspondence will not follow in lockstep, however. A single reduction in the direct-style program becomes a whole run of core transitions, and an entire subtree of child cores whenever the application splits.

Two desirable properties of the machine will require additional work to prove. The first is that the answer does not depend on the schedule. `pickCores` is nondeterministic by design (Section 3.3.6), which is what lets the SIMD/MIMD Machine model execution under limited resources. It must be shown that this choice never affects the computed value.

The second is progress. The WAIT and DONEWAIT rules both step a core to itself, so a machine that never gets stuck is not automatically one that makes progress. A parent could sit in *Wait* forever if `pickCores` never scheduled one of its children. Ruling that out needs a fairness condition on `pickCores`, together with the observation that the core tree is finite and its leaves are always runnable.

Given these two properties, the claim that the machine gets stuck only on a runtime error (Section 3.3.6) largely falls out of the simulation. For the primitives and basic values of Figure 2.1 the statement is even stronger, since runtime errors cannot occur at all.

Finally, the abstract semantics must be shown to be a safe abstraction in the usual sense [3]. Following NSAS, the abstraction must be safe with respect to the nonstandard semantics that computes a perfect analysis answer. The direction of safety has already been argued for in the previous section. What a proof guarantees is that the analysis never underapproximates: every MIMD split encountered during a concrete run must be reflected in the SIMD/MIMD analysis output.

Chapter 7

Conclusion

The overarching goal that this work serves is kernel extraction from REMORA programs. For a compiler to perform this operation, it needs to statically identify the points at which otherwise lockstep computation diverges, known as MIMD splits. What makes this difficult is that the notion of a MIMD split is entirely semantic and determined by whatever values flow to a given call site. We frame this as a control flow analysis, called the SIMD/MIMD analysis.

To derive this analysis, we follow the Non-Standard Abstract Semantics framework. The foundational first stage is to define a standard semantics of the object language that makes the desired property—in our case MIMD splits—explicitly observable. Formalizing such a standard semantics is the central contribution of this work.

The thesis describes the SIMD/MIMD Machine, an operational semantics for CPS REMORA in which the notion of a MIMD split is defined as a machine transition. The boundary of a SIMD fragment then becomes a formally defined notion, and the choice of which fragments to launch as kernels is left to a later stage. Additionally, the machine state tracks the overall shape of parallelism and the index space of compute elements at any given point in computation. The model can also describe execution on a system with a bounded number of processing units by changing its core-picking procedure.

Writing the model down in this much detail also unveiled a few things that a more abstract semantics leaves implicit. Namely, (1) parallelism grows when a procedure collection is applied and contracts when the call returns; (2) a continuation value has to capture dynamic parallel state when it is evaluated; (3) from the perspective of a running core, a MIMD application and a primitive application share many structural similarities.

The remaining work is laid out in Sections 6.2 to 6.4. We need a nonstandard semantics that accumulates an analysis result and a sound computable abstraction of this semantics. We also need a flexible and informative representation of the SIMD/MIMD analysis output that makes extraction of maximal kernels possible. Finally, we need a chain of soundness proofs that establish that the three semantics line up. All these tasks appear reasonably achievable given the foundational standard semantics of the SIMD/MIMD Machine.

Bibliography

[1] Guy E. Blelloch. *NESL: A Nested Data-Parallel Language (Version 3.1)*. Tech. rep. CMU-CS-95-170. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 1995.

[2] Guy E. Blelloch and Gary W. Sabot. "Compiling collection-oriented languages onto massively parallel computers". In: *Journal of Parallel and Distributed Computing* 8.2 (1990), pp. 119–134. DOI: https://doi.org/10.1016/0743-7315(90)90087-6.

[3] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages (POPL)*. New York, NY, USA: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[4] Milo Davis, William Meehan, and Olin Shivers. "No-Brainer CPS Conversion (Functional Pearl)". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110267. URL: https://doi.org/10.1145/3110267.

[5] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[6] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling Machine Learning Programs via High-Level Tracing". In: *Systems and Machine Learning*. Stanford, United States, 2018. URL: https://hal.science/hal-05188750.

[7] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. "Breaking the GPU programming barrier with the auto-parallelising SAC compiler". In: *Declarative Aspects of Multicore Programming (DAMP)*. ACM, 2011, pp. 15–23. DOI: 10.1145/1926354.1926359. URL: https://doi.org/10.1145/1926354.1926359.

[8] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2011. ISBN: 9780123838735. URL: https://books.google.com/books?id=gQ-fSqbLfFoC.

[9] Troels Henriksen. "Design and implementation of the Futhark programming language". PhD thesis. University of Copenhagen, Faculty of Science [Department of Computer Science], 2017.

[10] Intel Corporation. *Intel Implicit SPMD Program Compiler User's Guide*. Sections: Selecting the Compilation Target; Gang Convergence Guarantees. 2026. URL: https://ispc.github.io/ispc.html.

[11] Kenneth E. Iverson. "A Programming Language". In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE '62 (Spring). San Francisco, California: Association for Computing Machinery, 1962, pp. 345–351. ISBN: 9781450378758. DOI: 10.1145/1460833.1460872. URL: https://doi.org/10.1145/1460833.1460872.

[12] Kenneth E. Iverson. "Notation as a tool of thought". In: *Commun. ACM* 23.8 (Aug. 1980), 444–465. ISSN: 0001-0782. DOI: 10.1145/358896.358899. URL: https://doi.org/10.1145/358896.358899.

[13] Jsoftware Inc. *Jsoftware High-performance development platform*. URL: https://www.jsoftware.com/.

[14] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31. URL: https://doi.org/10.1109/MM.2008.31.

[15] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008). Section: The CUDA Paradigm, pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: https://doi.org/10.1145/1365490.1365500.

[16] NVIDIA Corporation. *CUDA C++ Programming Guide*. Section: SIMT Architecture. Accessed August 2026. 2026. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[17] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[18] Matt Pharr and William R Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–13.

[19] Benjamin Quiring, John Reppy, and Olin Shivers. "Analyzing binding extent in 3CPS". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547645. URL: https://doi.org/10.1145/3547645.

[20] John Reppy and Joe Wingerter. "λcu—An Intermediate Representation for Compiling Nested Data Parallelism". In: *International Workshop on Compilers for Parallel Computing (CPC)*. 2016.

[21] Olin Shivers, Justin Slepak, and Panagiotis Manolios. *Introduction to Rank-polymorphic Programming in Remora (Draft)*. 2020. arXiv: 1912.13451 [cs.PL]. URL: https://arxiv.org/abs/1912.13451.

[22] Olin Grigsby Shivers. "Control-Flow Analysis of Higher-Order Languages or Taming Lambda". PhD thesis. USA: Carnegie Mellon University, 1991.

[23] Justin Slepak. "A Typed Programming Language: The Semantics of Rank Polymorphism". PhD thesis. USA: Northeastern University, 2020. ISBN: 9798664759945.

[24] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An Array-Oriented Language with Static Rank Polymorphism". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 27–46. ISBN: 978-3-642-54833-8.

[25] Justin Slepak, Olin Shivers, and Panagiotis Manolios. *The Semantics of Rank Polymorphism*. 2019. arXiv: 1907.00509 [cs.PL]. URL: https://arxiv.org/abs/1907.00509.

[26] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69. URL: https://doi.org/10.1109/MCSE.2010.69.

[27] David Van Horn and Matthew Might. "Abstracting Abstract Machines". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. New York, NY, USA: Association for Computing Machinery, 2010, 51–62. ISBN: 9781605587943. DOI: 10.1145/1863543.1863553. URL: https://doi.org/10.1145/1863543.1863553.