Cost Semantics for Heterogeneous Parallel Functional Languages

Timothy A. K. Zakian

Keble College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary Term 2019

To 佳那; my one and only. 心から愛している.

Acknowledgements

This dissertation would not have been possible without the help and encouragement of many people. First and foremost among these people, I would like to thank my supervisor Jeremy Gibbons. During my time at Oxford, he has encouraged and refined my research style (and writing!) and has slowly molded me into an independent researcher. Jeremy, without your support, encouragement, and knowledge this dissertation would not have been possible. Thank you.

To the members of the Algebra of Programming group, thank you for putting up with my talks, and always providing interesting insights and comments. My work and perspective on programming languages and computer science has benefited greatly from the insights and perspectives gleaned from our meetings. I will truly miss them.

I would like to thank the Clarendon Fund for giving me the financial ability to research and write this dissertation. It is a special privilege to be afforded the freedom of intellectual enquiry that it provides, and being able to conduct research on practically any subject without having to worry about funding is truly incredible as a DPhil student. For this privilege, I am deeply grateful.

To my family, thank you. Your love, encouragement, and support throughout my life has been central to me getting here. Mom & Dad: I cannot name the ways that you have sacrificed for Chris and me, and I can never fully repay you for all that you have done. Chris, thank you for being an amazing brother, encouraging me to take H211 and C311 my first semester, and for introducing me to Dan—I wouldn't have gotten into computer science without you.

Speaking of becoming a computer scientist, I would not have kept on the path to where I am now without the help and encouragement of the professors and graduate students during my time as an undergraduate degree at Indiana University. I would especially like to thank my undergraduate supervisor Ryan Newton who showed me how to be a researcher, and William Byrd who showed me how to code. I would also like to thank Dan Friedman and Amr Sabry for their incredible support and

mentorship throughout my undergraduate as well. Finally, thank you to the PL Wonks, and especially Cam, Spenser, Jaime, and Praveen; I owe you all a drink at Important Business.

To all of my friends in Oxford, you have made this place what it is to me, and it wouldn't have been the same without you. So to Stefan[1], Anik, Adrianne, Libby, Shea, Patrick, Valerio, Olmo, Avraham, Max, Louis, Will, and Juan, thank you for being great friends.

To my housemates: coming home to all of y'all every evening was a true pleasure and joy throughout my time in Oxford—we all kept each other sane by going crazy in slightly different ways. I want you to know that I have had some of the best moments of my life with you, and I couldn't imagine what my life in Oxford would have been like if we hadn't met in 2FK of Acland our first year. Remember, we can all be Dr. Mediocre.

[1]Sorry but I can't figure out how to write the Ș.

Abstract

In today's world parallelism is ubiquitous, and different types of parallel models and processors are becoming increasingly common. Moreover, while our programs become increasingly heterogeneous in the parallelism they employ, user-visible parallelism in our languages becomes increasingly homogeneous, with the parallelization onus put more and more on the compiler or runtime. This increasing divide between what is seen and what happens in our languages makes being able to determine the costs of running the same program on different types of hardware evermore important. And, in such a world, having the means to profile programs by their underlying semantic properties is crucial.

However, while there are means of semantically profiling parallel programs written for the CPU, there are still no means to profile and reason about parallel programs that use multiple types of parallel processor based upon their underlying semantics. In this dissertation, we set out to solve this problem by presenting a cost semantic theory that allows profiling for both time and space in heterogeneous parallel programs, while also allowing higher-order constructs on the GPU. Specifically, in this dissertation we develop cost semantics for region-based heterogeneous parallel functional programming languages. We do so by developing it for a core calculus $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ that is readily extendable to other functional programming languages.

In order to develop a cost semantics for such a language, we first develop a novel cost semantics for a region-based homogeneous-parallel functional language residing on the CPU, along with a novel trace-based cost semantics for a similarly region-based homogeneous-parallel functional language residing on the GPU. We then develop and present a way to glue together these two languages that reside on different computational realms—statically, semantically, and operationally—to arrive at a region-based heterogeneous parallel language, and a cost theory that is able to profile programs written in that language. We then implement and evaluate our theory and show that it accurately reflects the real-world runtime characteristics of source programs.

Contents

List of Figures xi

List of Abbreviations xv

1 Introduction 1
- 1.1 Thesis Statement & Contributions 3

2 Background and Related Work 7
- 2.1 Cost Calculi 8
  - 2.1.1 Cost Semantics for Parallel Functional Languages 11
  - 2.1.2 Cost Semantics for Cache Behavior 25
  - 2.1.3 Cost Semantics for Self-Adjusting Computation 32
  - 2.1.4 Other Cost Semantic Theories 33
  - 2.1.5 Cost Types and Non-Semantics-Based Cost Theories 34
- 2.2 Memory and Layout Theories 35
  - 2.2.1 Linear Ordered Types for Memory and Data Layout 35
  - 2.2.2 Types for Data Layout Inference 38
- 2.3 Region Based Memory Models 39
  - 2.3.1 Example: an effect typing system for the STLC with regions . 42
- 2.4 Architectural Layout of the GPU 44
  - 2.4.1 Low-Level GPU Programming Models 46
  - 2.4.2 Moving Memory Between Worlds 47
  - 2.4.3 Warp Divergence 51
  - 2.4.4 RBMM for High-Level Heterogeneous Languages 53

3 Cost Semantics for Region-Based Parallel Languages 54
- 3.1 Introduction 55
- 3.2 Syntax 56
- 3.3 An Effect Type System for Λ_C 58
  - 3.3.1 Region Annotations & Polymorphism 60

3.3.2 Latent Effects . . . 61
3.3.3 Effect Polymorphism . . . 63
3.3.4 The Type System . . . 65
3.3.5 Typing Heap Values . . . 73
3.4 Memory Model . . . 76
3.4.1 Region Based Memory Management . . . 76
3.5 The Abstract Machine Model . . . 79
3.6 Cost Semantics for Λ_C . . . 82
3.6.1 Cost Graphs . . . 82
3.6.2 Bounded Evolutions of the Region Stack . . . 85
3.6.3 Semantic Rules . . . 87
3.6.4 Schedules . . . 92
3.6.5 Determining Costs . . . 97

4 Cost Semantics for GPU Parallelism **100**
4.1 Introduction . . . 101
4.2 Restricting Λ_C for the GPU . . . 102
4.3 The GPU, Abstractly . . . 104
4.3.1 GPU Memory Model . . . 104
4.3.2 Abstract Machine Model for the GPU . . . 105
4.3.3 The Discrete Memory Machine Model . . . 106
4.3.4 The Unified Memory Machine Model . . . 108
4.3.5 Putting it all Together: the HMM Model . . . 109
4.4 GPU Cost Graph Combinators . . . 111
4.4.1 Thread Graphs and Meta-edges . . . 112
4.4.2 Warp Graphs . . . 113
4.4.3 Warp Pools . . . 114
4.4.4 Device Boxes . . . 115
4.5 Cost Semantics for Λ^G . . . 115
4.5.1 Meta Locations . . . 115
4.5.2 Reading and Writing to Memory . . . 117
4.5.3 Thread Semantics . . . 118
4.5.4 Global Cost Semantics . . . 120
4.6 Scheduling GPU Cost Graphs . . . 126
4.6.1 Thread Schedules . . . 126
4.6.2 Warp Schedules . . . 130
4.6.3 Device Box Schedules . . . 133

4.6.4 Keeping Track of the Region Stack . . . 134
4.7 Determining Costs from GPU Cost Graphs . . . 135

5 Cost Semantics for Heterogeneous Parallel Languages 142
5.1 Introduction . . . 143
5.2 Combining Λ_C and Λ^G . . . 143
5.2.1 Computational Worlds . . . 144
5.2.2 Syntax . . . 145
5.2.3 Syntax and Typing . . . 145
5.3 Transferring Memory Between Computational Worlds . . . 151
5.4 Connecting Machines . . . 156
5.5 Cost Semantics for Λ_G^C . . . 157
5.5.1 Scheduling . . . 159
5.5.2 Determining Costs . . . 162
5.6 Wrapping Things Up . . . 165
5.6.1 Discussion: Effect Polymorphism . . . 165
5.6.2 Discussion: Costing of Device Boxes . . . 166

6 Implementation and Evaluation 168
6.1 Introduction . . . 169
6.2 The Compilation Pipeline . . . 169
6.2.1 Typechecking . . . 169
6.2.2 Profiling . . . 170
6.2.3 Normalization . . . 170
6.2.4 Closure Conversion and λ-Lifting . . . 171
6.2.5 Retyping . . . 173
6.2.6 Explicitly Passing the Heap . . . 173
6.2.7 Threading . . . 177
6.2.8 Lowering & Code Generation . . . 180
6.3 Profiling . . . 181
6.3.1 Representing Memory . . . 181
6.3.2 CPU . . . 182
6.3.3 GPU . . . 190
6.4 Primitive Cost Providers . . . 197
6.4.1 Deriving the ATPR . . . 199
6.5 Evaluation . . . 204
6.5.1 Comparing to Real-World Runtimes . . . 209

7 Conclusions and Future Work **213**
7.1 Limitations 214
7.2 Future Directions 215

Appendices

A Index of Notation **220**

B Proofs of Theorems **226**
B.1 Proof of Theorem 3.6.3 226
B.2 Proof of Theorem 3.4.1 227
B.3 Proof of Theorem B.3.1 228
B.4 Proof of Theorem 4.6.1 230
B.5 Proof of Theorem 5.2.1 231
B.6 Proof of Theorem 5.3.2 231

References **233**

List of Figures

2.1 Cost combinators on DAG-based cost models. 15
2.2 How the ideal cache model works. 27
2.3 Three different ways of laying out $(1,(2,3))$. 36
2.4 How memory is allocated for $y = (1,(2,3))$ using the first representation in Figure 2.3. 37
2.5 The syntax for a simple region-based function language adapted from Henglein, Makholm, and Niss [49] §3.5. 43
2.6 The type- and effect-system for the simple region-based language adapted from Henglein, Makholm, and Niss [49] §3.5. 44
2.7 Parallel vector addition in CUDA C (a) & (b), and in C using OpenMP (c). Note that in (a) we access the array using the thread index `threadIdx` and thus this code runs on each thread, whereas in in (b) we access it based upon the `blockDim` and therefore multiple results will be computed by the same thread. 48
2.8 Warp divergence due to branching instructions. 52

3.1 Syntax of Λ_{C} 57
3.2 Metafunctions over the syntax of Λ_{C}. 59
3.3 Size metafunction over the values in Λ_{C}. Note that boolean and integer arrays are stored unboxed. 60
3.4 Example adapted from Tofte and Talpin [8] showing how dangling pointers can be created in a RBMM system. ρ_1, ρ_2, and ρ_3 are allocated outside this section of code. Types are elided for brevity. 62
3.5 Translation of truncated effect polymorphism to Λ_{C}. 66
3.6 Well formedness rules for contexts 68
3.7 Well formedness rules for regions and effects 69
3.8 Well formedness rules for types 70
3.9 The base typing rules for Λ_{C} based on the Bounded Effect Calculus of Fluet and Morrisett [65] 71

3.10 The typing rules for Λ_{C} (continued). 72
3.11 Definition of runtime environment projection and embedding. 74
3.12 Typing rules for values. 75
3.13 Allocation and Reading 78
3.14 Layout of the PRAM model. 81
3.15 Cost graph operations 83
3.16 Cost combinators on DAG-based cost models. 84
3.17 Allocation and Reading Costs 85
3.18 Cost semantics for the serial portion of Λ_{C} 88
3.19 Cost Semantics for Λ_{C} (Cont.) 89

4.1 Syntax of Λ^{G} 104
4.2 Layout of both the DMM and UMM models. Note how the only difference is the layout of the address line. 107
4.3 Address groups, and how they relate to the memory banks of the DMM model. 109
4.4 Layout of the HMM model. 110
4.5 Definitions of the abstract machine states used for the GPU. 111
4.6 Thread graph definitions. 114
4.7 The various stages and layers that make up a device box. Each c_i is a thread graph, and the GPU is assumed to have warps and memory banks of width w, with p threads per work-group. The semantics of these definitions are explained in subsequent sections. 115
4.8 Read and write judgements for the GPU. Note that the location that has been read or writing is always returned and that no cost is associated with the actual judgement. 119
4.9 Local (thread) cost semantics for the GPU. 121
4.10 Local (thread) cost semantics for the GPU. (Continued) 122
4.11 Global cost semantics for the GPU. 125
4.12 Definition of the warp step relation in terms of schedulable and scheduled nodes as defined in Section 4.6.1. 132
4.13 Definition of the warp pool step relation in terms of warp steps where $\mathcal{W}(n) = (\kappa, \ell, \blacksquare\ell, \langle \mathsf{W}_n, \ldots, \mathsf{W}_{l+n} \rangle^{\sigma})$. 133
4.14 Definition of the device box step relation in terms of warp pool steps. `unzip :: Set (a,b) -> (Set a, Set b)` takes a set of tuples and returns a tuple of sets. 134

5.1 The syntax of $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ with the syntactic changes to Λ_{C} highlighted. 146
5.2 Modified typing rules for $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ based on the type system for Λ_{C} given in Figure 3.9. 148
5.3 Modified typing rules for $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ based on the type system given for Λ_{C} in Figure 3.10. 149
5.4 The process by which a region (R_1') that resided on G is merged back in with its parent region (R_1) that resides on C, and where both R_1 and R_1' evolved from (are bounded evolutions of) a parent region R_{parent} at the time the region was copied to the GPU. 154
5.5 Definition of $\widetilde{\mathsf{frv}}(E, e)$. Note that since $e \in \Lambda^{\mathsf{G}}$ we do not handle the `letregion` nor the `on` forms. 161

6.1 Calculated computational and space usage for the parallel Fibonacci functions. One using only one region, and the other using a new region at each recursive step. 207
6.2 Calculated computational and space usage for parallel addition over a small fixed-size array. Notice that we rightly determine that the cost of running on the GPU is much more costly, and similarly, that the useful amount of parallelism available to us on the CPU as we use more thread does not help due to the computational overhead of these threads. 208
6.3 Calculated computational and space usage for the parallel addition of vectors. Notice that the CPU graph grow almost linearly with the size of the array (recall that the threads are fixed), while the GPU code is predicted to stay almost constant in comparison. 209
6.4 Calculated computational and space usage for the parallel addition of vectors on the GPU. One (BRANCHGPUSUM) uses a branching operation during the summation, to perform addition of a different constant based upon parity of the elements. The other (GPUSUM) simply performs straightforward addition. 210
6.5 The ratios between using only one region and many (one for each resursive call) for the parallel Fibonacci function with $N = 10$. The blue line ("Real") are the observed ratios (both runtime and space) from the execution of the compiled function at the given thread counts. The other orange line ("Calculated") are the ratios between the calculated space usage and computational times for this function at the given thread counts. 211

6.6 The ratios between vector addition on the CPU and GPU. The green and red lines are the observed ratios (both runtime and space) from the execution of the compiled functions for addition (CPU, GPU, and GPU with branching). The other orange and blue lines are the ratios between the calculated space usage and computational times for the compiled vector addition functions. 212

List of Abbreviations

CPU . . . . . . Central Processing Unit, 1

GPU . . . . . . Graphical Processing Unit, 1

SIMT . . . . . . Single Instruction Multiple Thread, 5

GHC . . . . . . Glasgow Haskell Compiler, 8

DAG . . . . . . Directed Acyclic Graph, 10

PRAM . . . . . Parallel Random Access Machine, 12

SECD . . . . . . (Stack, Environment, Control, Dump) Machine, 12

SIMD . . . . . . Single Instruction Multiple Data, 37

LIFO . . . . . . Last In First Out, 39

RBMM . . . . . Region-Based Memory Model/Management, 39

STLC . . . . . . Simply Typed Lambda Calculus, 42

DRAM . . . . . Dynamic Random Access Memory, 45

SMP . . . . . . Streaming Multiprocessor, 45

ADT . . . . . . Algebraic Data Type, 53

PCP . . . . Primitive Cost Provider, 90

DMM . . . . . . Discrete Memory Machine, 105

UMM . . . . . . Unified Memory Machine model, 105

HMM . . . . . Hierarchical Memory Machine model, 105

ATPR . . . . . . Abstract To Primitive Ratio, 197

1 Introduction

Contents

1.1 Thesis Statement & Contributions 3

In the modern world with many different computational resources available to the programmer, it can be difficult to determine which should be utilized for a given program. Furthermore, as the languages that are utilized become ever further abstracted from the underlying parallelism and hardware, the need for systems that can automatically determine the costs of programs running on different computational resources becomes ever greater.

In order to automatically determine the costs of such programs, a way of determining the costs of these programs under different computational interpretations is needed; and a cost model provides the formal means by which we can measure such costs. As a step towards answering these questions, this dissertation develops a cost semantic theory and cost model for high-level functional programming languages with heterogeneous parallel constructs.

As things stood before the work presented here, given a program that could run on either a Central Processing Unit (CPU) or Graphics Processing Unit (GPU), the only means of determining where it should run are based on either empirically evaluating multiple runs of the program and using heuristics to determine how the computation should be distributed, or based purely on programmer know-how, instinct, and manual placement. While these methods can often work with good results, if we wish to be able to formally reason about and utilize these in a semantically sound, and formal context such “best guess”, “gut feeling”, and non-autonomous methods will not do.

If we want a formal way of reasoning about costs, in the homogeneous case we are almost spoiled for choice, and there are many theories that have been quite well developed such as amortized resource analysis [1], sized types [2], and cost semantics [3–6]. However, as we use more modern domains of computation we quickly run out of options: if we wish to analyze a parallel homogeneous program, cost semantics [6], and more empirical methods are the only tools still useful to us; and if we try to add any type of heterogeneity we find that there is no way to successfully profile our program other than through empirical and “best guess” methods. Moreover, as the user-visible lines between heterogeneous parts of the computation become increasingly blurred—with some languages practically eliding the distinction between CPU and GPU computations [7]—our lack of ability to formally profile heterogeneous parallel computations is a problem that will only get worse.

Building on the work of cost semantics [5, 6] this dissertation presents a theory that allows programmers to formally analyze high-level heterogeneous parallel programs and allows compiler writers to formally reason about, and score the optimizations that they perform during the compilation process. While developing such a theory offers a number of advantages and opens up many new opportunities for programmers and compiler writers alike, it also presents a number of challenges, and care

must be taken to account for a number of different computational patterns that can have significant impacts on the computational complexity of programs. In particular, a sufficient profiling theory must be able to account for the cost of memory transfers from the CPU to the GPU; for the different costs of parallelism on the CPU and GPU; for the different costs associated with the same computational structures on the CPU versus the GPU; and finally, for how different types of data might have different cost implications on the CPU and the GPU. Moreover, since we are interested in profiling higher-order functional heterogeneous parallel languages, we need to be able encode higher-order constructs in the GPU-runnable segments of our programs, and be able to take these into account in our cost system.

In order to allow higher-order constructs on the GPU, we utilize a region-based memory model [8] for the language we develop [7, 9]. As we will see later on, a region based memory model not only allows us to represent higher-order constructs on the GPU, but also simplifies the process of reasoning about space usage and memory transfer costs in our cost framework. Furthermore, the use of a region based memory management system makes the theory presented readily applicable to other recent developments in the programming languages community both for low-level languages such as Rust [10], and for distributed computing [11, 12] where grouping data with similar lifetimes together can provide significant benefits in terms of memory transfer, reclamation, and safety guarantees.

1.1 Thesis Statement & Contributions

The thesis statement for this dissertation is the following:

> *Cost semantics are a viable and accurate model for analyzing and profiling high-level higher-order heterogeneous parallel functional languages.*

> *Further, such cost semantics allow the determination of relative (parallel) costs of programs running on either the CPU or GPU and the ability to inform computational placement of these programs.*

While the work that is presented will provide the ability to determine relative costs of programs, there are still a number of different directions to take this work. In this sense, the work presented can be seen as foundational in nature; this work provides the necessary techniques and models that other areas can utilize to extend their work to take into account the different computational architectures available to modern-day programmers.

In particular, while cost models and cost semantics can be useful in and of themselves, they are also widely used as an underlying means of determining how much a given "program" costs; areas such as Cost Autonomous Mobility Systems (CAMS) or amortized complexity analysis require a cost semantics in order to handle the concept of "cost" of the programs that they work on. One of the consequences of this dissertation—having a cost semantics for heterogeneous parallelism—gives researchers in these fields the ability to build on the work presented and take into account heterogeneous parallelism in such systems.

The rest of this dissertation is laid out in the following manner. In Chapter 2 we give an in-depth overview of the current state of cost semantics, and other means of determining costs from programs with the aim of creating a natural thought progression into the research. After this, in Chapter 3, we introduce a region-based parallel language that we will be building on for the rest of the dissertation and detail an effect-typing system for the language. We also describe the layout of the regions, and invariants about the region-typing system and the region stack as they relate to the cost semantics. We then present the cost semantics for this language and introduce weighted cost graphs and their scheduling.

In Chapter 4, we present a cost semantics for a high-level parallel language running on the GPU—indeed almost the same as the CPU language with the exception of region allocation. As opposed to the cost semantics that we saw in Chapter 3, the cost semantics for the GPU portion of the language are *trace based* due to the highly contextual cost factors that need to be accounted for. In Chapter 5, the theory and languages that were introduced in Chapters 3 and 4 are combined and we describe how to statically restrict invalid GPU portions. We also show how to connect the two underlying abstract machine models for both the CPU and the GPU, and how to reason about memory transfers between the CPU and GPU. After this, we show how to perform cost analysis within the combined heterogeneous parallel language. We then wrap up in Chapter 6 by presenting a prototype implementation of the language, and evaluating a profiling system based on the theory presented in Chapters 3-5, and conclude with future directions and possible extensions of this work in Chapter 7. Additionally, Appendix A contains an index of the notation used in this dissertation.

To be specific, this dissertation makes the following contributions:

1. We define cost models and cost semantics for region-based parallel languages on the CPU, weighted cost graphs, and parameterized schedules for these weighted cost graphs.

2. We show how to derive cost models for memory transfers between host and device, and how to mix both *derived* and *abstract* costs.

3. We develop a cost model for GPU and Single Instruction Multiple Thread (SIMT) parallelism that takes into account work-group shared memory and memory characteristics based on a trace-based cost semantics, and delayed computations.

4. We develop a cost model and semantic theory for the GPU that allows us to profile device-side launching of kernels, or “nested” GPU parallelism; kernels executing on the GPU can themselves spawn other kernels to run on the GPU.
5. We develop a cost model for region-based functional languages with heterogeneous parallel constructs, and present a cost theory that takes into account memory transfers and computational migration between host and device.

2

Background and Related Work

Contents

2.1 Cost Calculi **8**
- 2.1.1 Cost Semantics for Parallel Functional Languages 11
- 2.1.2 Cost Semantics for Cache Behavior 25
- 2.1.3 Cost Semantics for Self-Adjusting Computation 32
- 2.1.4 Other Cost Semantic Theories 33
- 2.1.5 Cost Types and Non-Semantics-Based Cost Theories 34

2.2 Memory and Layout Theories **35**
- 2.2.1 Linear Ordered Types for Memory and Data Layout 35
- 2.2.2 Types for Data Layout Inference 38

2.3 Region Based Memory Models **39**
- 2.3.1 Example: an effect typing system for the STLC with regions 42

2.4 Architectural Layout of the GPU **44**
- 2.4.1 Low-Level GPU Programming Models 46
- 2.4.2 Moving Memory Between Worlds 47
- 2.4.3 Warp Divergence 51
- 2.4.4 RBMM for High-Level Heterogeneous Languages 53

2.1 Cost Calculi

Cost semantics as we know them today were introduced by Sansom in his dissertation [13] and in a paper with Peyton Jones [3] as a way to profile higher-order lazy functional languages. The cost-semantic-system was designed to have an abstract—and hence extensible—notion of cost, be syntax directed (i.e., extensional and compositional), and based on the operational semantics of the language. At this point, the main interest was in using cost semantics coupled with *cost centers* that the programmer added to their program in order to better enable profiling, as well as to enable compiler writers to prove invariance of costs (with respect to the user-annotated cost centers) under various transformations. However, there was a secondary goal that the cost semantics for a language "be a guide to implementers and a final arbiter of obscure cases" which can frequently arise in higher-order functional languages. While the use of cost centers coupled with cost semantics is useful from a language-profiling perspective, they are not needed if one simply wishes to reason about the cost of one's program as a whole, and as we will see later (cf. Spoonhower et al. [6]), the current technology no longer requires cost centers in order to profile languages.[1]

Even though user-annotated cost-centers did not continue to be used in cost-semantic theories, Sansom's idea of extending the operational semantics of the language to take into account the costs of evaluation was an important one: up to this point, the idea of a formal cost model for a language (if there was such an idea) was synonymous with automatic complexity analysis, with many people focusing on how to create a system of equations from the source program that would later be solved to give the overall complexity of the program [15]. This research is most similar to the modern day use of sized types, and type theories for complexity analysis. We don't

[1] Although it is worth noting that GHC uses a version of these to allow the user to profile their programs [14].

use any of these methods explicitly in the dissertation, but they are useful to know in the broader context of formalized cost models for languages. We will therefore discuss some of the historical, and more recent non-semantics-based theories and how they fit in with cost semantics briefly in Section 2.1.5.

The usual presentation of a cost semantics is through an annotated big-step semantics in which a cost (or costs) are associated with each reduction rule. For instance, a run-of-the-mill cost semantics for the untyped λ-calculus where $E; e \Downarrow v; c$ means that the expression e evaluates to the value v with cost c in environment E might look like:

$$\frac{E(x) = v}{E; x \Downarrow v; 1}\ \text{Var} \qquad \frac{}{E; \lambda x.e \Downarrow \lambda x.e; 1}\ \text{Lam}$$

$$\frac{E; e_1 \Downarrow \lambda x.e; c_1 \qquad E; e_2 \Downarrow v'; c_2 \qquad E[x \mapsto v']; e \Downarrow v; c_3}{E; e_1\ e_2 \Downarrow v; c_1 + c_2 + c_3}\ \text{App}$$

Wherein we say that looking up a variable in the environment and reducing a lambda expression have unit cost. The cost of applying e_1 to e_2 is the sum of the costs of evaluating e_1 and e_2 to values, plus the cost of evaluating the body of the function in the environment that maps x to the reduced value of e_2. While these rules are remarkably simple for now, as we will see later these semantics can become quite complex depending upon the costs that we wish to describe.

The formalization of the cost of a program in terms of its semantic contents permits easy analytical reasoning about that cost: the semantic rules for the language give a way to precisely *ask* and then *answer* questions about efficiency that would otherwise be hard, if not impossible, to discuss. The fact that the cost semantics is syntax-directed and uses the source language is crucial to the ease of reasoning that a cost model is meant to provide; the programmer need not translate into—or even understand—the denotational semantics or low-level translations for their language.

Thus the programmer (and compiler) can easily ask and answer efficiency questions the way they have always done: by looking at the syntax of the program [16]. However, by using a cost semantics, they can do this precisely and be confident in their conclusions. Moreover, the syntax-directed nature of the semantics means that compiler writers can easily reason—formally—about how the optimizations they write affect the cost of the underlying program based upon the syntactic transformations that the optimization performs.

Cost semantics are useful for the programmer and compiler writer to reason abstractly about the costs of a program. However, abstract costs are not very useful unless we can somehow relate these abstract costs to more explicit costs. In particular, a cost semantics needs to be mapped down to an abstract (or physical) machine model while at the same time preserving costs. This mapping of the source-level and abstract cost semantics to the abstract machine amounts to enforcing that all implementations of a language must respect the source-level cost bounds. Thus, a compiler preserving these costs means that we can *prove* time and resource bounds on the language and the implementation of that language. The choice of the underlying abstract machine is important, since it often affects the structure of the rules for the cost semantics as well as the structure of the costs themselves. The source-level cost semantics of Sansom and Peyton Jones [3] is not affected in any way that stands out to the reader, since the underlying machine model has been designed to replicate the source-level semantics i.e., there are not that many *intensional* properties of the program that are being described by the costs. As we will see, as the properties that we wish to describe become more intensional—such as parallelism (Section 2.1.1), and especially I/O efficiency (Section 2.1.3)—the more important the underlying abstract machine and machine model becomes, with fragments of the underlying machine appearing at times in the source-level cost semantics.

One of the interesting research questions that arises here is to determine a metric by which we can determine when certain properties become *too* intensional thus making the source-level cost semantics unreadable and how to combat this unreadability at the sake of intensionality. This is one of the secondary questions that we explore in the course of this dissertation.

2.1.1 Cost Semantics for Parallel Functional Languages

One of the main uses of cost semantics since its inception has been to describe parallel costs for programs. A cost-semantic approach is well suited to the challenges that parallel languages present since one doesn't need an underlying parallel big-step semantics in order to describe the costs or amount of parallelism for a given reduction sequence. Instead, the parallelism is described by the combinators on the costs associated with the semantics. This essentially amounts to "pushing" the parallelism from the semantics into the cost measures via a Directed Acyclic Graph (DAG) (or, in the early days, via spawn-tree depth counters). As we will see, this relocation of parallel information to the cost measure leads to elegant descriptions of schedules, time- and space-bounds, and possible parallelism.

The first time that cost semantics are used in the context of studying parallelism is by Blelloch and Greiner [17] in order to study the amount of parallelism in eager sequential functional languages—specifically, the λ-calculus with call-by-value semantics (or what Blelloch and Greiner [17] call the PAL model). However, in order to do this, two cost measures are used: one to keep track of the total amount of work done in the computation, and the other to keep track of the parallel (spawn-tree) depth of the computation in order to study the amount of parallelism available. Due to the fact that the language is inherently serial, the offloading of the parallel-cost reasoning to the cost measures for the semantics is ideal; pushing the parallelism into the cost measures means that the "parallel" constructs in the language are no

longer defined by the reduction rules, but by the combinators for the costs in the semantics.

In order to effectively offload the parallelism to the cost measures, *integer* counters are introduced that track the depth of the spawn-tree of the program as well as the total amount of work done (this as opposed to the cost being the spawn-trees themselves). Combining parallel costs (i.e., spawn-tree depths) then amounts to taking the greater of the two spawn-tree depths, and combining the amount of work done in both spawn-trees is the sum of the amounts of work done in both. As an example, consider the rule $E; e \Downarrow v; w; c$ which is read as "in the environment E, the expression e evaluates to the value v with work w and spawn-tree depth d". The application rule would look like the following:

$$\frac{E; e_1 \Downarrow \lambda x.e; w_1; d_1 \qquad E; e_2 \Downarrow v'; w_2; d_2 \qquad E[x \mapsto v']; e \Downarrow v; w_3; d_3}{E; e_1\ e_2 \Downarrow v; w_1 + w_2 + w_3 + 2; \max(d_1, \max(d_2, d_3)) + 2}\ \text{App} \tag{2.1}$$

Where we are saying that the amount of work in the application of e_1 to e_2 is the amount of work to evaluate and then apply both of expressions, and that the depth of the spawn-tree is the maximum of the two depths that we get from evaluating e_1 and e_2. The addition of 2 to both the work and depth costs is so that an exact correspondence can be made between the work and spawn-tree depth and the number of states in the abstract machine that is used to simulate evaluation. This is the first instance we see of an artifact of the underlying abstract machine model leading to a change in the source-level cost semantics. However, in reality we are not that concerned with constants like this appearing in the rules (beyond how it affects the way we map down to the abstract machine) since we are only concerned with the asymptotic complexity of the program.

Many times it is useful to map one cost semantics to multiple different machines in order to explore the potential parallelism in the different machine models, as well as to prove the correctness of the abstract cost semantics. In particular, Blelloch and

Greiner [17] map the abstract cost semantics down to an SECD machine [18], and then to a parallel version of the SECD machine—the P-ECD machine—in which the single state of the SECD machine is changed to a set of states that can be evaluated in parallel at each step. Blelloch and Greiner then map these two separate abstract machine representations down to serial—RAM [19]—and parallel—PRAM [20], butterfly network, and hypercube—machine models. It is by this mapping of the abstract costs to concrete costs that a *verified implementation* (with respect to that cost model) of the source language can be extracted.

It is important to note at this point that the parallelism being offloaded to the costs (in the source-level cost semantics) forces the underlying machine model to implement certain operations in parallel. Essentially, the parallelism in the source-level costs means that any faithful compilation of the source language to an abstract machine that respects those costs must implement the parallelism that we desire in the source language. This is what we see in the above mappings; first mapping down to the SECD machine (the degenerate "serial parallel" case) and then generalizing the serial operations of the SECD machine to be parallel in the underlying abstract machine with the P-ECD machine and butterfly and hypercube networks.

Although the cost semantics presented by Blelloch and Greiner [17] doesn't use a DAG to track the dependencies; instead combining computation depths via the $\max$ function and explicitly tracking the amount of work being done via a separate cost metric, the field quickly moved to using a DAG-based model for measuring both work and parallelism.

The first instance of using DAGs to keep track of the parallelism and amount of work done in a language is by Blelloch and Greiner [21] who use DAG-based costs to describe time- and space bounds for the *parallel language* NESL [22]: the edges of the DAG represent control dependencies and the nodes represent the units of computation. Thus the total amount of work is the number of nodes in the DAG,

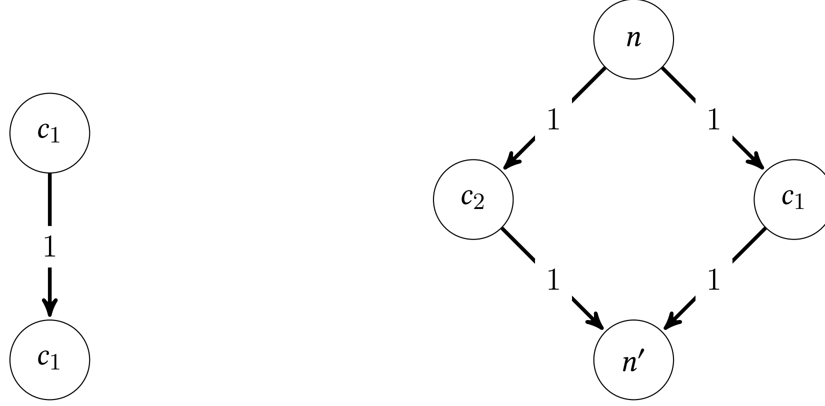
and the amount of parallelism is the width of the DAG. Despite the fact that NESL is a parallel language, the big-step semantics for it is still sequential; once again the parallelism is created (or tracked) via the cost-combinators for the language which will enforce parallelism in the underlying abstract machine model.

The DAGs that are used for describing the costs are special, in that each cost $g = (n, n', D)$ represents a DAG D coupled with a *unique* initial node n and *unique* terminal node n'. The combinators that we use to represent parallelism in these costs now change from those we saw earlier [17]—from ones that work on integers, to ones that determine how DAGs can be merged and composed. In particular, there are two major combinators: $\oplus$ or sequential composition; and $\otimes$ or parallel composition. An equivalent model for the example given in Equation (2.1) using these combinators might now look like:

$$\frac{E; e_1 \Downarrow \lambda x.e; c_1 \qquad E; e_2 \Downarrow v'; c_2 \qquad E[x \mapsto v']; e \Downarrow v; c_3}{E; e_1\ e_2 \Downarrow v; (c_1 \otimes c_2) \oplus c_3}\ \text{App}$$

Where we are saying that e_1 and e_2 can be evaluated *in parallel* with costs c_1 and c_2 and then evaluation has to be serialized and can then continue with cost c_3. Since the costs are DAGs with unique initial and terminal nodes, the $\oplus$ and $\otimes$ combinators serve as ways of merging and composing these two DAGs, using the terminal and initial nodes as the join-points for the new DAG. For an example of how these combinators join the two cost graphs see Figure 2.1.

It is through the representation of costs by DAGs that parallelism is offloaded from the semantic model of the language, and on to the cost model of the language. This representation then provides a framework in which it is easy to reason about time-bounds since the total amount of time that is taken is proportional to the height of the DAG. However, while the computation/cost DAGs will be invaluable for determining parallel space bounds the process of determining them is not as simple as that for time-bounds.



The serial composition of DAGs c_1 and c_2 given by $c_1 \oplus c_2$.

The parallel composition of DAGs $c_1 = (n_1, n_1', c_1)$ and $c_2 = (n_2, n_2', c_2)$ given by $c_1 \otimes c_2$. Note that both n and n' are newly created ("fresh") nodes in the graph.

Figure 2.1: Cost combinators on DAG-based cost models.

In order to determine space bounds based upon the cost semantics, an extra cost metric needs to be added to track the space that the computation takes. Thus, the rules need to be updated to take into account this new cost that needs to be measured. We therefore augment the rules with both a store σ and a cost s that measures the amount of space that the reduction takes. Our cost-semantic judgement now looks like $E; \sigma; R; e \Downarrow v; \sigma'; c; s$ where R is the set of *root locations* that point to the data needed by the continuation. The space s is then defined in terms of the number of root locations for the graph and the current memory σ (much like garbage collection):

$$space(R, \sigma) = \sum_{l \in S} \begin{cases} |\overline{v}| & \text{if } \sigma(l) = \overline{v} \\ |FV(e)| + 2 & \text{if } \sigma(l) = \lambda x.e \end{cases} \tag{2.2}$$

where $S = \bigcup_{l \in R} locs(l, \sigma)$, and

$$\begin{array}{rcl} locs(c,\sigma) & = & \{\} \\ locs(l,\sigma) & = & \{l\} \cup locs(\sigma(l),\sigma) \\ locs(\overline{v},\sigma) & = & \bigcup_{i=0}^{n-1} locs(v_i,\sigma) \\ locs(\lambda x.e,\sigma) & = & \bigcup_{l\in L} locs(l,\sigma) \\ & & \text{where } L = E(FV(e) - \{x\}) \end{array} \tag{2.3}$$

In essence, the *space* function determines the amount of space reachable from a set of root nodes R i.e., the space of all things that still need to be kept alive (in the current computation graph) since we could possibly need to access them in the continuation. Also, note that we have added vectors to our language. This addition is an important one, since in NESL the parallelism is not introduced by the λ-calculus part of the language (i.e., the λ-calculus part of the language is serial). Instead parallelism is introduced by parallel array comprehensions $\{e' \ : \ x \texttt{ in } e\}$, and various parallel array operations.[2]

Now that both the reduction rules have been augmented to track space, and we have defined the *space* function, we are able to present a cost semantics that tracks not only time and work but also space-usage:

$$\frac{}{E;\sigma;R;c \Downarrow c;\sigma;1;space(R,\sigma)}\ \text{Const} \qquad \frac{}{E;\sigma;R;x \Downarrow E(x);\sigma;1;space(R,\sigma)}\ \text{Var}$$

$$\frac{l \notin dom(\sigma)}{E;\sigma;R;\lambda x.e \Downarrow l;\sigma[l \mapsto Clos(E,_,x,e)];1;space(R\cup\{l\},\sigma)}\ \text{Lam}$$

$$\frac{\begin{array}{cc} E;\sigma;R\cup E(FV(e_2));e \Downarrow l;\sigma_1;c_1;s_1 & E;\sigma_1;R\cup\{l\};e_2 \Downarrow v_2;\sigma_2;c_2;s_2 \\ \sigma_1(l) = Clos(E',x',x,e) & E'[x'\mapsto l][x\mapsto v_2];\sigma_2;R;e \Downarrow v;\sigma_3;c;s \end{array}}{E;\sigma;R;e_1\ e_2 \Downarrow v;\sigma_3;1\oplus c_1\oplus c_2\oplus 1\oplus c;\max(s_1+1,s_2+1,s)}\ \text{App}$$

$$\frac{\begin{array}{c} E;\sigma;R\cup E(FV(e'));e \Downarrow l;\sigma_0;c;s \qquad \sigma_0(l)=\overline{v} \qquad n=|\overline{v}| \qquad l'\notin dom(\sigma) \\ E[x\mapsto v_j];\sigma_j;R\cup E(FV(e')-\{x\});e' \Downarrow v'_j;\sigma_{j+1};c_j;s'_j \qquad \forall j\in\{0,\ldots,n-1\} \end{array}}{E;\sigma;R;\{e' \ : \ x \texttt{ in } e\} \Downarrow l';\sigma_n[l'\mapsto\overline{v}];c\oplus\left(\bigotimes_{j=0}^{n-1}1\right)\oplus\left(\bigotimes\overline{c}\right),\max(s,n+\texttt{max}(\overline{s'}))}\ \text{Each}$$

[2] For this literature review. The core of NESL [21] is in fact larger and includes conditionals and other forms but this subset is a sufficient subset to describe their theory.

All of the rules with the exception of *Each* are fairly straightforward, since no parallelism is introduced in them and the only cost that needs to be tracked is space. The *Each* rule however is more interesting, since we are dealing with both the introduction of parallelism, and with arrays: in the *Each* rule we are saying that we first evaluate the expression to a sequence $\overline{v}$ and then the body e' is evaluated in the environment extended with $x \mapsto v_j$ for each value v_j in $\overline{v}$. Even though each subcomputation is independent and hence can be executed in parallel, the store is threaded through serially. This serialization of the stores in the semantics is an important design decision, since it specifies that we will measure space sequentially (even though computation may happen in parallel), since different orders of execution can effect the amount of space used by the program.

At this point, the semantics have been augmented with a way to measure the space of the computation given a store and a set of root locations. This then means that we can answer questions about *serial* space usage and space-bounds by simply looking at the space cost s at each step of the computation—in this case at each step that the abstract machine takes. This is formalized through the idea of *reachable space at a given step* in the underlying abstract machine (a parallel version of the CEK machine [21]) that the language is mapped down to: the reachable space $S_r(Q_i, \sigma_i)$ at a step i of the machine is the sum of

1. the *queue space* $S_q(Q_i) := 1 + |E| + |K|$ for states $(e, E, K) \in Q_i$ and
2. the *store space* $S_\sigma(Q_i, \sigma_i) := space(L(Q_i), \sigma_i)$

where $L(Q_i)$ returns the root values for the step i in the machine. With this formalization, we are then able to say that if

$$\bullet_E; \bullet_\sigma; \bullet_R; e \Downarrow v; \sigma; g; s$$

in n steps, then we can bound the *serial* space used by the machine for some constant k:

$$\max_{i=1,\ldots,n} S_r(Q_i, \sigma_i) \leq ks \tag{2.4}$$

However, we are not yet able to answer questions about *parallel* space usage and space-bounds. In order to do that we need to introduce some more machinery.

Since the schedule with which tasks are run (and the future tasks that are scheduled) affects the number of nodes in the DAG that can be run at any given moment in time, the schedule that we choose can have significant effects on the space usage of the program [6]. Thus, before we can discuss the parallel space usage of the program, we not only need to have a fixed way in which we inspect the stores in a parallel setting (which we've already done above) but also fix a schedule for the tasks in our DAG. Thus, a scheduling policy needs to be introduced before we can start answering the question of "how much space is used at a given step in the underlying abstract machine in the parallel case".[3]

In order to discuss the scheduling of tasks in the computation DAG, we need to define the traversals that we can perform on the graph; in this case p-traversals. A *p-traversal* of a DAG is a sequence of $k \geq 1$ steps, such that for each step i, there is a set of nodes V_i in the DAG that are scheduled (i.e., visited) at this step and such that the V_i partition the DAG, if n is an ancestor of n' and $n' \in V_i$, then $\exists j < i \,.\, n \in V_j$ (i.e., all ancestors of a node must be scheduled before the node), and $|V_i| \leq p$. A *greedy p-traversal* based on a 1-traversal T_1 is the traversal that schedules the first p earliest nodes in T_1, a *premature* node then is a node that appears in a prefix σ_p of the greedy p-traversal of the graph, that does not appear in the prefix of the greedy 1-traversal that contains only nodes in σ_p. We can now bound the number of premature nodes in any p-traversal: for any DAG of depth d and 1-traversal, the maximum number of premature nodes in the greedy p-traversal is $(d-1)(p-1)$.

[3] As we will see the scheduling policies can change [6] (i.e., the space usage can be parameterized by the schedule), but for now the schedule is fixed.

Now that we have notions of p-traversals and premature nodes in the computation DAG—and most importantly, bounds on the number of premature nodes in a greedy p-traversal—we can turn to space bounds in the parallel case: we define the schedule for the computation DAG g where A is an array that originally contains the root node of g by

1. Scheduling the first $\min(p, |A|)$ nodes from A
2. Replace each newly scheduled node by its ready children, in left-to-right order, in place in A
3. Repeat steps 1. and 2. until all nodes have been scheduled.

This amounts to a depth-first traversal of the graph, where we are advancing in steps of at most p nodes at each step. We can now say that if we have q processes, and

$$\bullet_E; \bullet_\sigma; \bullet_R; e \Downarrow v; \sigma; g; s \tag{2.5}$$

in n steps, and if g is of depth d then we can bound the *parallel* space used by the machine for some constant k:

$$\max_{i=1,\dots,n} S_r(Q_i, \sigma_i) \leq k(s + dq) \tag{2.6}$$

While the fixing of a particular schedule for the computations in the DAG was an important part of the work by Blelloch and Greiner [21], it is more restrictive than we would like. Moreover, we can't explore the question of just how much of an affect the scheduling policy has on the time- and space-usage of the program. Spoonhower et al. [6] explore exactly these questions by creating a cost-semantic framework for time- and space profiling for parallel functional programs. In order to be able to deal with the complexities of answering the same questions but with different schedules, we need to update the way we represent space costs in the semantics.

We update the representation of the space cost much like the way we updated the temporal cost representation earlier—from a numeric value to a DAG or *heap graph*: the cost representation for space is now a DAG, where the nodes point to their data dependencies (i.e., they point "backwards in time"). However, this graph differs from the computation DAG, in that some nodes are shared with computation DAG nodes (the node that allocates that data) and there are others that are not shared—locations. Specifically, each edge in the heap graph represents a dependency on a *value*. Thus edges from a node n (in the combined heap/computation graph) to a location ℓ means that the computation at node n depends on the value found at the location ℓ, and similarly, an edge from a location ℓ' to ℓ means that the value at ℓ' depends on the value at ℓ. The distinction between the two types of nodes in the heap graph is important in reasoning about how space may be reclaimed during the execution of the program: an edge from ℓ' to ℓ means that the memory associated with ℓ can't be reclaimed until after ℓ' is no longer in use; an edge from n to ℓ means that the memory associated with ℓ can't be reclaimed until the expression that n represents has been evaluated. The fact that we are sharing nodes between the heap graph and the computation graph is an important one, since as we will see later, the relation between the computation graph and the heap graph will prove critical to how we analyze the space usage based upon the scheduling policy that has been chosen.

While we are freeing ourselves from the previous restriction of only one valid scheduling policy, we still need to restrict the schedules that constitute valid scheduling policies for the computation graph. This restriction manifests itself in a restriction on the partial order $\triangleleft$ that a given schedule will induce: the *schedule order* $\trianglelefteq$ for the graph g is defined on g's nodes such that $\forall n \in g \;.\; n \trianglelefteq n$, and $\forall n_1, n_2 \in g \;.\; n_1 \prec_g n_2 \Rightarrow n_1 \triangleleft n_2$, where $n_1 \prec_g n_2$ means that n_1 is a parent of n_2.[4] This notion of a schedule between nodes can then be easily extended to schedule orders for sets of nodes:

[4]The ordering $n_1 \trianglelefteq n_2$ can be read as "n_1 is scheduled no later than n_2".

$$N_1 \trianglelefteq N_2 \iff \forall n_1 \in N_1, n_2 \in N_2 \;.\; n_1 \triangleleft n_2 \tag{2.7}$$

This then gives us a way to talk—globally—about all of the events that occur during the evaluation of a program. However as we will see in a moment, when we are discussing space usage it will be useful to be able to restrict the view at times to only those nodes that are scheduled *simultaneously* using the $\bowtie$ operator:

$$n_1 \bowtie n_2 \iff n_1 \trianglelefteq n_2 \wedge n_2 \trianglelefteq n_1 \tag{2.8}$$

Now that we have defined how a schedule is represented in the computation graph, we turn to building the tools that we will need in order to measure the space that is used by a computation using our new space and scheduling representation.

In order to talk about how the schedule advances across the graph, we first partition the graph into sets of simultaneously scheduled nodes called *steps*:

$$\mathsf{steps}(\trianglelefteq) = N_1, \ldots, N_k \;\text{ such that }\; \begin{cases} \forall n \in g \;.\; \exists i . n \in N_i, \\ N_1 \triangleleft \ldots \triangleleft N_k, \\ \forall i \;.\; \forall N_i \;.\; \forall n_1, n_2 \in N_i \;.\; n_1 \bowtie n_2 \end{cases} \tag{2.9}$$

From this, we can then define the *closure* $\widehat{N_i}$ of the step N_i as

$$\widehat{N_i} = \bigcup_{k=1}^{i} N_k \tag{2.10}$$

Thus while all the nodes in $\widehat{N_i}$ are not necessarily scheduled simultaneously, we have that $\widehat{N_i} \triangleleft N_{i+1}$. With this framework, we can then think of the schedule as defining a wavefront that progresses across the graph (where each "wave" is the next step N_i). Using this, we can now easily define those new nodes in the computation that become available at each step or *ready*: we say that a node n becomes ready at step N_{i+1} if all parents of n are in $\widehat{N_i}$ but $n \notin \widehat{N_i}$. At this point we have built the machinery

that we need in order to discuss how the schedule progresses across the graph, but we are not yet able to start measuring the space-usage.

In order to start measuring the space usage of computation, we need to use the tools for describing schedules that we have just created along with how we defined the collection policy on the heap graph to start defining how the heap graph uses space. In particular, given a schedule $\trianglelefteq$ with steps $\{N_i\}_{i=1}^{k}$, if we look at a moment in time represented by N_i then since $\widehat{N_i}$ contains all previously scheduled nodes we will have that for any edge (n, ℓ) in the heap graph that this edge will fall into one of three categories:

1. $n, \ell \notin \widehat{N_i}$: since the value associated with ℓ hasn't been allocated yet, the edge (n, ℓ) does not contribute to the space usage at step N_i.
2. $n, \ell \in \widehat{N_i}$: the space for ℓ has been allocated, but the use of the value is in the past. Therefore the edge (n, ℓ) does not contribute to the space used at step N_i.
3. $\ell \in \widehat{N_i}$ and $n \notin \widehat{N_i}$: n represents a (possible) future use of the value located at ℓ, and therefore the edge (n, ℓ) contributes to the space usage at step N_i.

This partitioning of the edges between nodes into these three categories and the fact that only the edges in the third category affect the space usage lead us to a definition of a *root location* similar to what we might find in garbage collection: if $e \Downarrow v; g; h$ then the *roots* after scheduling the nodes in N with respect to a location ℓ is the set of nodes $\ell' \in N$ such that either $\ell = \ell'$, or there is an edge in h that leads from some node n outside of N to ℓ'

$$\mathsf{roots}_{h,\ell}(N) := \{\ell' \in N \mid \ell' = \ell \vee (\exists n.(n, \ell) \in h \wedge n \notin N)\} \tag{2.11}$$

With the definition of root locations behind us, we can now finally turn to putting bounds on space usage as we progress through the computation.

While the root locations in the heap graph tell us what values are immediately reachable from the current program state, we need to be able to look past just those things that are immediately reachable in order to understand the total amount of space used by the current state of the program. In order to do this, we introduce a reachability relation $\ell_1 \preceq_h \ell_2$ which says that there is a path from ℓ_1 to ℓ_2 in the heap graph. We can now say that the space usage at a step N in the heap graph of a computation that computes a value v is the number of nodes that are reachable (under $\preceq_h$) from $\mathsf{roots}_{h,v}(N_i)$

$$\mathsf{space}_{v,h}(N) = |\{\ell \in h \mid \exists \ell' \in \mathsf{roots}_{h,v}(N).\ell \preceq_h \ell\}| \tag{2.12}$$

From which we can then say that the space required by a schedule $\trianglelefteq$ that computes a value v is the maximum space usage required by any step in the schedule;

$$\mathsf{space}_{v,h}(\trianglelefteq) = \max\left\{\mathsf{space}_{v,h}(N) \mid N \in \mathsf{steps}(\trianglelefteq)\right\} \tag{2.13}$$

Now that we have defined the space usage of a schedule for a computation and heap graph, we can start to refine the definition of the permitted schedules to limit ourselves to either a more realistic class of schedules, or to particular scheduling policies. The use of graphs to represent costs is beneficial here, since this representation allows us to state the additional scheduling requirements in a simple and clear way. For example we can say that a scheduling policy $\trianglelefteq$ on p processors is *greedy* if

$$\forall n_1, n_2. n_1 \triangleleft n_2 \Rightarrow \exists n_3.(n_3 \prec_g n_2 \wedge n_1 \bowtie n_3) \vee |\{n \mid n \bowtie n_1\}| = p \tag{2.14}$$

i.e., that if n_2 is scheduled after n_1, then either there was a sequential dependency to n_2 from some other node n_3 that was scheduled simultaneously with n_1 or there were $p-1$ nodes that were scheduled simultaneously with n_1. We can similarly recreate the p-traversals from Blelloch and Greiner [21]: a p-traversal defines a strict linear order

$<_p$ on the nodes of the computation graph, we can therefore denote a p-traversal as

$$\forall n_1, n_2 . n_1 \lessdot n_2 \Rightarrow (n_1 <_p n_2) \vee \exists n_3 . (n_3 \prec_g n_2 \wedge n_1 \bowtie n_3) \vee |\{n \mid n \bowtie n_1\}| = p \quad (2.15)$$

Other schedules can be described in a similarly concise manner, and in fact the scheduling policy can be made explicit in the source language, where the constraints in Equations 2.14 and 2.15 serve as examples of the ways allowable schedules may be described. The fact that the descriptions of schedules are represented via these extra restrictions on the computation and heap graphs means that we have a purely semantic means by which we can describe the schedules permitted on the computation.

The fact that we can also determine the space usage in a purely semantic manner based upon these restrictions means that we can now profile a program based entirely upon its semantics; we now have a *purely semantic profiling framework*. This has a number of benefits, some of which are that:

1. We can profile (closed) programs written in the language in an implementation agnostic manner;

2. We can compare the semantic bounds and profiling results with those of an implementation to verify that the implementation conforms to the cost-semantic specification;

3. We can reason about the performance of scheduling policies or hardware configurations even without full implementations of these policies or access to the hardware that we are wanting to model.

For more information on how the semantic profiling framework is implemented and how this can be used to find errors and profile programs abstractly, the reader is directed to Spoonhower et al. [6].

As we have seen, there has been a great deal of work in the use of cost semantics for describing the amount of parallelism in a language, to profile for time and space, and to put time- and space-bounds on parallel programs. We have seen how different schedules can be described in the cost-models used, and the various ways in which they are encoded in the cost model for the language. One thing we have not seen however, is how this work on time- and space-profiling, possible parallelism, and space- and time-bounds in the homogeneous parallel setting relates to the distributed, and heterogeneous parallel setting. To the author's knowledge, there is no such work on cost semantics for the heterogeneous setting. One of the central questions that this dissertation seeks to answer is how to create meaningful cost semantics for heterogeneous parallel systems.

2.1.2 Cost Semantics for Cache Behavior

To the author's knowledge, there is only one work [5] on describing the effects of memory layout, bandwidth, and locality in a cost semantic setting. The interested reader is directed there for the fine details of the model and how it is related to the machine model they use. Here we will focus on giving a high-level overview of the core ideas found in that work.

One of the central issues when looking at the performance of a program is that of cache-locality and I/O efficiency; accessing memory at different levels of the memory hierarchy can have massive effects on the speed at which that data can be accessed, read, and written. In order to account for this difference in speed, the I/O [23] and ideal-cache [24] models have been created that have a two-level memory hierarchy—a fixed-sized primary memory ("the cache")—and an unbounded secondary memory ("the heap"). In this way, these models replicate the memory models and hierarchies that most programmers deal with on a day-to-day level. However, up to this point there had been no way of describing the costs of programs in these machine models.

The majority of them describing them (as we saw previously) in terms of the RAM, or PRAM setting.

Since we want to deal with the cost of the memory movement between the cache and main memory, or main memory and disk, the costs are concerned with the number of memory transfers between main memory and the cache or the *cache complexity*—specifically loading and eviction from a read-cache, and allocation and eviction (to main memory) to and from a nursery for newly allocated objects. The cost of a program then is the total number of transfers from the cache to main memory. This falls in line with what we would expect: every load is a cache-miss, and we must evict from the cache in order to load new memory into cache.

The ideal-cache memory system is parameterized by the *block size* B and the *cache size* $M = c \times B$ for some constant c.[5] The transfers between the caches are done at the block level; only blocks can be evicted and loaded into the caches. However, allocation sizes need not be bounded below by the block size: a memory μ is a finite mapping that assigns a memory object (a stack frame or a value) to either a value location ℓ or a stack location s. A *neighbor* relation $\equiv_\mu$ is then defined on these locations, such that the equivalence classes of $\equiv_\mu$ partition μ, and each equivalence class is of size B (i.e., each equivalence class defines a block in our memory). Given a memory μ and a block β such that $\mathsf{dom}(\mu) \cap \mathsf{dom}(\beta) = \emptyset$ we can *expand* μ by β, $\mu \oplus \beta$, by setting it to be the memory μ' where $\ell \equiv_{\mu'} \ell'$ iff $\ell \equiv_\mu \ell'$ or $\ell \equiv_\beta \ell'$. We similarly denote the contraction of the cache ρ by a block β (i.e., an eviction to main memory) by ρ', where $\rho' \oplus \beta = \rho$ and write $\rho' = \rho \ominus \beta$. We have a nursery (or allocation cache) ν for newly allocated objects that has a linear ordering $\prec_\nu$ between the locations called the *allocation ordering*,[6] as well as a means of finding the live locations in an allocation cache ν given a root set of locations R; $\mathsf{live}(R, \nu)$. From this, the *scan* $\mathsf{scan}(R, \nu)$ of a nursery ν with respect to a subset $R \subseteq \mathsf{dom}(\nu)$ is the block β of the oldest live locations

[5] e.g., cache-line, or page size.

[6] Which keeps track of the temporal (and hence spatial) locality of allocations into ν.

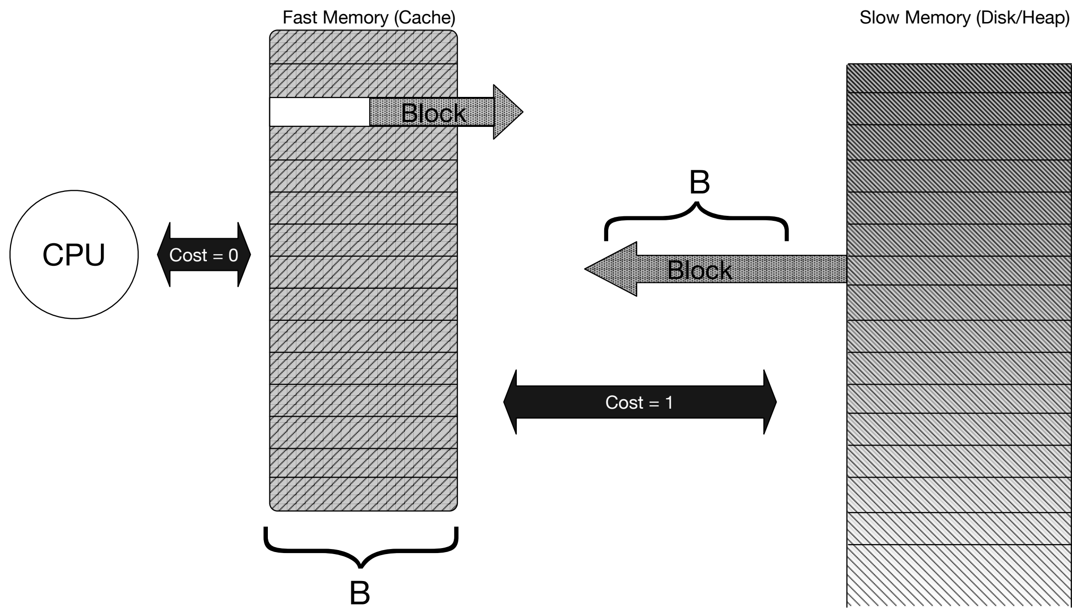


Figure 2.2: How the ideal cache model works.

in $\mathsf{live}(R, \nu)$ (we can determine this since we have a linear order $\prec_\nu$ on ν and hence on $\mathsf{live}(R, \nu)$). A store σ is then defined as a triple of the main memory μ, the read cache ρ and the allocation cache ν: $\sigma = (\mu, \nu, \rho)$. An example of how the two cache, and main memory interact can be found in Figure 2.2. For more details on the memory model, and how expansion and contraction works, the reader is referred to Morrisett, Felleisen, and Harper [25].

With the memory hierarchy and details for the ideal-cache model behind us, we now turn to how to represent the cache complexity in a cost semantics. Since we are interested in the cache complexity of programs, we need to make the allocation, eviction, and loading of blocks into the caches explicit. Moreover, to account for the memory traffic arising from the implicit control stack, the semantics must ensure

that any data that would appear in the stack is kept alive by the semantics.[7] These considerations lead to a semantics judgement form $\sigma@e \Downarrow_R^n \sigma'@\ell$. This states that the expression e when evaluated in the store σ and with root locations $R \subseteq \mathsf{dom}(\sigma)$ evaluates to a new store σ' and the value that e evaluates to can be found at the location ℓ in σ'. The semantics are that for a call-by-value typed PCF [26] language, where functions keep track of their name in order to allow for recursion:

$$e ::= x \mid \mathtt{z} \mid \mathtt{s}(e) \mid \mathtt{ifz}(e; e_0; x.e_1) \mid \mathtt{fun}(x, y.e) \mid \mathtt{app}(e_1, e_2)$$

Given this, we then need to define the cost for *reading from memory*:

$$\sigma@\ell \downarrow^n o@\sigma$$

and *allocating memory*:

$$\sigma@v \uparrow_R^n \sigma'@\ell$$

where v is a value, and o is a memory object. The read judgement states that reading a location ℓ results in a new store σ' and results in an object o with cost $n = 1$ or $n = 0$ for reading a location that is in memory, or already in the cache respectively.[8] Similarly, the allocation judgement states that allocating the value v in the store σ results in a modified store σ and location ℓ such that $\ell \in \mathsf{dom}(\sigma')$, and is charged cost $n = 1$ if the allocation causes an eviction of block from the nursery—otherwise it is charged $n = 0$ cost. With this in mind, we can present the rules for lookup and allocation that measure the movement of blocks between memory hierarchies:

[7] This once again betrays some of the intensional properties that we are describing with a mostly extensional semantics.

[8] Since this can load and/or evict cache blocks in σ.

$$\texttt{ReadIn1}\ \frac{l \in \mathsf{dom}(\rho)}{(\mu,\rho,\nu)@\ell \downarrow^{0} (\mu,\rho,\nu)@\rho(\ell)} \qquad \texttt{ReadIn2}\ \frac{l \in \mathsf{dom}(\nu)}{(\mu,\rho,\nu)@\ell \downarrow^{0} (\mu,\rho,\nu)@\nu(\ell)}$$

$$\texttt{ReadOutSpace}\ \frac{l \notin \mathsf{dom}(\rho) \cup \mathsf{dom}(\nu) \qquad |\mathsf{dom}(\rho)| \leq M - B}{(\mu,\rho,\nu)@\ell \downarrow^{1} (\mu,\rho \oplus [\mu,\ell],\nu)@\mu(\ell)}$$

$$\texttt{ReadOutEvict}\ \frac{l \notin \mathsf{dom}(\rho) \cup \mathsf{dom}(\nu) \qquad |\mathsf{dom}(\rho)| = M \qquad \beta \subseteq \rho}{(\mu,\rho,\nu)@\ell \downarrow^{1} (\mu,\rho \ominus \beta \oplus [\mu,\ell],\nu)@\mu(\ell)}$$

$$\texttt{AllocNoEvict}\ \frac{|\mathsf{live}(R \cup \mathsf{locs}(o),\nu)| < M \qquad \ell \notin \mathsf{dom}(\nu)}{(\mu,\rho,\nu)@o \uparrow^{0}_{R} (\mu,\rho,\nu[\ell \mapsto o])@\ell}$$

$$\texttt{AllocEvict}\ \frac{\mathsf{live}(R \cup \mathsf{locs}(o),\nu)| = M \qquad \beta = \mathsf{scan}(R \cup \mathsf{locs}(o),\nu) \qquad \ell \notin \mathsf{dom}(\nu)}{(\mu,\rho,\nu)@o \uparrow^{1}_{R} (\mu \oplus \beta,\rho,(\nu \ominus \beta)[\ell \mapsto o])@\ell}$$

For example, the `AllocEvict` rule states that if we wish to allocate a block for a new object o in nursery ν, and if ν is already at capacity, we find the oldest live block in ν via $\mathsf{scan}(R \cup \mathsf{locs}(o),\nu)$ and evict it to create room for the new allocation of $\ell \mapsto o$ and we charge this operation a cost of 1 since we are moving a block from the nursery to main memory. The read judgements `ReadIn1` and `ReadIn2` state that reading from cache has zero cost (since they do not have any effect on the memory and therefore have zero cache traffic). The judgements `ReadOutSpace` and `ReadOutEvict` state that reading a location ℓ from main memory will trigger a loading of the neighborhood (block) that the location is in (in main memory) into the read cache, and will therefore incur a cost of one since there is cache traffic that is induced. If the read cache is already full (the case of `ReadOutEvict`) a block must first be evicted from the read cache. Note that in this system the eviction strategy is non-deterministic, but may be made deterministic to agree with other models.[9]

One of the critical things to note at this point is the fact that the liveness of objects in the allocation cache can be determined *without accessing main memory* (and hence

[9] For more on this see § 3 of Blelloch and Harper [5].

no cache traffic is induced). This invariant is enforced by the fact that an object in the nursery cannot be live solely because of a pointer back to it from main memory (an object may only refer to objects allocated earlier in the computation, and not to objects in the future) and is a consequence of both immutability, and the explicit allocation of stack frames in the semantics.[10]

Now that the means to measure cache traffic induced in the store based on the reading and writing of locations has been defined, we can now give a cost semantics for the language that measures cache complexity:

$$\texttt{Zero}\ \frac{\sigma@\texttt{z} \uparrow^{n}_{R} \sigma'@\ell'}{\sigma@\texttt{z} \Downarrow^{n}_{R} \sigma'@\ell'}$$

$$\texttt{Succ}\ \frac{\sigma@\ \texttt{s}\ (-) \uparrow^{n}_{R\cup locs(e')} \sigma'@s' \qquad \sigma'@e' \Downarrow^{n'}_{R\cup\{s'\}} \sigma''@\ell'' \qquad \sigma''@\ \texttt{s}\ (\ell'') \uparrow^{n''}_{R} \sigma'''@\ell'''}{\sigma@\ \texttt{s}\ (e') \Downarrow^{n+n'+n''}_{R} \sigma'''@\ell'''}$$

$$\texttt{ifzz}\ \frac{\sigma@\ \texttt{ifz}\ (-; e_2, x.e_3) \uparrow^{n_1}_{R\cup locs(e_1)} \sigma_1@s_1 \qquad \sigma_1@e_1 \Downarrow^{n_1'}_{R\cup\{s_1\}} \sigma_1'@\ell_1' \qquad \sigma_1'@\ell_1' \downarrow^{n_1''} \sigma_2@\ \texttt{z} \qquad \sigma_2@e_1 \Downarrow^{n_2}_{R} \sigma'@\ell'}{\sigma@\ \texttt{ifz}\ (e_1; e_2; x.e_3) \Downarrow^{n_1+n_1'+n_1''+n_2}_{R} \sigma'@\ell'}$$

$$\texttt{fun}\ \frac{\sigma@\ \texttt{fun}\ (x, y.e) \uparrow^{n}_{R} \sigma'@\ell}{\sigma@\ \texttt{fun}\ (x, y.e) \Downarrow^{n}_{R} \sigma'@\ell}$$

$$\texttt{ifz}\ \frac{\sigma@\ \texttt{ifz}\ (-; e_2, x.e_3) \uparrow^{n_1}_{R\cup locs(e_1)} \sigma_1@s_1 \qquad \sigma_1@e_1 \Downarrow^{n_1'}_{R\cup\{s_1\}} \sigma_1'@\ell_1' \qquad \sigma_1'@\ell_1' \downarrow^{n_1''} \sigma_2@\ \texttt{s}\ (\ell_1'') \qquad \sigma_2@[\ell_1''/x]e_3 \Downarrow^{n_2}_{R} \sigma'@\ell'}{\sigma@\ \texttt{ifz}\ (e_1; e_2; x.e_3) \Downarrow^{n_1+n_1'+n_1''+n_2}_{R} \sigma'@\ell'}$$

$$\texttt{app}\ \frac{\begin{array}{c}\sigma@\ \texttt{app}\ (-; e_2) \uparrow^{n_1}_{R\cup locs(e_1)} \sigma_1@s_1 \qquad \sigma_1@e_1 \Downarrow^{n_1'}_{R\cup\{s_1\}} \sigma_1'@\ell_1' \\ \sigma_1'@\ell_1' \downarrow^{n_1''} \sigma_1''@\ \texttt{fun}\ (x, y.e) \qquad \sigma_1''@\ \texttt{app}\ (\ell_1'; -) \uparrow^{n_1'''}_{R} \sigma_2@s_2 \\ \sigma_2@e_2 \Downarrow^{n_2}_{R\cup\{s_2\}} \sigma_2'@\ell_2' \qquad \sigma_2'@[\ell_1', \ell_2'/x, y]e \Downarrow^{n_2}_{R} \sigma'@\ell'\end{array}}{\sigma@\ \texttt{app}\ (e_1; e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma'@\ell'}$$

[10] Once again, we see some of the intensionality of the semantics leaking out.

Where, for example, the rule app says that we need to evaluate the application e_1 to e_2 and determine the cache complexity in the store σ with roots R. In order to do this, we allocate a stack frame (s_1) that represents the pending evaluation of e_2 while we evaluate e_1. The frame s_1 is now "live" (i.e., all of its data must be kept alive) even though it doesn't occur in any other expressions in the judgement. We then evaluate e_1 in the store that contains the s_1 stack frame (we ensure it's treated as live by setting $R = R \cup \{s_1\}$) which returns a location ℓ'_1 which we then read to get a function. We then create another frame for the suspended application of e_1, and evaluate e_2 with s_2 as live. We then evaluate the function body, replacing the "self" variable by ℓ'_1 (so that the function body has a reference back to itself) and the arguments to the function by ℓ'_2. The overall cost of this computation is the sum of the costs of each of the component parts. The other rules can be read similarly.

If the reader is interested to see how this cost semantics is compiled down to the abstract machine, and how they implement a compiler reflecting these semantics the details can be found in Blelloch and Harper [5, §4-5].

Using this cost semantics, Blelloch and Harper [5] are able to prove tight asymptotic bounds on various well-known algorithms such as `mergeSort` ($O\left(\frac{n}{B}\log\frac{n}{M}\right)$), and `kWayMergeSort` ($O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$). They are also able to show that the common optimization of blocking matrix multiplication has the desired speedup using these cost semantics: they are able to prove that naïve matrix multiplication has complexity $\Theta(n^3)$ whereas blocked matrix multiplication has complexity $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$. For more information on the algorithms, and algorithm bounds, and how they go about proving them, the reader is referred to Blelloch and Harper [5, §6].

One of the interesting future pieces of work is the measurement of cache complexity in the presence of parallelism (and different types of parallelism). Indeed, as we will see this ties in with one of the central themes that inspired this dissertation: dealing with multiple memory hierarchies in the parallel setting. However, dealing

with cache in this manner in the parallel setting is difficult, and beyond the scope of the work in this dissertation; we refer the intrigued reader to Section 7.2 for a brief discussion on possible ways of doing this and some of the problems one may encounter in the process.

2.1.3 Cost Semantics for Self-Adjusting Computation

One of the major feats in cost semantics has been creating a cost semantics for self-adjusting computation [27] and how the costs vary based upon the update to the computation. Ley-Wild, Acar, and Fluet [4] detail how the cost semantics is developed, not on the syntax of the language solely, but based upon the syntax plus a type of edit distance from the previous trace that the computation is begin updated on.

Creating a cost semantics for self-adjusting computation creates a number of challenges primarily due to the intensionality of the properties being described. The primary challenge being: when updating a computation, how do you measure the cost of the update? This is particularly hard to do since we must then measure how the two computations differ, but cannot simply look at the store—since this is too fine-grained and will not be able to measure the full amount of computation that can be reused from run to run (e.g., $[1,3]$ and $[1,2,3]$). In order to get around this the semantics must measure the cost of updating the computation based upon traces of the store operations (`put`, `get`, `set`, `Mem`) as the program executes. The cost is then measured in terms of (essentially) the longest matching subsequence between the two traces. The cost d is then a tuple; $d = \langle c_1, c_2 \rangle$ where c_1 is the amount of work discarded from the original trace T_1 and c_2 is the new work performed for the new run and trace T_2.

In order to track the changes in the trace sufficiently, the idea of *syncing* and *searching* in the trace is introduced: a search through the trace is conducted until a place is

found where the two traces can be synced with one another. The costs are then built up from the inductive definition of these two syncing and searching "states." One thing that is especially worth noting here, is that there is non-determinism built-in to the system by the search/sync rule since this allows it to transition at any point from syncing to searching while looking through the store traces.

After the development of a way to sync and search on two computation, the idea a *failure action* that forces the changing of state from syncing to searching is introduced. This then permits synchronizing parts of the bodies of memoized functions, and then searching the remainder of one of the bodies against the tail of the other trace. This essentially allows us to blur the edge around the trace of the memoized function so that we can reuse more of the original computation.

A way of compiling down to a lower-level language with self-adjusting/self-updating computation built in is introduced, and they show how the source language can be translated down to it while preserving costs (this is done via relating traces in the source language and the target language). A memoization rule (looking up a part of the trace that represents a computation done by an expression e) is introduced and then from this change propagation, and how—since the target language store operations contain explicit continuations (and is written in CPS)—they can try and recover from inconsistent store operations from run-to-run defined.

2.1.4 Other Cost Semantic Theories

Beyond the methods discussed above there are a number of other cost-semantic theories that are more specific, but are not as central to the dissertation. We include them here for reference.

Blelloch, Gibbons, and Matias [28], Greiner and Blelloch [29], and Greiner [30] prove similar results to those of Blelloch and Greiner [21] and Spoonhower et al.

[6] except with a view towards speculative evaluation [29], and fine-grained parallelism [28]. The work of Ennals [31] is similar to the work discussed earlier about the results from Sansom and Peyton Jones [3], and Spoonhower et al. [6] wherein they develop a cost semantics using a cost-graph approach similar to that of Spoonhower et al. [6] in order to show that optimistic evaluation is faster (or at the very least, no slower) than lazy evaluation, and as a basis for their online profiling techniques.

While all of the cost semantics we have seen up to this point have been based on an operational semantics, there has been recent work on creating a denotational cost-semantic theory. In particular, Van Stone [32] recently created a denotational cost semantics by interpreting the programs in certain enriched categories, and through this is able to express costs for higher-order types, and for call-by-value and call-by-name evaluation strategies. Danner, Licata, and Ramyaa [33] extend this work to take into account inductively defined types.

2.1.5 Cost Types and Non-Semantics-Based Cost Theories

Some of the other means that have been used in order to describe the complexity of programs have been using program transformations to convert a source program into equations that compute the original program complexity [15, 34]. Resource-annotated types [1, 35] and sized-types [2, 36, 37] have also been a popular and useful means for automatically describing and studying the complexity of programs, however, they take a more static approach to the description of costs. Loidl and Hammond [38] introduce a sized type theory for an eager functional parallel language, specifically looking at being able to infer these sized types, and then using them to determine the cost of evaluating expressions and proving time and space bounds and to use this information to inform the scheduling policies that are used. One other interesting example of parallelism in this area can be found in the work of Gimenez

and Moser [39], who use the inherent parallelism found in interaction nets [40] coupled with sized types to prove complexity results in an extended λ-calculus.

Another means of determining the resource use of languages has been through the use of code instrumentation. In particular the CerCo project, worked on creating a way to determine the cost of functions (and of whole programs) via an instrumentation process (see e.g., Amadio et al. [41] and Amadio and Régis-Gianas [42]). In particular, their method involves adding labels to the source code such that all loops and function calls (along with a few other things) must go through a label, and verifying the translation process from the labeled source-code to assembly. They then analyze the number of (assembly) instructions executed by each labeled section of code, and then use this to annotate the source code with cost annotations.

2.2 Memory and Layout Theories

2.2.1 Linear Ordered Types for Memory and Data Layout

One of the central issues in reasoning about cache locality and the overhead that non-cache hits incur in higher-order and functional languages is the fact that there are often multiple valid ways of representing a source-level value in memory. For example take the expression $(1, (2, 3))$. Then there are at least three valid ways of representing this in memory, all with different cache characteristics described in Figure 2.3.

This represents a similar problem to one that we found in cost semantics: we wish to describe low-level and implementation properties at the operational (source) level in the language. In order to describe these low-level details at a high level, Petersen et al. [43] present a type theory for describing at a high-level how data structures such as tuples are laid out in memory.

Looking at the different layout strategies in Figure 2.3 we see that there are only two central properties that need to be described: adjacency and indirection. As is

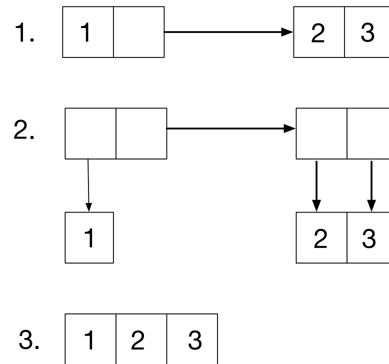


Figure 2.3: Three different ways of laying out $(1,(2,3))$.

taught in introductory algorithms classes, these two properties of the data can have a significant impact on the performance of the program. The question now becomes, how do we know when two variables in the source language that refer to memory are adjacent to each other? The answer to this lies in how memory is allocated, and associating temporal locality (i.e., things that were allocated together—things that are adjacent in the *frontier*) with spatial locality (i.e., memory cells that are adjacent to one another in the heap). This ties in with the standard memory model in which we have an allocation pointer, with the heap being to the left of the allocation pointer, and our frontier (or free-space) is to the right of the allocation pointer, and where we are only permitted to allocate prefixes of the frontier into the heap as described in Figure 2.4.

This association of temporal and spatial locality gives rise to the idea of an *ordered linear type theory* wherein the typing context Ω is not only linear, but also ordered: we can only use hypotheses (linear ordered type variables) in the order in which they were assumed. This means that not only can we enforce that we are only moving prefixes from the frontier to the heap, but we can talk about being able to fuse allocations with each other and statically guarantee these optimizations via the type system.

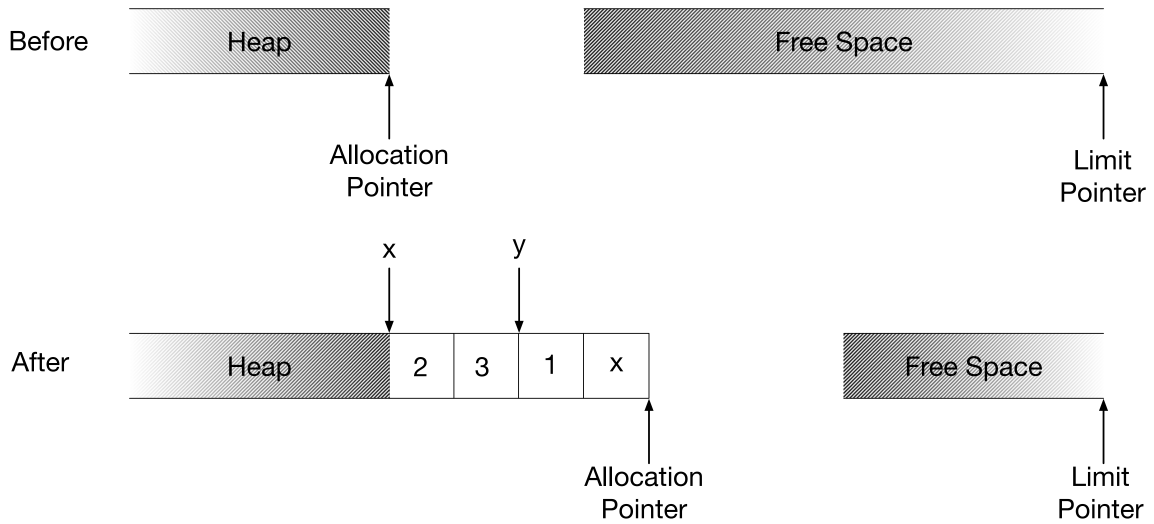


Figure 2.4: How memory is allocated for $y = (1,(2,3))$ using the first representation in Figure 2.3.

The linear ordered context that represents the allocation frontier is then accompanied by a non-ordered linear context Γ that models memory in the heap. The typing rules for the language are then phrased in terms of the interaction between these two contexts; how one goes about removing variables from the ordered context Ω and adding them to the unordered context Γ (i.e., how to allocate memory onto the heap), and how to discuss adjacency of cells in memory.

The linearity of the frontier Ω is critical to the movement of the memory from the frontier onto the heap: the fact that the types are linear ensures that we can coerce the memory to be of the same type as the memory that is expected, and that it can then be packaged up as a heap value. Using this, coupled with the ordered linearity, the types of expressions now encode not only the type and adjacency of memory, but also where that memory resides (in the frontier, or on the heap). With this, coalescing reservation (i.e., fusing allocation) then becomes easy to express in a well-typed manner. Moreover, it is easy to determine from the type of an expression whether two pointers point to adjacent locations in memory. For more information, the reader is referred to Petersen et al. [43].

2.2.2 Types for Data Layout Inference

Another impact of data layout beyond simply cache locality, is that of allowing better data parallelism. In particular, how data is laid out can have drastic effects on how programs can make use of Single Instruction Multiple Data (SIMD) operations in looping iterations. The amount of benefit from these optimizations is well known, and people go to great lengths to implement compiler optimizations such as polyhedral loop optimization [44, 45] to allow the use of SIMD operations. Šinkarovs and Scholz [46, 47] present a way to infer data layouts to better allow auto-vectorization of code based on a type system. In particular, they use arrays of types in their typing environment[11] that are then refined through the type inference process in order to infer those layouts that will allow some vectorization in the program: each function application correlates n potential layouts in the calling context with m potential layouts in the called context; likewise map and reduce operations along with other primitive operators refine the possible layouts. The possible acceptable data layouts can then be inferred through a standard type inference process. This then gives the compiler an automatic way to transform data layout in order to admit better vectorization. They prove [48] that the automatic transformations done by the compiler using the information gleaned in their type inference process preserves the correctness of the source program. One of the main areas left unanswered that they mention as future work that would make their work more practical is a a cost model for how these transformations affect the amount of parallelism available: "All that is missing is a cost model to estimate an expected performance of transformed programs." For more information on the proof and the type inference process, the reader is referred to Šinkarovs and Scholz [47].

[11] Essentially, the environment is a matrix in which each column represents a different layout combination of the data.

2.3 Region Based Memory Models

Memory management is a crucial part of any modern language: many times the total memory allocated by a program as it runs will far exceed the actual physical memory of the computer. Thus requiring some way of reclaiming memory after it is no longer used. Common techniques for this are to either have the user explicitly manage this memory (e.g. C), have a garbage collector (e.g. Java, Haskell), or have some kind of stack-discipline in which allocation and deallocation are matched up based upon the block structure of the language. While each of these methods has their own merits, they also have properties that need to be taken into account: user-managed memory is difficult to reason about and errors and memory leaks are easy to introduce, but if done properly can be the most efficient method; garbage collection frees the user entirely from having to worry about deallocating space but can cause unpredictable runtime behavior and delays; and a stack discipline places restrictions on what can be allocated, and what can be returned from function bodies while being nearly as efficient as user-managed memory.

In order to get around the shortcomings of the stack discipline approach while also preserving the efficiency aspects of it, Tofte and Talpin [8] introduced a new technique for type-safe memory management based on *regions* of memory which has many of the benefits of a *stack discipline* memory management system while simultaneously removing many of the restrictions that this system put in place. In their calculus, regions are contiguous areas of memory that hold heap-allocated data. Each piece of data that is allocated in the heap is then annotated with the region in which it resides, and the "pointer" to that object consists of the region offset along with the offset into the region where the object resides. Each of these regions are introduced with a lexically scoped construct

$$\texttt{letregion}\ \rho\ \texttt{in}\ \texttt{e}$$

and thus they have last-in-first-out (LIFO) lifetimes following the block structure of the program. In the example above, a region in memory corresponding to ρ is created upon entering the expression, and for the duration of the expression e data can be allocated into the region. After e has evaluated to a value all of the data allocated within the region is reclaimed, and the value is returned.

While the underlying memory system for Region-Based Memory Management (RBMM) is rather straightforward, ensuring that the language that uses an RBMM system is sound presents a challenge: a language in which only region creation forms are added, and nothing is done to the type system is *unsafe*; since a standard type system cannot capture the accesses to regions from the environment part in lexical closures, important properties of the computation itself are not reflected in the (types of) the values produced by those computations, and thus in this setting well-typed programs could go wrong. In order to ensure that the type system is safe, an *effect typing system* is introduced that tracks the regions that are able to be accessed during evaluation, and through this ensures the safety of this allocation and deallocation scheme. In this setting, the types of objects that are allocated on the heap are then annotated with the region where they live. For example the type

$$(\mathtt{int} \times \mathtt{int}, \rho)$$

describes a pair of integers where the pair lives in the region ρ.

While regions allow us to reason about where data is allocated, *region polymorphism* allows us to abstract over the regions that a computation manipulates. Furthermore, in the effect-typing system, function types include an effect which records the set of regions that are read or written to in the body of the function—or, the set of regions that must still be allocated (or live) for the computation to be safe. For example a function `fst` that takes in a pair of integers and returns the first component could have a type of the form

$$\mathtt{fst} : \forall \rho \ . \ (\mathtt{int} \times \mathtt{int}, \rho) \xrightarrow{\{\rho\}} \mathtt{int}$$

This function is polymorphic over the region ρ where the pair resides, so that the caller can effectively re-use the function regardless of where the data were allocated by instantiating the expression with a region ρ' e.g. `fst` $[\rho']$. However, the effect $\{\rho\}$ on the function arrow indicates that whatever region we instantiate ρ with must still be alive when `fst` is called in order for the program to be type correct.

A unique feature of the typing system of Tofte and Talpin is that it is legal for evaluation to lead to values with *dangling pointers*: a pointer to data in a region that has been reclaimed. Consider for example the following program:

```
1  letregion ρa in
2    let g = letregion ρb in
3      let p = ((1,2) at ρa, (3,4) at ρb) at ρa in
4      λ z: unit . fst[ρa] p
5    in g ()
```

The pair `p` and its first component are allocated in the outer region ρ_a, whereas `p`'s second component is allocated in the inner region ρ_b. The closure bound to `g` is a thunk that then calls `fst` on the pair `p`. However, the region ρ_b is deallocated before the thunk `g` is run, and thus `g`'s closure contains a dangling pointer to an object that is never dereferenced, and therefore does not cause an exception to be raised. The effect type system introduced by Tofte and Talpin, is strong enough to show that this code is safe.

While region polymorphism allows us to express computations that are polymorphic in the regions that they utilize, we run into issues when expressing higher-order functions such as `map`. For example, consider the following type for `map`:

$$\texttt{map} : (((\tau_1 \xrightarrow{\varphi} \tau_2, \rho_1) \times (\texttt{List}\ \tau_1, \rho_2), \rho_3) \xrightarrow{\varphi' \cup \varphi} (\texttt{List}\ \tau_2, \rho_4), \rho_5)$$

Since each function type is annotated with its set of latent effects, the type of the function that we are mapping over the vector will necessarily be different based upon the set of latent effects φ for the function being mapped. However, while the blue

section of the code is polymorphic over φ, the latent effect $\varphi' \cup \varphi$ is not included in this polymorphism, but nevertheless depends upon the regions in the effect φ. This lack of polymorphism over $\varphi' \cup \varphi$ restricts the type for `map` to functions that have latent effects φ. A different definition of `map` would then be needed for each different set of latent effects that may need to be handled. This lack of reusability of code would present a serious issue to useability of a region-based language.

In order to get around this, and allow true parametric polymorphism in RBMM-based languages, Tofte and Talpin introduce *effect polymorphism*, which allows function types to be polymorphic over their set of latent effects i.e. polymorphic over the regions that are in the effect. With effect polymorphism at our disposal, we can annotate `map` with the following type:

$$\texttt{map} : (\epsilon \;.\; (((\tau_1 \xrightarrow{\epsilon} \tau_2, \rho_1) \times (\texttt{List}\;\; \tau_1, \rho_2), \rho_3) \xrightarrow{\varphi' \cup \epsilon} (\texttt{List}\;\; \tau_2, \rho_4), \rho_5), \rho_6)$$

Where ϵ is an *effect variable*, and serves as a placeholder for a set of latent effects, that can later be instantiated, e.g. given the following effect

$$\epsilon \;.\; \{\rho_1, \rho_3\} \cup \epsilon$$

this could be instantiated with $\epsilon = \{\rho_2, \rho_5\}$ to get the following effect:

$$\{\rho_1, \rho_3, \rho_2, \rho_5\}$$

With this new type, `map` is now polymorphic in the latent effect of the function that is being mapped; the latent effect of the `map` is parameterized by the latent effect of the function being mapped, which can be instantiated at the call site.

2.3.1 Example: an effect typing system for the STLC with regions

In this section we present a simple region-based language with region and effect polymorphism and instantiation with a simple effect typing system based on the

$$
\begin{array}{lcl}
\rho & \in & \mathsf{RegionVariable} \\
\epsilon & \in & \mathsf{EffectVariable} \\
r & ::= & \bullet \mid \rho \\
\varphi & ::= & \{r_1, \ldots, r_k, \epsilon_1, \ldots, \epsilon_n\} \\
e & ::= & u \mid x \mid e\ e \mid e[\rho] \mid \texttt{letregion}\ \rho\ \texttt{in}\ e \\
u & ::= & v \mid (\lambda x.e)\ \texttt{at}\ r \mid (\lambda \rho.e)\ \texttt{at}\ r \\
v & ::= & (\lambda x.e)_r \mid (\lambda \rho.e)_r \\
\Gamma & ::= & \cdot \mid x : \tau, \Gamma \\
\tau & ::= & (\tau_1 \xrightarrow{\varphi} \tau_2, r) \mid (\Pi \rho\ .^{\varphi}\ \tau, \rho) \mid \forall \epsilon\ .\ \tau
\end{array}
$$

Figure 2.5: The syntax for a simple region-based function language adapted from Henglein, Makholm, and Niss [49] §3.5.

type system of Tofte and Talpin and adapted from Henglein, Makholm, and Niss [49]. This will be useful to us later on when we present the type system for our language.

The syntax of the language is given in Figure 2.5. The typing judgement is of the form $\Gamma \vdash e : \tau, \varphi$ which is read "in typing context Γ, e has type τ with effects φ." The typing judgements for the language are given in Figure 2.6. $\Pi\rho\ .^{\varphi}\ \tau$ binds the region variable ρ in τ and φ, and $\forall\epsilon\ .\ \tau$ binds the effect variable ϵ in τ. The set of free region variables and free effect variables of a type τ are denoted by $\mathsf{frv}\,(\tau)$ and $\mathsf{fev}\,(\tau)$ respectively, and are extended to contexts and effects in the obvious manner. Finally, the capture avoiding substitution of region r for the region variable ρ and the effect φ for effect variable ϵ are written $[\rho \mapsto r]$ and $[\epsilon \mapsto \varphi]$ respectively.

Most of the rules are fairly self-explanatory with the most interesting ones being the App and RApp rules. The former ensures that the (latent) effect of the function being applied is subsumed by the current set of live regions that have been inferred so-far during the typechecking process. Similarly for the latter RApp rule, which ensures that the bounding region φ' attached to the region abstraction is subsumed—after substitution of the region with the instantiating region—by the current effect that has been determined for the expression.

$$\text{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash e : \tau, \varphi} \qquad \text{Abs}\ \frac{\Gamma, x : \tau_1 \vdash \tau_2, \varphi_2 \qquad r \in \varphi}{\Gamma \vdash (\lambda x.e)\ \texttt{at}\ r : (\tau_1 \xrightarrow{\varphi_2} \tau_2, r), \varphi} \qquad \text{Clos}\ \frac{\Gamma, x : \tau_1 \vdash \tau_2, \varphi_2}{\Gamma \vdash (\lambda x.e)_r : (\tau_1 \xrightarrow{\varphi_2} \tau_2, r), \varphi}$$

$$\text{App}\ \frac{\begin{array}{c}\Gamma \vdash e_0 : (\tau_1 \xrightarrow{\varphi_2} \tau_2, r) \\ \Gamma \vdash e_1 : \tau_1, \varphi \qquad r \in \varphi \qquad \varphi_2 \subset \varphi\end{array}}{\Gamma \vdash e_0\ e_1 : \tau_2, \varphi} \qquad \text{RAbs}\ \frac{\Gamma \vdash u : \tau, \varphi' \qquad \rho \notin frv(\Gamma) \qquad r \in \varphi}{\Gamma \vdash (\lambda \rho\ .\ u)\ \texttt{at}\ r : (\Pi \rho\ .^{\varphi'}\ \tau, r), \varphi}$$

$$\text{RClos}\ \frac{\Gamma \vdash u : \tau, \varphi' \qquad \rho \notin frv(\Gamma)}{\Gamma \vdash (\lambda \rho\ .\ u)_r : (\Pi \rho\ .^{\varphi'}\ \tau, r), \varphi} \qquad \text{RApp}\ \frac{\begin{array}{c}\Gamma \vdash e : (\Pi \rho\ .^{\varphi'}\ \tau, r), \varphi \\ r \in \varphi \qquad [\rho \mapsto r']\varphi' \subset \varphi\end{array}}{\Gamma \vdash e[r'] : [\rho \mapsto r']\tau, \varphi}$$

$$\text{LetReg}\ \frac{\Gamma \vdash e : \tau, (\varphi, \rho) \qquad \rho \notin frv(\Gamma, \tau)}{\Gamma \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau, \varphi} \qquad \text{EPoly}\ \frac{\Gamma \vdash e : \tau, \varphi \qquad \epsilon \notin fev(\Gamma, \varphi)}{\Gamma \vdash e : \forall \epsilon\ .\ \tau, \varphi} \qquad \text{EInst}\ \frac{\Gamma \vdash e : \forall \epsilon\ .\ \tau, \varphi}{\Gamma \vdash e : [\epsilon \mapsto \varphi']\tau, \varphi}$$

Figure 2.6: The type- and effect-system for the simple region-based language adapted from Henglein, Makholm, and Niss [49] §3.5.

For further information on region-based memory management we refer the reader to the definitive work on the subject by Tofte and Talpin [8], and the overview chapter by Henglein et al. in *Advanced Topics in Programming Languages* [49].

2.4 Architectural Layout of the GPU

A Graphical Processing Unit (GPU) is a specialized circuit originally designed to accelerate computations that build and manipulate images. However, in recent years GPUs have started to be designed and utilized to accelerate many different computing applications other than simply image processing; it has become increasingly common for GPUs to be utilized as a form of stream or vector processor that run computation kernels. In applications that require large numbers of vector operations,

and many independent tasks, using a GPU can increase performance by several orders of magnitude, and in many fields such as machine learning [50], medical science [51, 52], finance [53], graphics [54], and computer vision [55] GPUs have become a de-facto requirement.

The GPU circuit is designed to be a Single Instruction Multiple Data (SIMD) architecture in which a single program operates in a massively parallel manner over different pieces of data.[12] However, at the same time, the GPU as a whole acts in a Single Processor Multiple Data or SPMD manner: a GPU consists of several processing units or Streaming Multiprocessors (SMPs) each tracking many different thread contexts at once, and that operate in a SIMD fashion; however each SMP does not need to be executing the same instructions as the other SMPs. Thus, we can think of SMPs providing task parallelism, while the threads within each SMP provide (internal) data parallelism to the computations. This internal data-parallelism within SMPs gives the illusion that the GPU consists of thousands of threads—however compared to CPU threads these GPU threads are rather simple; GPUs make little or no use of speculation or out-of-order execution, and instead hide latency by switching between threads as their dependencies are satisfied. Because of these computational characteristics, while GPUs can perform any computation that a CPU can, GPUs are not as well suited for certain types of parallel computation as much as CPUs are. However as was mentioned before, when the parallel workload does fit that of a GPU the speedup can be truly impressive due to the number of threads—and the data parallelism that they provide—compared to the CPU and the computational throughput that this provides.

The threads of the GPU are grouped into sets of threads called *thread blocks*, and every thread within a thread block runs on the same SMP.[13] This grouping of threads

[12] The GPU is also a Single Instruction Multiple Thread (SIMT) architecture.

[13] Furthermore, each set of threads within a thread block is further broken up into into sets of threads called *warps* which correspond to the number of threads that will executed for any one “step” of a SMP.

is important due to the way that memory is structured on the GPU: most memory falls into global memory which every SMP can see, and which resides in an off-chip Dynamic Random Access Memory (DRAM). However, the GPU also provides a small limited amount of very fast local memory for each SMP which can be thought of as similar to L2 cache on the CPU, and while changes to local SMP memory are only visible within the thread block, changes to global memory can be seen by any CUDA (or OpenCL) thread.[14] Thus, threads within the same thread block can communicate with each other much faster using this shared memory and atomic operations (amongst other things).

Furthermore, local SMP or shared memory is organized into banks of memory, and each bank can only service one memory request at a time. In order to try and increase the memory bandwidth, memory addresses are interleaved, and on modern GPUs this interleaving is usually 32 bits, and the total number of memory banks is fixed. This division of local memory into equally-sized banks of memory means that any load or store of n addresses in local memory that spans all n banks of memory can be performed simultaneously—thus we can get up to n times the bandwidth of a single memory bank. However, if a memory request has multiple addresses within the same memory bank, the accesses must be serialized. Furthermore, the GPU will split the memory request into as few conflict-free address sets as possible, and therefore a bank conflict will reduce the bandwidth of local memory by the number of conflicting addresses.

2.4.1 Low-Level GPU Programming Models

One of the most common ways to express computation on GPUs is using CUDA, or the Compute Unified Device Architecture [56], which is the computational engine for NVIDIA GPUs. With CUDA, programmers are able to access virtual instruction

[14] In fact, local SMP memory is approximately $100X$ faster.

sets and memory for the parallel computational elements on NVIDIA GPUs. There are also other ways of expressing GPU computations, such as with OpenCL [57] that presents similar low-level constructs as CUDA, but using different terminology.

Each computation on the GPU is expressed through a computational *kernel* in which the programmer writes a single program that will run on *each* GPU thread. This is as opposed to normal programs that explicitly manage threads within the function. Due to each kernel being assigned to a thread, within each kernel has access to its unique thread ID usually through a built-in variable (`threadIdx` in the case of CUDA). To see an example of this, compare the two definitions of vector addition in Figure 2.7. Kernels are normally launched from the CPU or *host* and then run on the GPU, however in recent years launching kernels from within kernels, or *dynamic parallelism* has also become possible [58, 59]. When launching a kernel, the number of threads that we use, along with how those threads are partitioned into work groups can have a significant impact on the overall runtime of the kernel. Thus, when launching a kernel we also specify both the number of work groups, along with the number of threads in each work group (which is a multiple of 32). Thus in CUDA, we could launch the `vecAdd` kernel with $k \geq n$ threads, and 1 work group over vectors `A` and `B` and returning the result in `C` with the following

```
// Launch a kernel with 1 work group and k threads
vecAdd<<<1, k>>>(A,B,C, n);
// 8 work groups with ceiling of (k/8) threads each
vecAdd<<<8, (k-1)/8 + 1>>>(A,B,C, n);
```

2.4.2 Moving Memory Between Worlds

In order to run this kernel the GPU needs to be able to access the memory for these vectors. However, while the CPU and GPU can share the same address space using

```
__global__
void vecAdd(int* A, int* B, int* C, int n) {
 // Get the thread index
 int idx = threadIdx.x;
 if (idx < n) {
  C[idx] = A[idx] + B[idx];
 }
}
```

(a) Vector addition in CUDA of two vectors `A` and `B` both of length n, storing the result in `C`.

```
__global__
void vecAdd(int* A, int* B, int* C, int n) {
 // Get the thread index
 int idx = threadIdx.x;
 int stride = blockDim.x;
 for (int i = idx; i < n; i += stride) {
  C[i] = A[i] + B[i];
 }
}
```

(b) Vector addition in CUDA using blocking.

```
void vecAddMP(int* A, int* B, int* C, int n) {
  #pragma omp parallel
  #pragma omp for
  for (int ids x = 0; i < n; i++) {
   C[idx] = A[idx] + B[idx];
  }
}
```

(c) Parallel vector addition in OpenMP of two vectors `A` and `B` both of length n, storing the result in `C`.

Figure 2.7: Parallel vector addition in CUDA C (a) & (b), and in C using OpenMP (c). Note that in (a) we access the array using the thread index `threadIdx` and thus this code runs on each thread, whereas in in (b) we access it based upon the `blockDim` and therefore multiple results will be computed by the same thread.

Unified Virtual Addressing (UVA), doing so can be incredibly expensive: the memory for the CPU and GPU is not shared and the two together constitute a very simple distributed system. Thus how we migrate memory back and forth between the CPU and the GPU is important. Indeed, if we wanted to have a single address space and used UVA, this would require that we first pin memory [60] on the host, and the GPU would then need to access memory over the PCI-Express bus, which has low bandwidth and high latency. Thus this is not really an option for general purpose GPU computation.

In order to get around this problem there are two options available for managing memory that needs to be shared with the GPU.

Option 1: Explicit Copying Using this approach each piece of memory from one computational world that is needed in the other must be explicitly copied and, after the GPU computation has completed, transferred back to the host. Thus we no longer have a single address space, but at least two (one for the host, and one for each GPU). In this case, the CPU and GPU represent a distributed system in the fullest sense, and therefore any data structures that are transferred between the CPU and GPU (and back) must have deep copies performed on them.

Using this method, the kernel call above would look like the following:

```
size_t size = n * sizeof(int);
int* A = malloc(size);
...
int *d_A, *d_B, *d_C;
// Allocate space on the GPU for the vectors
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);
// Copy the input vectors over
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
vecAdd<<<1, k>>>(d_A,d_B,d_C, n);
```

```
// Wait for the kernel to complete
cudaDeviceSynchronize();
// Copy the result memory back to the host
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
// Free the memory on the GPU
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
...
free(A);
```

Option 2: Universal Managed Memory Recent GPUs and runtime systems now support either Unified Managed Memory (UMM) in the case of NVIDIA GPUs [61], or heterogeneous Unified Memory Access (hUMA) for AMD. Both of these provide a single pool of memory that is shared between the CPU and the GPU. Most importantly, we get the advantage of a single address space, and thus always have a *single* pointer to data. This then means that when moving data from one computational world to another, we no longer have to perform deep copies of the data structures that are being utilized on the GPU: where previously a full retracing of pointers would have to be done, a shallow copy will now suffice. However, as opposed to UVA, in which a single address space is achieved by accessing host memory over the PCI-E, with UMM (hUMA) we get a single address space, and the underlying system also properly migrates memory from the host to device and back again—thus being vastly more efficient than UVA.

Using these unified memory systems, memory can only be allocated in host code, but both host and device code can read and write to the memory.[15] Furthermore, unified memory that is to be shared between host and device must be allocated and deallocated by special functions—in the case of CUDA, using `cudaMallocManaged`,

[15]As we will see later this restriction will be reflected in the language that we develop.

and `cudaFree` respectively. Using this framework the above kernel call could instead be written as the following:

```
size_t size = n * sizeof(int);
int* A;
cudaMallocManaged(&A, size);
...
vecAdd<<<1, k>>>(A,B,C, n);
// Wait for the kernel to complete
cudaDeviceSynchronize();
...
cudaFree(A);
```

While unified memory allows the programmer to not worry about how, or when the memory is migrated, it may be useful to control when the memory is moved. If no hints are provided pages of memory are usually migrated based upon page faults (although other heuristics are also used to minimize the number of these as well). Beyond this, where the data should be allocated can be guided via functions such as `cudaMemAdvise()` or explicitly migrated to the device via other methods such as `cudaMemPrefetchAsync()`. Thus in this setting we can both have a single unified address space, while also having control over the memory movement between the computational worlds. If we wanted to prefetch the data used by the `vecAdd` kernel above, we could do so by inserting calls to `cudaMemPrefetchAsync()` before the kernel call. Note however, that we don't have to explicitly "re-fetch" the data back from the GPU—the written data is migrated back at the end of the kernel.

2.4.3 Warp Divergence

Due to the SIMD nature of the SMPs that constitute the GPU, we encounter an issue when dealing with `if` or any type of branching statements on the GPU: since threads in a warp execute in a SIMD manner, each warp-step of an SMP can only handle threads *performing the same action* on multiple pieces of data. However when a

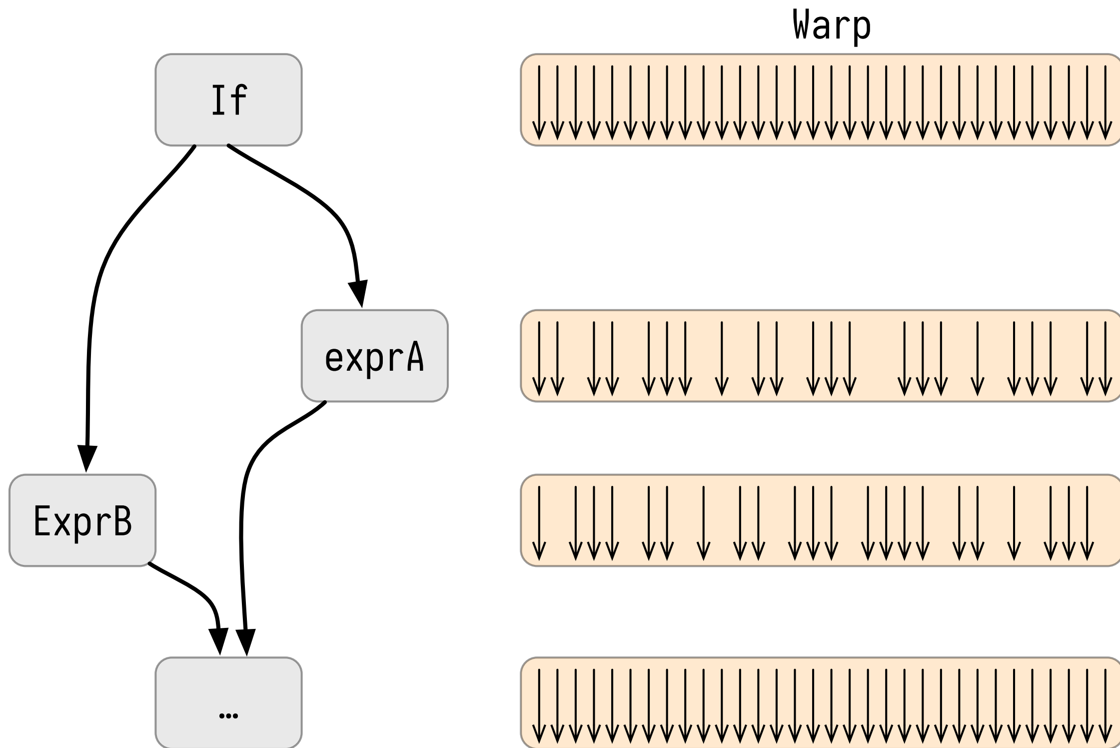


Figure 2.8: Warp divergence due to branching instructions.

warp encounters a branching statement some of the threads in it may need to compute the consequent branch of the `if`, and others may need to compute the alternate branch. Now some of the threads in the warp need to execute one computation, while others need to execute a different computation. This is what is called *warp divergence*.

The GPU handles this divergence of execution paths within the warp by serializing the different execution paths present within that warp. Thus it first runs one set of execution paths within the warp, and then runs another set of execution paths, and then the next etc. This serialization of the execution of the threads within the warp can lead to significant decreases in efficiency of the computation: what used to be one single step of the warp can become up to 32 (or the size of the warp) steps.

2.4.4 RBMM for High-Level Heterogeneous Languages

Besides the lower-level languages that were discussed previously, there are also a number of higher-level functional languages that also allow GPU parallelism such as Harlan [7, 9], Copperhead [62], and Accelerate [63, 64] amongst others. However the most interesting one of these to us in this dissertation will be Harlan—and specifically, its use of a region-based memory management system to allow for high-level constructs such as Algebraic Data Types (ADTs) and higher-order functions to be used across both host and device sections of code.

The latter two of these languages—Copperhead and Accelerate—constrain users to largely rectangular arrays since these can easily be mapped down to the underlying GPU. However, many problems do not fit nicely into rectangular arrays, and other data structures may be desired. To combat this, Harlan uses a RBMM system to provide a uniform memory representation on both the CPU and GPU and a means for easily transferring whole data structures from one device to another—by transferring the appropriate regions of memory. In this RBMM system, when a kernel is invoked a copy of each region that contains the data structures used within that kernel rather than trying to trace each individual element and pointer—much akin to what must be done by a tracing garbage collector.

Just as in Harlan, we will be using a region-based system for the language and cost semantics that we build up in this dissertation for a heterogeneous system with the view of the CPU/GPU being a specific type of distributed system. Thus this research can be viewed as being dually applicable to both the CPU/GPU system viewed as a special case of a distributed system, and as developing a cost semantics for a non-distributed memory region-based language, that can be extended to a distributed setting later on.

3

Cost Semantics for Region-Based Parallel Languages

Contents

3.1 Introduction **55**
3.2 Syntax **56**
3.3 An Effect Type System for Λ_C **58**
3.3.1 Region Annotations & Polymorphism 60
3.3.2 Latent Effects 61
3.3.3 Effect Polymorphism 63
3.3.4 The Type System 65
3.3.5 Typing Heap Values 73
3.4 Memory Model **76**
3.4.1 Region Based Memory Management 76
3.5 The Abstract Machine Model **79**
3.6 Cost Semantics for Λ_C **82**
3.6.1 Cost Graphs 82
3.6.2 Bounded Evolutions of the Region Stack 85
3.6.3 Semantic Rules 87
3.6.4 Schedules 92
3.6.5 Determining Costs 97

3.1 Introduction

As we have seen in Chapter 2 there has been an extensive amount of work on creating cost-semantic methods for determining time- and space-usage for garbage collected parallel languages. However, as we also saw in Section 2.4.4 if we wish to permit higher-order constructs to reside on the GPU, we will instead want to use a region-based memory management system instead of a normal garbage collection strategy in order to allow easy determination of which portions, and the amount of memory that needs to be transferred to and from the GPU.

This need for an RBMM-based language gives rise to the work in this chapter. Thus, we first develop a region-based memory system for a simple parallel language Λ_{C}. We then develop a cost semantics for this language that can determine time- and space-usage—including the cost of allocating and deallocating regions. We then present the final results of this chapter in Section 3.6.5; a cost semantics and definition of scheduling policies for Λ_{C} that accounts for non-unit costs due to region allocations and deallocations. We conclude the chapter by presenting the key definitions for time and space usage of a program in the cost semantics as functions of the chosen scheduling policy in Equations (3.13) and (3.14).

Viewed as part of the larger aim of this dissertation, this chapter develops the CPU portion of the language and cost-semantic system for the final heterogeneous parallel language. This larger view towards this chapter is important since at points we will have to impose certain restrictions on the language as well as change the development of the type- and region-system in such a way that the GPU-based part of the language can be easily integrated.

3.2 Syntax

The core definition of Λ_C is given in Figure 3.1, and is an extended form of the language presented by Fluet and Morrisett [65]. It consists of a standard λ-calculus along with pairs (e_1, e_2) `at` ρ, parallel composition $e_1 \| e_2$ `at` ρ, arrays $[e_1, \ldots, e_n]$ `at` ρ, and parallel traversals of these arrays `parmap` f $[e_1, \ldots, e_n]$ `at` ρ. Each λ-abstraction is annotated with the set of regions that it uses, and the closure for it must be allocated in the specified region ρ—which is why closures do not contain a region annotation. Furthermore, while the language is particularly sparse for a region-based parallel language since we wish it to be a usable intermediate language for a compiler, we have also designed it to be suitably expressive so as to allow us to fairly easily and directly express interesting parallel computations within it.

The values of Λ_C consist of integers i, booleans b, and heap values that can be stored in regions of memory R. Heap values consist of arrays of values $[v_1, \ldots, v_n]$, function closures `clos` $(x : \tau, e, E)$ consisting of the parameter $x : \tau$, body e and environment E at the time the closure was created, locations in memory ℓ, and finally region closures `rclos` $(\rho, \varphi, \varphi', u)$ consisting of the region parameter ρ, body u and the upper- and lower bounding effects φ and φ' for u respectively. Locations $\ell = (r, o)$ consist of a region name r along with the offset o into that region. Region closures `rclos` $(\rho, \varphi, \varphi', u)$ represent region-polymorphic code and are created via the region abstraction form $\lambda\rho \succeq \varphi.^{\varphi'} u$ `at` ρ, these are then instantiated in much the same way as we would a function closure via the region instantiation form $e[\rho]$ (and we will see precisely how this works in the operational semantics in Section 3.6).

It is important to note that while the definition of Λ_C follows quite closely that of a standard call-by-value typed λ-calculus with extensions for regions, allocation within regions, and parallel computation, Λ_C *does not* have any way of expressing concurrency. This lack of concurrent operations in the language is by design; the

$$i \in \mathbb{Z} \mid r \in \mathsf{RNames} \mid f, x \in \mathsf{Vars} \mid \zeta, \rho \in \mathsf{RVars} \mid o \in \mathbb{N}$$

$$
\begin{array}{lcll}
\ell & ::= & (r, o) & \text{Region Locations} \\
R & ::= & (\mathsf{ap}, s, \mathsf{refcnt}, \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\}) & \text{Regions} \\
\mathcal{S} & ::= & \underline{\rho} \mapsto \underline{R} \mid \mathcal{S}, r \mapsto R & \text{Region Stack} \\
\varphi & ::= & \{\rho_1, \ldots, \rho_n\} & \text{Effect Sets} \\
\tau & ::= & \texttt{bool} \mid \texttt{int} \mid (\mu, \rho) & \text{Types} \\
\mu & ::= & \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \texttt{Vec}\ \tau \mid \Pi \rho \geq \varphi.^{\varphi'} \tau & \text{Boxed Types} \\
\Omega & ::= & \{\underline{\rho}\} \mid \rho, \Omega & \text{World Contexts} \\
\Delta & ::= & \cdot \mid \Delta, \rho \geq \varphi & \text{Region Constraints} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau & \text{Typing Contexts} \\
e & ::= & x \mid i \mid \texttt{True} \mid \texttt{False} \mid e_1 < e_2 \mid e_1 * e_2 & \text{Terms} \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f & \\
 & \mid & \lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho \mid e_1\ e_2 & \\
 & \mid & \lambda \zeta \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho \mid e[\rho] & \\
 & \mid & (e_1, e_2)\ \texttt{at}\ \rho \mid \texttt{fst}\ e \mid \texttt{snd}\ e & \\
 & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e \mid \texttt{fix}\ f : \tau.u \mid e_1 \| e_2\ \texttt{at}\ \rho & \\
 & \mid & [e_1, \ldots, e_n]\ \texttt{at}\ \rho \mid \texttt{parmap}\ e\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho & \\
u & ::= & \lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho \mid \lambda \zeta \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho & \text{Abstractions} \\
v & ::= & i \mid b \mid (v_1, v_2) \mid [v_1, \ldots, v_n] & \text{Values} \\
 & \mid & \texttt{clos}\ (x : \tau, e, E) \mid \texttt{rclos}\ (\rho, \varphi, \varphi', u) \mid \ell &
\end{array}
$$

Figure 3.1: Syntax of Λ_{C}

introduction of concurrent operations would not only make proving region-safety guarantees about the language more difficult, but—as we will see later—would make reasoning about the cost characteristics of GPU device sections in the program nearly impossible when we introduce them in Chapter 5. We therefore leave the introduction of concurrency to the language for future investigation.

During the formal development of Λ_{C} we will find a number of metafunctions over the syntax and regions of the language useful: we define $\mathsf{frv}\,(e)$ and $\mathsf{fv}\,(e)$ to be the set of all free region variables and free variables occurring in the expression e respectively. Further, we define a metafunction $|v|$ that given a heap value v calculates the size of v. The definitions of both $\mathsf{frv}\,(e)$ and $\mathsf{fv}\,(e)$ are straightforward, with the

`letregion` and region abstraction forms serving as binders for region variables in $\mathsf{frv}\,(e)$. The definitions of these metafunctions are given in Figures 3.2 and 3.3.

3.3 An Effect Type System for Λ_{C}

In Λ_{C} , all memory exists as part of some region of which there are two types:

- A *single* (restricted in size) heap region which exists for the entirety of the program. We denote this special region by $\underline{\rho}$, and this maps to the starting region $\underline{R}$ in our region stack.
- Dynamic regions, which have lexically scoped lifetimes, but permit unlimited allocation into them.

Static values such as booleans are allocated into the heap region, and all other—dynamically-allocated—values are allocated in dynamic regions. We can see this difference in allocation characteristics for static values in the rules for booleans in Λ_{C}: booleans are heap allocated (and hence reside in $\underline{\rho}$) and are therefore not annotated with a region. However, all other types are dynamically allocated, and therefore must be annotated with the region where they reside. For the rest of this dissertation when we say "region" we will mean a dynamically-allocated region unless otherwise stated.

Each region variable in the program is assumed to be unique, and each region resides in a region stack $\mathcal{S}$.[1] Regions are lexically scoped, and are introduced via the `letregion` form in the language, and region variables can be bound via the region abstraction form $\lambda\zeta \geq \varphi'.^{\varphi}u$ `at` ρ. However, while $\lambda\zeta$ . . . . can *bind* region variables it cannot *create* new regions in memory, thus the memory for a region variable is deallocated precisely when execution leaves the scope of the `letregion` form that

[1]Assumption of uniqueness of region names does not limit the language at all, since this property can be easily ensured by performing α-conversion on any offending program.

$$\boxed{\mathsf{fv}\,(e)}$$

$$
\begin{array}{lcl}
\mathsf{fv}\,(x) & = & \{x\} \\
\mathsf{fv}\,(i \texttt{ at } \rho) & = & \{\} \\
\mathsf{fv}\,(\,\texttt{True}\,) & = & \{\} \\
\mathsf{fv}\,(\,\texttt{False}\,) & = & \{\} \\
\mathsf{fv}\,(\,\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \mathsf{fv}\,(e_1) \cup \mathsf{fv}\,(e_2) \cup \mathsf{fv}\,(e_3) \\
\mathsf{fv}\,(\lambda x : \tau.^{\varphi}\, e \texttt{ at } \rho) & = & \mathsf{fv}\,(e) \setminus \{x\} \\
\mathsf{fv}\,(e_1\ e_2) & = & \mathsf{fv}\,(e_1) \cup \mathsf{fv}\,(e_2) \\
\mathsf{fv}\,(\lambda \rho \geq \varphi'.^{\varphi}\, u \texttt{ at } \rho) & = & \mathsf{fv}\,(u) \\
\mathsf{fv}\,(e[\rho]) & = & \mathsf{fv}\,(e) \\
\mathsf{fv}\,((e_1, e_2) \texttt{ at } \rho) & = & \mathsf{fv}\,(e_1) \cup \mathsf{fv}\,(e_2) \\
\mathsf{fv}\,(\,\texttt{fst } e) & = & \mathsf{fv}\,(e) \\
\mathsf{fv}\,(\,\texttt{snd } e) & = & \mathsf{fv}\,(e) \\
\mathsf{fv}\,(\,\texttt{letregion } \rho \texttt{ in } e) & = & \mathsf{fv}\,(e) \\
\mathsf{fv}\,(\,\texttt{fix } f : \tau.u) & = & \mathsf{fv}\,(u) \setminus \{f\} \\
\mathsf{fv}\,(e_1 \| e_2 \texttt{ at } \rho) & = & \mathsf{fv}\,(e_1) \cup \mathsf{fv}\,(e_2) \\
\mathsf{fv}\,([e_1, \ldots, e_n] \texttt{ at } \rho) & = & \bigcup_{i=1}^{n} \mathsf{fv}\,(e_i) \\
\mathsf{fv}\,(\,\texttt{parmap } e\,[e_1, \ldots, e_n] \texttt{ at } \rho) & = & \mathsf{fv}\,(e) \cup \left(\bigcup_{i=1}^{n} \mathsf{fv}\,(e_i) \right)
\end{array}
$$

$$\boxed{\mathsf{frv}\,(e)}$$

$$
\begin{array}{lcl}
\mathsf{frv}\,(x) & = & \{\} \\
\mathsf{frv}\,(i \texttt{ at } \rho) & = & \{\rho\} \\
\mathsf{frv}\,(\,\texttt{True}\,) & = & \{\} \\
\mathsf{frv}\,(\,\texttt{False}\,) & = & \{\} \\
\mathsf{frv}\,(\,\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \mathsf{frv}\,(e_1) \cup \mathsf{frv}\,(e_2) \cup \mathsf{frv}\,(e_3) \\
\mathsf{frv}\,(\lambda x : \tau.^{\varphi}\, e \texttt{ at } \rho) & = & \mathsf{frv}\,(e) \cup \{\rho\} \\
\mathsf{frv}\,(e_1\ e_2) & = & \mathsf{frv}\,(e_1) \cup \mathsf{frv}\,(e_2) \cup \{\rho\} \\
\mathsf{frv}\,(\lambda \zeta \geq \varphi'.^{\varphi}\, u \texttt{ at } \rho) & = & (\mathsf{frv}\,(u) \cup \{\rho\}) \setminus \{\zeta\} \\
\mathsf{frv}\,(e[\rho]) & = & \mathsf{frv}\,(e) \cup \{\rho\} \\
\mathsf{frv}\,((e_1, e_2) \texttt{ at } \rho) & = & \mathsf{frv}\,(e_1) \cup \mathsf{frv}\,(e_2) \cup \{\rho\} \\
\mathsf{frv}\,(\,\texttt{fst } e) & = & \mathsf{frv}\,(e) \\
\mathsf{frv}\,(\,\texttt{snd } e) & = & \mathsf{frv}\,(e) \\
\mathsf{frv}\,(\,\texttt{letregion } \rho \texttt{ in } e) & = & \mathsf{frv}\,(e) \setminus \{\rho\} \\
\mathsf{frv}\,(\,\texttt{fix } f : \tau.u) & = & \mathsf{frv}\,(u) \\
\mathsf{frv}\,(e_1 \| e_2 \texttt{ at } \rho) & = & \mathsf{frv}\,(e_1) \cup \mathsf{frv}\,(e_2) \cup \{\rho\} \\
\mathsf{frv}\,([e_1, \ldots, e_n] \texttt{ at } \rho) & = & \left(\bigcup_{i=1}^{n} \mathsf{frv}\,(e_i) \right) \cup \{\rho\} \\
\mathsf{frv}\,(\,\texttt{parmap } e\,[e_1, \ldots, e_n] \texttt{ at } \rho) & = & \mathsf{frv}\,(e) \cup \left(\bigcup_{i=1}^{n} \mathsf{frv}\,(e_i) \right) \cup \{\rho\}
\end{array}
$$

Figure 3.2: Metafunctions over the syntax of Λ_{C}.

$$
\boxed{|v|}
$$

$$
\begin{array}{lcl}
|i| & = & 1 \\
|b| & = & 1 \\
|(v_1, v_2)| & = & |v_1| + |v_2| + 1 \\
|[i_1, \ldots, i_n]| & = & n + 1 \\
|[b_1, \ldots, b_n]| & = & n + 1 \\
|[v_1, \ldots, v_n]| & = & \left(\sum_{i=1}^{n} |v_i|\right) + n + 1 \\
|\,\texttt{clos}\ (x : \tau, e, E)| & = & 1 + |E| + 1 \\
|\,\texttt{rclos}\ (\rho, \varphi, \varphi', u)| & = & 2
\end{array}
$$

Figure 3.3: Size metafunction over the values in Λ_{C}. Note that boolean and integer arrays are stored unboxed.

bound it, with no deallocation occurring when a region variable bound by a region abstraction form goes out of scope.

3.3.1 Region Annotations & Polymorphism

Other than booleans and integers, all other values in Λ_{C} are boxed, and all pointers point into one, and only one region. We therefore annotate each boxed type with the region that the pointer points to (i.e. where the actual value is allocated); thus types τ are either `bool`, `int`, or a boxed type μ coupled with a region ρ. In order to allow allocation into different regions, Λ_{C} has *region polymorphism*; region abstraction $\lambda\zeta \geq \varphi.^{\varphi'} u \texttt{ at } \rho$ introduces a region abstraction that is allocated at ρ, and allows the abstraction u (with latent effects φ') to be polymorphic over the region ζ. However, we must be careful about the amount of polymorphism allowed through this form; we need to ensure that whatever region instantiates $\lambda\zeta \geq \varphi.^{\varphi'} u \texttt{ at } \rho$ outlives the given set of regions φ that are required by u.[2] Region polymorphism is an important property of Λ_{C} and allows programs to manipulate boxed values regardless of which region they may have been allocated in, as well as allowing polymorphic recursion. Furthermore, since regions adhere to a lexical structure, we can guarantee equality

[2] In a sense φ acts as a lower bound on the regions that the instantiating region must outlive.

of (unknown polymorphic) regions via comparison of the region variable names in the program.

We also include a term for fixed points `fix` $f : \tau.u$. Since Λ_C is call-by-value, the body of the `fix` operator is restricted to an abstraction u. To see an example of how polymorphic (region) recursion can be performed using the `fix` form and region abstraction/instantiation forms, consider the following definition of the Fibonacci function which uses a pair of integers for tabulation

```
1  fix fib.
2    λζ ≥ {ρ, ρp}.{} .
3     λ i : int .
4      λ p : (int × int , ρ) .
5       if i ≤ 1
6       then snd p
7       else
8        letregion ρ′
9        in (fib[ρ′] (i - 1)) (snd p, fst p + snd p) at ρ′
10     at ρ
11    at ζ
12   at ρp
```

where the the original pair is allocated in ρ_p. Region-polymorphic recursion is then performed by instantiating the region abstraction within the `fix` on line 2 with a new region at the recursive call site on line 9.

3.3.2 Latent Effects

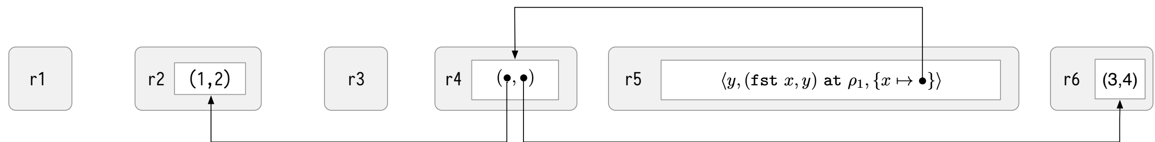
While regions in Λ_C follow lexically-based lifetimes, values allocated in a region ρ do not necessarily have to follow this scoping structure and can escape the scope of ρ; values allocated within a function closure can escape the scoping of the `letregion` form that they are allocated under. In order to solve this issue of values being able to escape the life of their backing region, each function type $\tau_1 \rightarrow \tau_2$ is annotated with a set of *latent effects* φ that describe the regions of memory that are utilized by

```
1 letregion ρ4
2 in letregion ρ5
3    in letregion ρ6
4       in (λ x.  (λ y. (fst x, y) at ρ1) at ρ5)
5          (((1,2) at ρ2, (3,4) at ρ6) at ρ4)
6    (5,6) at ρ3
```

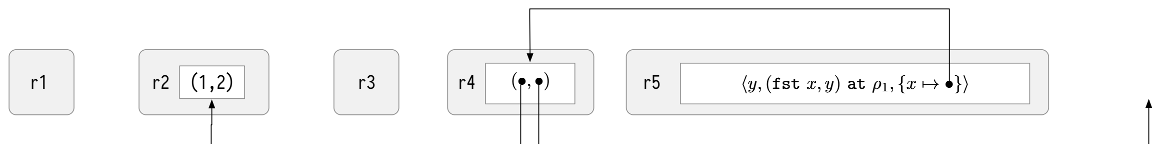
Figure 3.4: Example adapted from Tofte and Talpin [8] showing how dangling pointers can be created in a RBMM system. ρ_1, ρ_2, and ρ_3 are allocated outside this section of code. Types are elided for brevity.

the function. Thus function types are of the form $f : \tau_1 \xrightarrow{\varphi} \tau_2$ and can be read as "f is a function from τ_1 to τ_2 with effects φ."

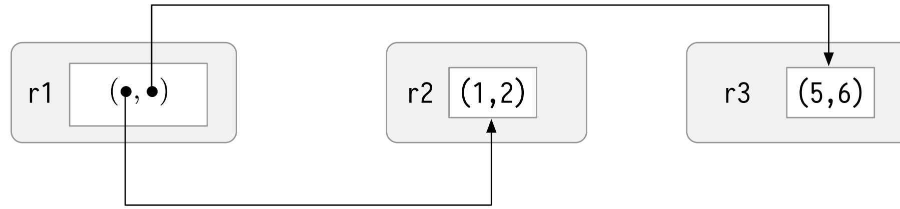
The latent effects for a function type denotes the set of current regions that are accessed by the body of the function—both for reading and writing. By typing functions with their latent effects we are lifting a coarse model of the memory access characteristics of the function to the type-level. This can then be used to statically ensure that the regions of memory that are accessed within a given function body are valid wherever that function may be applied—whilst allowing such things as dangling pointers to memory as long as it can be shown (via the type system) that the memory is not accessed [8]. An example of this can be seen in the code sample in Figure 3.4 which gives us the following three snapshots of the region stack during its evaluation; the first at line 5 just after the inner closure has been allocated:



the second at line 6 just before the closure is applied:



and finally the last one after the entire expression has run:



We want the code above to be type-correct, since we can show that the dangling pointer can never be accessed. This is precisely what the latent effects in the effect type system are meant to capture.

3.3.3 Effect Polymorphism

While we can express region-polymorphic code in $\Lambda_{\mathtt{C}}$ we run into issues when trying to express higher-order functions such as `map`. To see this, consider the following type for `map`:

$$\mathtt{map} : (((\tau_1 \xrightarrow{\varphi} \tau_2, \zeta_1) \times (\mathtt{Vec}\ \tau_1, \zeta_2), \zeta_3) \xrightarrow{\varphi' \cup \varphi} (\mathtt{Vec}\ \tau_2, \zeta_4), \zeta_5)$$

Since each function type is annotated with its set of latent effects, the type of the function being mapped over the vector will necessarily be different based upon the set of latent effects for the function being mapped. However, while the blue section of the code is polymorphic over φ, the latent effect $\varphi' \cup \varphi$ is not included in this polymorphism, but nevertheless depends upon what φ is. This lack of polymorphism over $\varphi' \cup \varphi$ restricts the type for `map` to functions that have latent effects φ. A different definition of `map` would then be needed for each different set of latent effects that may need to be handled. This lack of reusability of code would present a serious issue to useability of the language.

As we saw in Section 2.3 in order for `map` to have true parametric polymorphism, we need to have polymorphic effects (i.e. effects that are polymorphic over the regions

that they encompass). If we wanted this in Λ_{C}, the *boxed* types in Figure 3.1 would need to be amended with effect variables:

$$\begin{array}{rcll} \mu & = & \dots & \\ & | & \epsilon\ .\ \tau & \text{Effect polymorphism} \\ & | & \tau[\epsilon \mapsto \varphi] & \text{Effect instantiation} \end{array} \tag{3.1}$$

and effects amended so that they can make use of these effect variables:

$$\varphi ::= \dots \mid \varphi \cup \epsilon \tag{3.2}$$

and with this updated definition, we could then annotate `map` with the following effect-polymorphic type:

$$\texttt{map} : (\epsilon\ .\ (((\tau_1 \xrightarrow{\epsilon} \tau_2, \zeta_1) \times (\,\texttt{Vec}\ \ \tau_1, \zeta_2), \zeta_3) \xrightarrow{\varphi' \cup \epsilon} (\,\texttt{Vec}\ \ \tau_2, \zeta_4), \zeta_5), \zeta_6)$$

However, if we added effect polymorphism in this manner to the language we would run into issues later on when we start moving computations between the CPU and the GPU in Chapter 5.

All is not lost however. While we may lack full effect polymorphism in the language, large parts of polymorphism can still be simulated by clever use of region abstraction and instantiation, and by "truncating" the size of the effects that we are polymorphic over; limiting the number of parameterized regions allowed in the effect. Thus in Λ_{C} the type above for `map` is translated into the following (for a 2-truncation):

$$\begin{aligned} \texttt{map} : &(\Pi \rho_1' \geq \{\}.\overset{\{\rho\}}{-}\ .\ (\Pi \rho_2' \geq \{\}.\overset{\{\rho\}}{-}\ . \\ &(((\tau_1 \xrightarrow{\{\rho_1', \rho_2'\}} \tau_2, \zeta_1) \times (\,\texttt{Vec}\ \ \tau_1, \zeta_2), \zeta_3) \xrightarrow{\varphi' \cup \{\rho_1', \rho_2'\}} (\,\texttt{Vec}\ \ \tau_2, \zeta_4), \zeta_5), \zeta_6), \zeta_6) \end{aligned}$$

Truncating the polymorphism in effects this way allows us to translate effect polymorphism in the following manner: given a type $\tau' = (\epsilon\ .\ \tau, \zeta_1)$, we start by first replacing the effect variable ϵ with the set of all possible regions that the effect could be instantiated with—thus in the case of n-truncated effect polymorphism we replace $\varphi \cup \epsilon$ with $\varphi \cup \{\rho_1, \dots, \rho_n\}$ where $\rho_i \notin \varphi$, and where each "fresh" ρ_i is introduced by

a region abstraction with no bound, and the region abstraction is boxed in the same region as the effect abstraction:

$$(\Pi \rho_1 \succeq \{\}^{\{\rho\}}_{-} \ldots (\Pi \rho_n \succeq \{\}^{\{\rho\}}_{-} \,.\, \mathbb{E}^n_\tau(\tau, \{\epsilon \mapsto \{\rho_1, \ldots, \rho_n\}), \zeta_1) \ldots, \zeta_1)$$

After this translation, instantiating an effect variable amounts to instantiation by a given set of regions—the set of regions that the effect variable ϵ stood for originally—thus effect instantiation can be translated into a series of region instantiations after performing the above translation on effect-polymorphic code. It is worth emphasizing that since effects are *sets*, we can represent effect polymorphism up to and including n different regions on an n-polymorphic type.

The translation described above is formalized in Figure 3.5, where we are translating from the type- and effect system described in Equations (3.1) and (3.2) to the types and effects presented in Figure 3.1. It is worth noting again that while we can recover a large amount of effect polymorphism by this translation, we don't have full polymorphism over effects; we can only translate effects with a bounded number of regions in the effect set. While this lack of full, unbounded, effect polymorphism is still somewhat of a limitation, we can nevertheless represent large amounts of polymorphism in Λ_C and write functions that are functionally polymorphic while still avoiding the difficulties that come with full effect polymorphism.

3.3.4 The Type System

The typing judgements for the type system are syntax-directed and take the standard form for a region-typing system:

$$\Omega; \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$$

where Ω is the current set of valid computational regions, Δ is the current set of region constraints, Γ is a mapping from term variables to types, and φ is a set of

$$\boxed{\mathbb{E}^n_\tau(\tau,\sigma)}$$

$$\begin{array}{rcl}
\mathbb{E}^n_\tau(\texttt{bool},\sigma) & = & \texttt{bool} \\
\mathbb{E}^n_\tau(\texttt{int},\sigma) & = & \texttt{int} \\
\mathbb{E}^n_\tau((\mu,\rho),\sigma) & = & \mathbb{E}^n_\mu(\mu,\sigma,\rho)
\end{array}$$

$$\boxed{\mathbb{E}^n_\mu(\mu,\sigma,\zeta)}$$

$$\begin{array}{rcl}
\mathbb{E}^n_\mu(\tau_1 \xrightarrow{\varphi} \tau_2,\sigma,\zeta) & = & \left(\mathbb{E}^n_\tau(\tau_1,\sigma) \xrightarrow{\mathbb{E}^n_\varphi(\varphi,\sigma)} \mathbb{E}^n_\tau(\tau_2,\sigma),\zeta\right) \\
\mathbb{E}^n_\mu(\tau_1 \times \tau_2,\sigma,\zeta) & = & (\mathbb{E}^n_\tau(\tau_1,\sigma) \times \mathbb{E}^n_\tau(\tau_2,\sigma),\zeta) \\
\mathbb{E}^n_\mu(\texttt{Vec}\ \tau,\sigma,\zeta) & = & (\texttt{Vec}\ \mathbb{E}^n_\tau(\tau,\sigma),\zeta) \\
\mathbb{E}^n_\mu(\Pi\rho \geq \varphi \,.^{\varphi'}\, \tau,\sigma,\zeta) & = & \left(\Pi\rho \geq \mathbb{E}^n_\varphi(\varphi,\sigma) \,.^{\mathbb{E}^n_\varphi(\varphi',\sigma)}\, \mathbb{E}^n_\tau(\tau,\sigma),\zeta\right) \\
\mathbb{E}^n_\mu(\epsilon\ .\ \tau,\sigma,\zeta) & = & \left(\Pi\rho_1 \geq \{\} \,.^{\{\underline{\rho}\}} \ldots \left(\Pi\rho_n \geq \{\} \,.^{\{\underline{\rho}\}}\right.\right. \\
& & \quad \left.\left.\mathbb{E}^n_\tau(\tau,\sigma[\epsilon \mapsto \{\rho_1,\ldots,\rho_n\}]),\zeta\right)\ldots,\zeta\right) \\
& & \rho_1,\ldots,\rho_n \text{ fresh},\ \epsilon \notin \mathsf{dom}(\sigma) \\
\mathbb{E}^n_\mu(\epsilon\ .\ \tau,\sigma,\zeta) & = & \left(\Pi\rho_1 \geq \{\} \,.^{\{\underline{\rho}\}} \ldots \left(\Pi\rho_k \geq \{\} \,.^{\{\underline{\rho}\}}\, \mathbb{E}^n_\tau(\tau,\sigma),\zeta\right)\ldots,\zeta\right) \\
& & \epsilon \in \mathsf{dom}(\sigma),\ \sigma(\epsilon) = \{\rho_1,\ldots,\rho_k\} \\
\mathbb{E}^n_\mu(\tau[\epsilon \mapsto \varphi],\sigma,\zeta) & = & \left(\mathbb{E}^n_\tau(\tau,\sigma[\epsilon \mapsto \mathbb{E}^n_\varphi(\varphi,\sigma)]),\zeta\right)
\end{array}$$

$$\boxed{\mathbb{E}^n_\varphi(\varphi,\sigma)}$$

$$\begin{array}{rcl}
\mathbb{E}^n_\varphi(\{\rho_1,\ldots,\rho_k\},\sigma) & = & \{\rho_1,\ldots,\rho_k\} \\
\mathbb{E}^n_\varphi(\varphi \cup \epsilon,\sigma) & = & \mathbb{E}^n_\varphi(\varphi,\sigma) \cup \sigma(\epsilon)
\end{array}$$

Figure 3.5: Translation of truncated effect polymorphism to Λ_{C}.

bounding effects for the expression e, and is read "expression e has type τ and bounding effects φ." Regions in Λ_{C} have a natural subtyping relationship associated with them due to the fact that the region stack behaves in a LIFO fashion. We express this relationship formally through the judgement $\Omega;\Delta \vdash_{\mathrm{rr}} \rho \geq \rho'$ and the region constraint relationship $\rho \geq \rho'$, which is read as "ρ' outlives ρ," and that ρ occurs to the left of ρ' in the (leftwards-growing) region stack. Similarly, $\Omega;\Delta \vdash_{\mathrm{rr}} \rho \geq \varphi$ is read as saying that every region in φ outlives ρ.

Context and Region Judgements The context and type well-formedness rules for Λ_C are presented in Figures 3.6 to 3.8 and the typing judgements for Λ_C are presented in Figure 3.9 and Figure 3.10. Region contexts Ω are ordered lists of region variables, and region constraint contexts Δ are ordered lists of region constraints. Value contexts Γ are lists of variables and types. The various type, context, and effect well-formedness judgements are summarized in the following table:

Judgement	Meaning
$\vdash_{\text{reg}} \Omega$	Region context Ω is well-formed
$\Omega \vdash_{\text{rctxt}} \Delta$	Region constraint context Δ is well-formed
$\Omega; \Delta \vdash_{\text{vctxt}} \Gamma$	Typing context Γ is well-formed
$\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi$	In valid contexts Ω, Δ, and Γ, the latent effect φ is well-formed.
$\Omega; \Delta \vdash_{\text{btype}} \mu$	The boxed type μ is well-formed
$\Omega \vdash_{\text{place}} \rho$	The region ρ is a valid region variable in Ω
$\Omega; \Delta \vdash_{\text{type}} \tau$	The type τ is well-formed
$\Omega; \Delta \vdash_{\text{eff}} \varphi$	The latent effect φ is well-formed w.r.t. Ω and Δ
$\Omega; \Delta \vdash_{\text{rr}} \rho \succeq \rho'$	If region ρ is alive, then ρ' is live as well (region ρ' outlives ρ)
$\Omega; \Delta \vdash_{\text{re}} \rho \succeq \varphi$	If region ρ is alive, then all regions in φ are alive
$\Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho$	The region ρ is a region in the latent effect φ
$\Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'$	The latent effect set φ' is a subset of the latent effect φ
$\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$	The term e has type τ and effects bounded by φ

and where $\mathit{regions}(\Delta)$ returns the set of all regions contained within Δ, and is defined as follows:

$$\begin{array}{lcl} \mathit{regions}(\cdot) & = & \{\} \\ \mathit{regions}(\Delta, \rho \succeq \varphi) & = & \mathit{regions}(\Delta) \cup \varphi \cup \{\rho\} \end{array}$$

While the judgements regarding the well-formedness of the various contexts and types are straightforward, the judgements for region liveness ($\Omega; \Delta \vdash_{\text{rr}} \rho \succeq \rho'$ and $\Omega; \Delta \vdash_{\text{re}} \rho \succeq \varphi$), region membership ($\Omega; \Delta \vdash_{\text{ee}} \varphi \ni \rho$), and effect subsumption ($\Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'$) are rather more interesting. In particular, there is a mutually recursive relationship between the effect subsumption and region liveness judgements that is introduced via the following rules that respectively allow us to "pick out" and judge region liveness from the effect subsumption judgement, and also judge effect subsumption from region liveness:

$\boxed{\vdash_{\text{reg}} \Omega}$

$$\frac{}{\vdash_{\text{reg}} \{\underline{\rho}\}} \qquad \frac{\vdash_{\text{reg}} \Omega \qquad \rho \in RVars}{\vdash_{\text{reg}} \rho, \Omega}$$

$\boxed{\Omega \vdash_{\text{place}} \rho}$

$$\frac{\vdash_{\text{reg}} \Omega \qquad \rho \in \Omega}{\Omega \vdash_{\text{place}} \rho}$$

$\boxed{\Omega \vdash_{\text{rctxt}} \Delta}$

$$\frac{}{\vdash_{\text{rctxt}} \cdot} \qquad \frac{\begin{array}{cc}\Omega \vdash_{\text{rctxt}} \Delta & \rho \notin regions(\Delta) \\ \Omega \vdash_{\text{place}} \rho & \Omega; \Delta \vdash_{\text{eff}} \varphi\end{array}}{\Omega \vdash_{\text{rctxt}} \Delta, \rho \succeq \varphi}$$

$\boxed{\Omega; \Delta \vdash_{\text{vctxt}} \Gamma}$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega; \Delta \vdash_{\text{vctxt}} \cdot} \qquad \frac{\begin{array}{cc}\Omega; \Delta \vdash_{\text{vctxt}} \Gamma & \Omega; \Delta \vdash_{\text{type}} \tau \\ \multicolumn{2}{c}{x \notin \mathsf{dom}(\Gamma)}\end{array}}{\Omega; \Delta \vdash_{\text{vctxt}} \Gamma, x : \tau}$$

$\boxed{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}$

$$\frac{\Omega; \Delta \vdash_{\text{vctxt}} \Gamma \qquad \Omega; \Delta \vdash_{\text{eff}} \varphi}{\vdash_{\text{ctxt}} \Omega; \Gamma; \varphi}$$

Figure 3.6: Well formedness rules for contexts

$$\frac{\begin{array}{cc}\Omega \vdash_{\text{rctxt}} \Delta & \Omega \vdash_{\text{place}} \rho \\ \multicolumn{2}{c}{\Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}}\end{array}}{\Omega; \Delta \vdash_{\text{rr}} \rho \succeq \rho_i} \qquad \frac{\begin{array}{cc}\multicolumn{2}{c}{\Omega \vdash_{\text{rctxt}} \Delta} \\ \Omega; \Delta \vdash_{\text{rr}} \rho \succeq \rho_i & i = 1, \ldots, n\end{array}}{\Omega; \Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

Further, note that the rules for both $\Omega; \Delta \vdash_{\text{rr}} \rho \succeq \rho'$ and $\Omega; \Delta \vdash_{\text{re}} \rho \succeq \varphi$ simply formalize the reflexive and transitive closure of the syntactic constraints placed on Δ; asserting a particular "outlived by" relation between a region and an effect.

$\boxed{\Omega; \Delta \vdash_{rr} \rho \succeq \rho'}$

$$\frac{\Omega \vdash_{rctxt} \Delta \qquad \Omega \vdash_{place} \rho \qquad \Delta \vdash_{re} \rho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}}{\Omega; \Delta \vdash_{rr} \rho \succeq \rho_i} \qquad \frac{\Omega \vdash_{place} \rho}{\Delta \vdash_{rr} \rho \succeq \rho}$$

$$\frac{\Omega; \Delta \vdash_{rr} \rho \succeq \rho' \qquad \Omega; \Delta \vdash_{rr} \rho' \succeq \rho''}{\Omega; \Delta \vdash_{rr} \rho \succeq \rho''}$$

$\boxed{\Omega; \Delta \vdash_{re} \rho \succeq \varphi}$

$$\frac{\Omega \vdash_{rctxt} \Delta \qquad \Omega; \Delta \vdash_{rr} \rho \succeq \rho_i \quad i = 1, \ldots, n}{\Omega; \Delta \vdash_{re} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

$\boxed{\Omega; \Delta \vdash_{er} \varphi \ni \rho}$

$$\frac{\Omega \vdash_{place} \rho_i \qquad \Omega; \Delta \vdash_{eff} \{\rho_1, \ldots, \rho_n\}}{\Omega; \Delta \vdash_{er} \{\rho_1, \ldots, \rho_n\} \ni \rho_i}$$

$\boxed{\Omega; \Delta \vdash_{ee} \varphi \supseteq \varphi'}$

$$\frac{\Omega; \Delta \vdash_{eff} \varphi \qquad \Omega; \Delta \vdash_{er} \varphi \ni \rho_i}{\Omega; \Delta \vdash_{ee} \varphi \supseteq \{\rho_i\}_{i=1}^{n}}$$

Figure 3.7: Well formedness rules for regions and effects

Typing Rules The key judgement in any region calculus is the typing rule for `letregion`:

$$\frac{\Omega; \Delta \vdash_{type} \tau \qquad \vdash_{ctxt} \Omega; \Delta; \Gamma; \{\rho_1, \ldots, \rho_n\} \qquad \rho, \Omega; \Delta, \rho \succeq \{\rho_1, \ldots, \rho_n\}; \Gamma \vdash_{exp} e : \tau, \{\rho_1, \ldots, \rho_n, \rho\}}{\Omega; \Delta; \Gamma \vdash_{exp} \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau, \{\rho_1, \ldots, \rho_n\}}$$

where there is an implicit antecedent that $\rho \notin \mathit{regions}(\Delta)$, and the judgements $\Omega; \Delta \vdash_{type} \tau$ and $\vdash_{ctxt} \Delta; \Gamma; \varphi$ ensure that ρ does not appear in the result type, or in the types of the value environment. The new region is then related to the current set of effects φ

$\boxed{\Omega; \Delta \vdash_{\text{btype}} \mu}$

$$\frac{\Omega; \Delta \vdash_{\text{type}} \tau_1 \quad \Omega; \Delta \vdash_{\text{eff}} \varphi \quad \Omega; \Delta \vdash_{\text{type}} \tau_2}{\Omega; \Delta \vdash_{\text{btype}} \tau_1 \xrightarrow{\varphi} \tau_2} \qquad \frac{\Omega; \Delta \vdash_{\text{type}} \tau_1 \quad \Omega; \Delta \vdash_{\text{type}} \tau_2}{\Omega; \Delta \vdash_{\text{btype}} \tau_1 \times \tau_2}$$

$$\frac{\Omega; \Delta \vdash_{\text{eff}} \varphi' \quad \rho, \Omega; \Delta, \rho \geq \varphi' \vdash_{\text{eff}} \varphi \quad \rho, \Omega; \Delta, \rho \geq \varphi' \vdash_{\text{type}} \tau}{\Omega; \Delta \vdash_{\text{btype}} \Pi \rho \geq \varphi' .^{\varphi} \tau}$$

$\boxed{\Omega \vdash_{\text{place}} \rho}$

$$\frac{\rho \in \Omega}{\Omega \vdash_{\text{place}} \rho}$$

$\boxed{\Omega; \Delta \vdash_{\text{type}} \tau}$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega; \Delta \vdash_{\text{type}} \texttt{int}} \qquad \frac{\Omega \vdash_{\text{rctxt}} \Delta}{\Omega; \Delta \vdash_{\text{type}} \texttt{bool}} \qquad \frac{\Omega; \Delta \vdash_{\text{btype}} \mu \quad \Omega \vdash_{\text{place}} \rho}{\Omega; \Delta \vdash_{\text{type}} (\mu, \rho)}$$

$\boxed{\Omega; \Delta \vdash_{\text{eff}} \varphi}$

$$\frac{\Omega \vdash_{\text{rctxt}} \Delta \quad \Omega \vdash_{\text{place}} \rho_i}{\Omega; \Delta \vdash_{\text{eff}} \{\rho_i\}_{i=1}^{n}}$$

Figure 3.8: Well formedness rules for types

by adding the constraint that ρ is outlived by all regions in φ, and is also added in to the set of effects for the typechecking of the body of the `letregion` form.

$$\boxed{\Omega; \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} i : \texttt{int}, \varphi}
\qquad
\frac{\begin{array}{c}\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 : \texttt{int}, \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_2 : \texttt{int}, \varphi\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 * e_2 : \texttt{int}, \varphi}$$

$$\frac{\begin{array}{c}\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 : \texttt{int}, \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_2 : \texttt{int}, \varphi\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 < e_2 : \texttt{bool}, \varphi}
\qquad
\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \texttt{False} : \texttt{bool}, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \texttt{True} : \texttt{bool}, \varphi}
\qquad
\frac{\begin{array}{c}\Omega; \Delta; \Gamma \vdash_{\text{exp}} e_b : \texttt{bool}, \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_t : \tau, \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_f : \tau, \varphi\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau, \varphi}$$

$$\frac{\begin{array}{c}\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \\ x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) = \tau\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} x : \tau, \varphi}
\qquad
\frac{\begin{array}{c}\Omega; \Delta; \Gamma, x : \tau_1 \vdash_{\text{exp}} e' : \tau_2, \varphi' \\ \Omega \vdash_{\text{place}} \rho \qquad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \lambda x : \tau_1.^{\varphi'} e'\ \texttt{at}\ \rho : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho), \varphi}$$

$$\frac{\begin{array}{c}\Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho), \varphi \\ \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau_1, \varphi \qquad \Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1\ e_2 : \tau_2, \varphi}$$

$$\frac{\begin{array}{c}\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_1 : \tau_1, \varphi \\ \Omega; \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau_2, \varphi \\ \Omega \vdash_{\text{place}} \rho \qquad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} (e_1, e_2)\ \texttt{at}\ \rho : (\tau_1 \times \tau_2, \rho), \varphi}
\qquad
\frac{\begin{array}{c}\Omega; \Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho), \varphi \\ \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \texttt{fst}\ e : \tau_1, \varphi}$$

$$\frac{\begin{array}{c}\Omega; \Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho), \varphi \\ \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho\end{array}}{\Omega; \Delta; \Gamma \vdash_{\text{exp}} \texttt{snd}\ e : \tau_2, \varphi}$$

Figure 3.9: The base typing rules for Λ_C based on the Bounded Effect Calculus of Fluet and Morrisett [65]

$$\frac{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e_i : \tau,\varphi \qquad \Omega \vdash_{\text{place}} \rho \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} [e_1,\ldots,e_n]\ \texttt{at}\ \rho : (\texttt{Vec}\ \tau,\rho),\varphi}$$

$$\frac{\rho,\Omega;\Delta,\zeta \succeq \varphi'';\Gamma \vdash_{\text{exp}} u' : \tau,\varphi' \qquad \Omega \vdash_{\text{place}} \rho \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} \lambda\,\zeta \succeq \varphi''.^{\varphi'} u'\ \texttt{at}\ \rho : (\Pi\zeta \succeq \varphi''.^{\varphi'}\tau,\rho),\varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e : (\Pi\rho \succeq \varphi''.^{\varphi'}\tau,\rho_1),\varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_1 \qquad \Omega \vdash_{\text{place}} \rho_2 \qquad \Omega;\Delta \vdash_{\text{re}} \rho_2 \succeq \varphi'' \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi[\rho_2/\rho] \supseteq \varphi'[\rho_2/\rho]}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e[\rho_2] : \tau[\rho_2/\rho],\varphi}$$

$$\frac{\Omega;\Delta;\Gamma,f : \tau \vdash_{\text{exp}} u : \tau,\varphi}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} \texttt{fix}\ f : \tau.u : \tau,\varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_1 : \tau_1,\varphi_1 \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_2 : \tau_2,\varphi_2 \qquad \Omega \vdash_{\text{place}} \rho \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_2}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e_1 \| e_2\ \texttt{at}\ \rho : (\tau_1 \times \tau_2,\rho),\varphi}$$

$$\frac{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e : (\tau_1 \xrightarrow{\varphi'} \tau_2,\rho'),\varphi \qquad \Omega \vdash_{\text{place}} \rho' \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi' \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_i : \tau_1,\varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho' \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} \texttt{parmap}\ e\ [e_1,\ldots,e_n]\ \texttt{at}\ \rho : (\texttt{Vec}\ \tau_2,\rho),\varphi}$$

$$\frac{\Omega;\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\{\rho_1,\ldots,\rho_n\} \qquad \rho,\Omega;\Delta,\rho \succeq \{\rho_1,\ldots,\rho_n\};\Gamma \vdash_{\text{exp}} e : \tau,\{\rho_1,\ldots,\rho_n,\rho\}}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau,\{\rho_1,\ldots,\rho_n\}}$$

Figure 3.10: The typing rules for Λ_{C} (continued).

A subtle—yet important—difference arises in the pair and parallel evaluation rules:

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_1 : \tau_1,\varphi \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_2 : \tau_2,\varphi \qquad \Omega \vdash_{\text{place}} \rho \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} (e_1,e_2)\ \texttt{at}\ \rho : (\tau_1 \times \tau_2,\rho),\varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho \qquad \Omega \vdash_{\text{place}} \rho \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_1 : \tau_1,\varphi_1 \qquad \Omega;\Delta;\Gamma \vdash_{\text{exp}} e_2 : \tau_2,\varphi_2 \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_2}{\Omega;\Delta;\Gamma \vdash_{\text{exp}} e_1 \| e_2\ \texttt{at}\ \rho : (\tau_1 \times \tau_2,\rho),\varphi}$$

In particular, the set of effects that are inferred for e_1 and e_2 are inferred separately, and then joined to form φ. Such a φ is guaranteed to exist since we can always take $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. It is straightforward to see that we could also elide the separate effect sets for e_1 and e_2, however keeping the effect sets for parallel expressions disjoint in this manner is useful as this will provide a natural way to separate and reason about the regions used by the different threads of execution.

As mentioned earlier, each abstraction carries with it a set of latent effects that detail the memory access characteristics of the abstraction. These latent effects are then used in the two application judgements in the type system in order to ensure that the set of effects that are inferred for the application subsume the latent effects attached to the abstraction that is being applied. We can see this in the rule for function application where we ensure that the latent effect of the function being applied is subsumed by the effect inferred; $\Omega; \Delta \vdash_{\text{ee}} \varphi \supset \varphi'$. In the case of region application, we also need to ensure that the region that is being applied outlives the lower-bound effects for the region abstraction that the region is being applied to.

3.3.5 Typing Heap Values

In order to type heap values—which may contain region *variables*—we need a way to connect region variables in Ω to their corresponding region names in the region stack $\mathcal{S}$. Further, since values come about through reductions specified by the semantics of Λ_{C} and are themselves stored in the region stack, we need a way of relating the behavior of regions in the dynamic semantics and the static semantics for heap values. In order to do this, we define the *runtime region context* $\mathcal{E}$ which replicates the mapping of region variables to region names that will be done in the environment E in the cost semantics. Note that the runtime environment E maps variables to heap

$$\boxed{embed(\mathcal{E})}$$

$$\begin{array}{lcl} embed([]) & = & [] \\ embed((\rho, r) : rest) & = & (\texttt{inl}\ \rho, \texttt{inr}\ r) : embed(rest) \end{array}$$

$$\boxed{proj(E)}$$

$$\begin{array}{lcl} proj([]) & = & [] \\ proj((\texttt{inl}\ \rho, \texttt{inl}\ r) : rest) & = & (\rho, r) : proj(rest) \\ proj(x : rest) & = & proj(rest) \end{array}$$

Figure 3.11: Definition of runtime environment projection and embedding.

values, and region variables to region names by the following where $+$ represents the coproduct:

$$E : (\mathsf{Vars} + \mathsf{RVars}) \rightarrow (v + \mathsf{RNames})$$

and $\mathcal{E}$ is given by the following mapping:

$$\mathcal{E} : \mathsf{RVars} \rightarrow \mathsf{RNames}$$

We can therefore easily define the *embedding*, $embed(\mathcal{E})$, of $\mathcal{E}$ into E as well as the *projection*, $proj(E)$, of E into $\mathcal{E}$. The definitions of these are given in Figure 3.11

Figure 3.12 presents the rules for the primary judgement $\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v : \tau$, as well as judgements for $\mathcal{E} \stackrel{\cong}{\leftarrow} \mathcal{S}$, $\mathcal{E} \stackrel{\cong}{\rightarrow} \mathcal{S}$,$\mathcal{E} \cong \mathcal{S}$, and $\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega$. The former judgement asserts that the heap value v is well-typed while the latter judgements express region stack and runtime region-context consistency relations, culminating in $\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega$ which relates runtime region stacks and contexts with well-formedness of the static region context Ω. The rules for $\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v : \tau$ are similar to those for the related terms in Figures 3.9 and 3.10. Note that there are no effects associated with these judgements; once an expression has been evaluated to a value then it no longer gives rise to a computational effect.

$\mathcal{E} : \mathsf{RVars} \to \mathsf{RNames}$ Runtime region context

$\mathcal{E} \overset{\cong}{\to} \mathcal{S} := \forall \rho \in \mathsf{dom}(\mathcal{E}) \,.\, \mathcal{E}(\rho) \in \mathsf{dom}(\mathcal{S})$

$\mathcal{E} \overset{\cong}{\leftarrow} \mathcal{S} := \forall r \in \mathsf{dom}(\mathcal{S}) \,.\, r \in \mathsf{range}(\mathcal{E})$

$\mathcal{E} \cong \mathcal{S} := \mathcal{E} \overset{\cong}{\leftarrow} \mathcal{S} \wedge \mathcal{E} \overset{\cong}{\to} \mathcal{S}$

$\boxed{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega}$

$$\frac{\mathcal{E} \cong \mathcal{S} \qquad \mathsf{dom}(\mathcal{E}) = \Omega}{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega}$$

$\boxed{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v : \tau}$

$$\frac{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} i : \texttt{int}} \qquad \frac{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} b : \texttt{bool}}$$

$$\frac{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v_1 : \tau_1 \qquad \mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v_2 : \tau_2}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v_i : \tau}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} [v_1, \ldots, v_n] : \texttt{Vec}\ \tau}$$

$$\frac{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega \qquad \Omega; \cdot; x : \tau_1 \vdash_{\text{exp}} e : \tau_2, \varphi}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} \texttt{clos}\ (x : \tau_1, e, E) : \tau_1 \xrightarrow{\varphi} \tau_2}$$

$$\frac{\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega \qquad \Omega, \rho; \cdot, \rho \geq \varphi; \cdot \vdash_{\text{exp}} u : \tau, \varphi'}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} \texttt{rclos}\ (\rho, \varphi, \varphi', u) : \Pi \rho \geq \varphi .^{\varphi'} \tau}$$

$$\frac{\ell = (r, o) \qquad r \in \mathsf{dom}(\mathcal{S}) \qquad (\rho, r) \in \mathcal{E} \qquad \mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} \mathcal{S}(\ell) : \tau}{\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} \ell : (\tau, \rho)}$$

Figure 3.12: Typing rules for values.

3.4 Memory Model

This section describes the memory model and layout for Λ_C and details the creation, layout, and access of regions in the underlying memory model for the language. This is a region-based memory model with immutable regions and reference counting to deal with task parallelism. While region-based memory models (RBMM) have been well-studied in the general setting [8], region-based memory models for heterogeneous languages are a relatively new avenue of research having only been introduced in the past three years by Holk et al. [7] and Holk [9]. Since Λ_C is designed with the view towards later on being generalized in Chapter 5 to a heterogeneous language, we use this recent work on regions to inform the memory model that is developed here.

3.4.1 Region Based Memory Management

Representing the heap as a series of "heaplets" on the *region stack* that grows and shrinks as we encounter `letregion` forms during the reduction will greatly simplify the determination of the space usage in the cost semantics.[3] However, introducing a RBMM system especially in a parallel setting introduces its own challenges that need to be overcome. In particular, we need to be careful about disallowing race conditions for allocation within regions. Before we can conquer these challenges we must first define the lay of the land. We thus start out defining what exactly a region 'is':

Definition 3.4.1 (Region). *We define a* region R *to be a tuple* $(\mathsf{ap}, s, \mathsf{refcnt}, v)$ *where:*

- ap *is the allocation pointer for the region. Access is atomic;*
- s *is the size (in bytes) of the region. Access is atomic;*

[3]And as we will see later, determining transfer costs to- and from the GPU.

- refcnt *is the reference count for the region. Access is atomic;*
- $\nu : \mathbb{N} \rightarrow v$ *is a mapping from offsets* o *in the underlying memory for the region to heap values* v*. The offsets are bounded by the size* s *of the region.*

Since regions are simply a 4-tuple of values to manage the book-keeping for the underlying memory of the region, creating them is a fairly straightforward process with the exception of initializing the allocation pointer; since we must record the base of the region (i.e., where the region begins in global memory) this involves in the simplest case, global synchronous access to a global offset. Therefore in order to simplify things we assume that creation of regions involves a singular atomic fetch-and-add on the global allocation pointer (where we add a predetermined base size $\mathsf{allocSize}$ to the global allocation pointer).

Bumping a global allocation pointer to determine allocation areas for new regions is a non-optimal algorithm since we will quickly run out of space in which to create regions, with a better solution being to use a global free-list of start locations for regions (and push and pop to these). We do not do this however since it is not critical for the theory, and merely complicates things. We thus make the following simplifying assumptions about allocation (and de-allocation) of regions throughout the rest of this dissertation:

1. All creation of regions is race-free (either through a critical section, or since they can be created atomically);
2. Once a region is popped from the region stack the portion of memory used by that region may now be used either by surrounding regions, and/or may be used to create other regions;
3. The size of global memory is large enough that all allocations of new regions succeed.

$$\frac{|v| + \mathsf{ap} > |R| \qquad o = \mathsf{ap} \qquad n = max(\mathsf{allocSize}, |v| + \mathsf{ap} - |R|)}{R; v \downarrow o@(\mathsf{ap} + |v| + 1, s + n, \mathsf{refcnt}, \nu[o \mapsto v])}$$

$$\frac{|v| + \mathsf{ap} \leq |R| \qquad o = \mathsf{ap}}{R; v \downarrow o@(\mathsf{ap} + |v| + 1, s, \mathsf{refcnt}, \nu[o \mapsto v])} \qquad \frac{\ell = (r, o) \qquad \mathcal{S}(r) = (\mathsf{ap}, _, _, \nu) \qquad o < \mathsf{ap} \qquad o \in \mathsf{dom}(\nu) \qquad v = \nu(o)}{\mathcal{S}; \ell \uparrow v}$$

Figure 3.13: Allocation and Reading

In order to lessen the notational burden, we shall often conflate the region $R = (\mathsf{ap}, s, \mathsf{refcnt}, \nu)$ with its underlying memory ν where there is no ambiguity. e.g., $|R| = |\nu|$, $\mathsf{dom}(R) = \mathsf{dom}(\nu)$ etc.

Once a region has been allocated, allocations within that region simply involve an atomic bumping of the region allocation pointer, with the only difficulty arising when we run out of space in the underlying memory for the region. In the latter case we extend the existing region's underlying memory with enough space and then retry the allocation.[4] This allocation into regions is formalized by the judgement $R; v \downarrow o"@"R'$ which says that allocation of the heap value v is allocated at offset o in the modified region R'. We similarly have a rule $\mathcal{S}; \ell \uparrow v$ for reading locations in memory which says that reading location ℓ results in value v with respect to region stack $\mathcal{S}$. The judgements for these two forms can be found in Figure 3.13.

A critical aspect of the regions for Λ_{C} is that their memory is immutable. Firstly, by enforcing immutability along with the allocation strategy within regions, the memory within a region is not subject to race conditions. Secondly, making regions immutable provides the internal structure that we will need later on in order to easily merge regions that have evolved in parallel on both host and device; due to immutability, during the merge process we know that those sections of the region

[4]As a simplifying assumption, we will assume that this simply involves increasing the size of the underlying memory, and that we do not have to reallocate the whole underlying memory buffer in some other area of global memory.

that are shared across host and device will be the same and therefore we don't have to worry about "merge conflicts." If we allowed mutable memory, we could still disallow race-conditions for allocation, but we would lose this ability to easily merge regions together that have evolved in parallel on both host and device.

Since the $\geq$ relation naturally defines a preorder on the region variables, it is natural to expect that the underlying regions of memory that these map to will have a similar structure. In particular, there is a relationship between region variables in Δ and the backing regions in the region stack. To see this, we first define a relation $:\geq$ on the region stack.

Definition 3.4.2 (Definition of $:\geq$)**.** *Given a region stack $\mathcal{S}$, we say that $\mathcal{S} \vdash R_1 :\geq R_2$ if there exist region stacks $\mathcal{S}_1$,$\mathcal{S}_2$, and $\mathcal{S}_3$ (all possibly empty) such that for the current region stack $\mathcal{S}$,*

$$\mathcal{S} = \mathcal{S}_1, r_1 \mapsto R_1, \mathcal{S}_2, r_2 \mapsto R_2, \mathcal{S}_3$$

If $\mathcal{S} \vdash R_1 :\geq R_2$ we say that R_1 is an *ancestor* of R_2. We also define any region to be an ancestor of itself (i.e., $\mathcal{S} \vdash R :\geq R$ for any region R) (so $:\geq$ is a reflexive relation). Furthermore, the region stack $\mathcal{S}$ forms a bounded lattice with respect to this relation $:\geq$:

Theorem 3.4.1. *$(\mathcal{S}, :\geq, \underline{R}, \overline{R})$ forms a bounded lattice with upper bound $\underline{R}$ and lower bound $\overline{R}$. Where $\underline{R}$ is the starting region, and $\overline{R}$ is the region at the top of the stack $\mathcal{S}$.*

Proof. See Appendix B.2 for proof. □

3.5 The Abstract Machine Model

As was mentioned in § 2.1 of the literature review, a cost semantics is meant to be a *mostly* extensional way of describing the costs of programs. However, in order to

actually get concrete costs out of these models a certain amount of *intensionality* must be added. In particular, since we wish to describe intensional properties of the language we need to also take these intensional properties of the language into account. Yet, we still wish the cost semantics to be as extensional as possible.

In order to reconcile the need for intensionality, with our desire for extensionality, we parameterized the costs that we build up by an *abstract machine model* that describes the intensional aspects of the underlying machine. This then allows us to inject the intensional aspects of costs into our (extensional) semantic model. In order to take full advantage of this parameterization—which we will need in Chapter 5 when we start talking about combining CPU and GPU computation—the cost semantics is described in terms of a low-level (big-step) operational semantics which reads and writes to locations in memory.[5] We can then phrase the costs that are derived in terms of the amount of time it takes to access, communicate, and write to the memory in this abstract machine model. This conscious decoupling of the machine model that determines the "base" costs from the actual operational-semantic description of how to combine the costs that we accrue, means that the actual concrete costs that we eventually derive are always with respect to a given machine model; we have one cost semantics which can be instantiated with a given machine model to derive concrete costs.

Abstract machine models exist for many different architectures and types of computational devices (indeed, we will be using one of these in Chapter 4 to describe the costs there), and are (in general) quite well studied artifacts. They represent an important abstraction, since they are meant to factor in the major contributors to machine costs such as memory access and communication costs, while at the same time not providing such a detailed theory that it becomes unusable and cumbersome.

[5]With the exception of `bool`s and `int`s which are stored unboxed in memory but still incur a read-/write overhead.

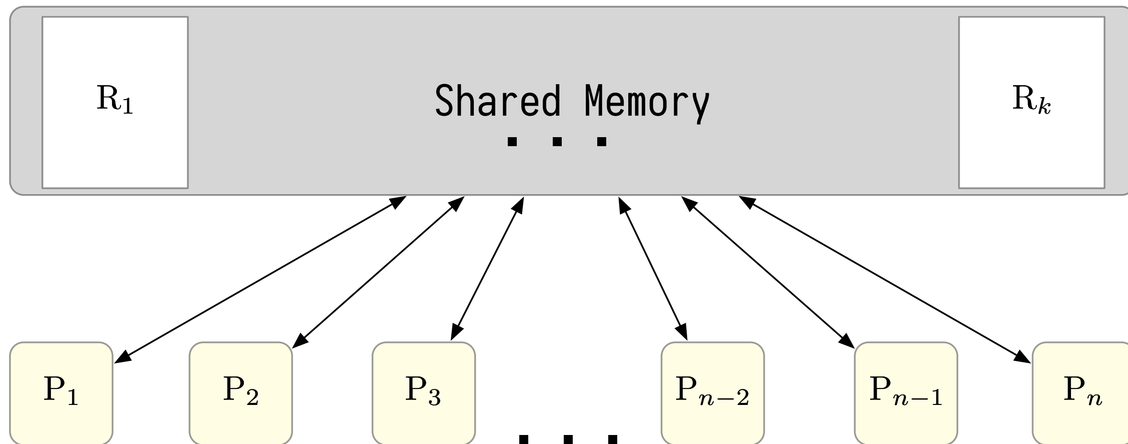


Figure 3.14: Layout of the PRAM model.

There are a number of different well-known abstract machine models that exist for the CPU, however, we will be using (only) one of the most common of these here: the Parallel Random Access Machine or PRAM [20] model with Concurrent Read and Concurrent Write (CRCW). While other models such as LogP [66] and BSP [67] can be used, we have chosen the PRAM model due to its simplicity while still offering relatively accurate descriptions of asymptotic complexity.

The PRAM model consists of a set of P processors that share a single memory. Any location in this shared memory can be accessed in a single unit of time (one single instruction), and each processor can perform an arithmetic, logical, or memory access operation in a single time unit. Moreover each execution step performed by a given processor is synchronized with all other currently running processors. A diagram of the PRAM model can be found in Figure 3.14.

The main factor that we are utilizing in the PRAM model is the fact that the memory access times are uniform. However, as we saw in Section 2.1.2 accessing memory at different levels of the memory hierarchy can have significant effects on the speed at which that data can be accessed, read, and written. Thus, if we wanted to get more accurate cost predictions we could change the underlying machine model that

we consider to either the I/O [23] or ideal-cache [24] models. While we could possibly use these more advanced machine models here instead of the PRAM model—which would give us tighter bounds on the costs that are predicted—the amount of complexity that they would add to the cost semantics is significant and not critical to the work being presented here, and serves to complicate the exposition without much payoff. We therefore leave the exploration of how these other machine models might effect profiling accuracy to future investigation and we discuss this briefly in Section 7.2.

3.6 Cost Semantics for Λ_{C}

Now that we have defined the machine model for Λ_{C}, we can develop the cost theory. Since we are dealing with a parallel language, the scheduling of the threads can directly affect the space and time usage of a program. In order to account for these different schedules, we use a similar technique to that introduced by Spoonhower et al. [6] whereby the cost semantics build a series-parallel DAG of operations on which we then impose different traversal strategies. We then use the machine model that we have just defined to develop costs for the nodes that represent the underlying memory accesses of the computation in the cost graph that we build.

3.6.1 Cost Graphs

While we use similar techniques to what we have seen before in the building of the DAG, we make a slight departure from what we have seen previously in Chapter 2. While previous techniques have built an *unweighted* DAG we instead build a weighted DAG for the cost graph. This weighting of the graph edges is used to factor in that certain “single steps” (graph edges) in the cost graph might be more expensive; such as allocating a new region of memory. With this graph structure in mind,

$$
\begin{array}{llll}
\alpha \in VVars \mid \delta \in \mathbb{N} & & & \\
e & ::= & (\alpha_1, \alpha_2, \delta) & \text{Directed Weighted Edges} \\
V & ::= & \{\alpha_1, \ldots, \alpha_n\} & \text{Vertex Set} \\
E & ::= & \{e_1, \ldots, e_n\} & \text{Edge Set} \\
G, c & ::= & (\alpha_1, \alpha_2, V, E) & \text{Graphs}
\end{array}
$$

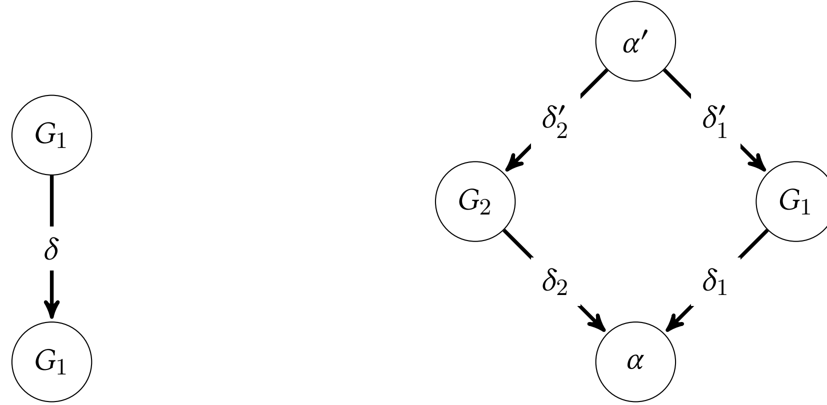
$$
\begin{array}{ll}
[\alpha] & = (\alpha, \alpha, \{\alpha\}, \emptyset) \\
[(\alpha_1, \alpha_2, \delta)] & = (\alpha_1, \alpha_2, \{\alpha_1, \alpha_2\}, \{(\alpha_1, \alpha_2, \delta)\}) \\
(\alpha'_1, \alpha_1, V, E) \oplus_\delta (\alpha'_2, \alpha_2, V', E') & = (\alpha'_1, \alpha_2, V \cup V', E \cup E' \cup \{(\alpha_1, \alpha'_2, \delta)\}) \\
(\alpha'_1, \alpha_1, V, E) \; {}^{\delta'_1}_{\delta_1}\!\otimes^{\delta'_2}_{\delta_2} \; (\alpha'_2, \alpha_2, V', E') & = \\
\quad (\alpha', \alpha, V \cup V' \cup \{\alpha', \alpha\}, E \cup E' \cup \{(\alpha', \alpha'_1, \delta'_1), (\alpha', \alpha'_2, \delta'_2), (\alpha_1, \alpha, \delta_1), (\alpha_2, \alpha, \delta_2)\}) & (\alpha, \alpha' \text{ fresh})
\end{array}
$$

Figure 3.15: Cost graph operations

we define the grammar for graphs along with core graph combinators in Figure 3.15 and show how these combinators join cost graphs in Figure 3.16. More generally, the notation, terminology, and structuring of the graphs that we use does not differ much from standard graph terminology.

Since we aren't concerned with expressing data parallelism—and hence only need to express task parallelism in the cost graph—the graphs that we build simply need to be able to express two computations either running in parallel or in series. We thus only need two core graph combinators: $G_1 \oplus_\delta G_2$ for serial composition of graphs (run one computation after the other) by an edge with weight δ; and $G_1 \; {}^{\delta'_1}_{\delta_1}\!\otimes^{\delta'_2}_{\delta_2} \; G_2$ for parallel composition of the graphs (run the two computations in parallel), where δ'_1 and δ'_2 represent the weights of the incoming edges to G_1 and G_2 respectively, and likewise δ_1 and δ_2 the outgoing edge weights for the new edges that are added to the graph. In the case that each edge weight is 1, we simply elide these weights and write $\oplus$ and $\otimes$.

The low-level (heap-location-based) operational semantics for the language are responsible for building the cost graphs that we analyze to determine the cost of a



The serial composition of DAGs G_1 and G_2 given by $G_1 \oplus_\delta G_2$.

The parallel composition of DAGs G_1 and G_2 given by $G_1 \; {}^{\delta_1'}_{\delta_1}\!\otimes^{\delta_2'}_{\delta_2} \; G_2$ Note that both α and α' are newly created ("fresh") nodes in the graph.

Figure 3.16: Cost combinators on DAG-based cost models.

program. The rules for the operational semantics and how they build the cost graph are presented in Figures 3.18 and 3.19 where the judgement

$$E; \mathcal{S}; e \Downarrow v; \mathcal{S}'; c$$

is read "in environment E and with region stack $\mathcal{S}$, the expression e evaluates to value v with updated region stack $\mathcal{S}'$ and cost graph c." The fact that this is a low-level operational semantics is important for the correctness of the theory, since the underlying costs are calculated based on the cost of reads and writes to memory.

Now that we are interested in the costs of allocations, lookups, and deallocations, we need to update the allocation judgements that we presented in Figure 3.13 in order to reflect the underlying machine costs. These updated judgements are presented in Figure 3.17 where $R; v \downarrow^\delta o@R'$ is read as "value v is allocated at offset o in region R and has cost $\delta \in \mathbb{N}$," and $\mathcal{S}; \ell \uparrow^\delta v$ is read "looking up location ℓ in the region stack $\mathcal{S}$ has value v and cost $\delta \in \mathbb{N}$." Furthermore, in order lessen the amount of case analysis of lookups in the semantics, we extend the lookup judgement to work over values $\mathcal{S}; v \uparrow^\delta v$ which when given a non-location value simply returns that value.

$$\text{ExpandW}\frac{\begin{array}{c} n = max(\mathsf{allocSize}, |v| + \mathsf{ap} - |R|) \\ |v| + \mathsf{ap} > |R| \quad o = \mathsf{ap} \quad \delta = \Delta^s(n) \end{array}}{R; v \downarrow^{\delta} o@(\mathsf{ap} + |v| + 1, s + n, \mathsf{refcnt}, \nu[o \mapsto v])}$$

$$\text{NoExpandW}\frac{|v| + \mathsf{ap} \leq |R| \quad o = \mathsf{ap}}{R; v \downarrow^{1} o@(\mathsf{ap} + |v| + 1, s, \mathsf{refcnt}, \nu[o \mapsto v])}$$

$$\text{Lookup}\ell\frac{\begin{array}{c} \ell = (r, o) \quad \mathcal{S}(r) = (\mathsf{ap}, _, _, _, \nu) \\ o < \mathsf{ap} \quad o \in \mathsf{dom}(\nu) \quad v = \nu(o) \end{array}}{\mathcal{S}; \ell \uparrow^{1} v} \qquad \text{Lookup}v\frac{v \neq \ell}{\mathcal{S}; v \uparrow^{0} v}$$

Figure 3.17: Allocation and Reading Costs

It is worth mentioning that since the costs that we determine in the semantics are based on the costs that we associate with these allocation judgements—and which themselves are derived from the abstract machine model—if we wished to use a different machine model to determine costs we would do so by updating the costs associated with these allocation judgements. In fact, we will see this in Chapter 5 where we will need to update these allocation judgements to take into account the characteristics of the abstract machine model for the GPU. [6]

3.6.2 Bounded Evolutions of the Region Stack

In order to discuss the evolution of the region stack over time in the semantics, we introduce the notion of a *bounded evolution* of the region stack. This is meant to represent the evolution in time of a previous region stack $\mathcal{S}$, such that any program that would be valid in $\mathcal{S}$ is also valid in any bounded evolution $\mathcal{S}_1$ of $\mathcal{S}$. Thus, this evolution can be viewed as being bounded by the scoping boundaries of the `letregion` forms.

[6] However, in order to do this we will need to introduce a local store to the semantics, and allocation judgements. This would likewise be needed if we were to use something like the I/O or Ideal-Cache model for the underlying abstract machine model for the CPU.

Definition 3.6.1 (Bounded Evolution of Region Stack). *Given two regions stacks $\mathcal{S}_1$ and $\mathcal{S}_2$, we say that $\mathcal{S}_2$* is a bounded evolution of $\mathcal{S}_1$, *or $\mathcal{S}_1 \rightsquigarrow \mathcal{S}_2$ if both of the following hold:*

1. $\mathsf{dom}(\mathcal{S}_1) \subseteq \mathsf{dom}(\mathcal{S}_2)$
2. *Let* $(_, s_1, _, _) = \mathcal{S}_1(r)$ *and* $(_, s_2, _, _) = \mathcal{S}_2(r)$*, then* $s_2 \geq s_1$ *for all* $r \in \mathsf{dom}(\mathcal{S}_1)$.

It is worth noting that this definition only works due to the immutability of regions—the allocation rules only increment the size field of a region until that region is deallocated, at which point the region variable will no longer be in the domain of the region stack. It is easy to see that $\rightsquigarrow$ is a reflexive, antisymmetric, transitive relation.

Now given two region stacks $\mathcal{S}_1$ and $\mathcal{S}_2$ such that either $\mathcal{S}_1 \rightsquigarrow \mathcal{S}_2$ or $\mathcal{S}_2 \rightsquigarrow \mathcal{S}_1$ we define their join $\mathcal{S}_1 \diamond \mathcal{S}_2$ to be given by the following rules:

$$\begin{array}{rcll} \mathcal{S}_1 \diamond \mathcal{S}_2 & = & \mathcal{S}_1 & \text{if } \mathcal{S}_2 \rightsquigarrow \mathcal{S}_1 \\ \mathcal{S}_1 \diamond \mathcal{S}_2 & = & \mathcal{S}_2 & \text{if } \mathcal{S}_1 \rightsquigarrow \mathcal{S}_2 \end{array}$$

Theorem 3.6.1 (Well Joinedness). *Given $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 \rightsquigarrow \mathcal{S}_2$ or $\mathcal{S}_2 \rightsquigarrow \mathcal{S}_1$. Then $\mathcal{S}_1 \diamond \mathcal{S}_2$ always exists.*

Proof. Straightforward case analysis on the definition of $\mathcal{S}_1 \diamond \mathcal{S}_2$ □

Theorem 3.6.2 (Invariance Under Bounded Evolutions). *Let $E; \mathcal{S}; e \Downarrow v_1; \mathcal{S}'; c$, and such that $trace(\mathcal{S}', v_1) = v$. Let $\mathcal{S} \rightsquigarrow \mathcal{S}_1$. Then $E; \mathcal{S}_1; e \Downarrow v_2; \mathcal{S}'_1; c'$, and $trace(\mathcal{S}'_1, v_2) = v$. Where $trace$ is defined as follows:*

$$\begin{array}{lcl} trace(\mathcal{S}, i) & = & i \\ trace(\mathcal{S}, b) & = & b \\ trace(\mathcal{S}, \ \mathtt{clos}\ (x : \tau, e, E)) & = & \mathtt{clos}\ (x : \tau, e, E) \\ trace(\mathcal{S}, \ \mathtt{rclos}\ (\rho, \varphi, \varphi', u)) & = & \mathtt{rclos}\ (\rho, \varphi, \varphi', u) \\ trace(\mathcal{S}, (v_1, v_2)) & = & (trace(\mathcal{S}, v_1), trace(\mathcal{S}, v_2)) \\ trace(\mathcal{S}, [v_1, \ldots, v_n]) & = & [trace(\mathcal{S}, v_1), \ldots, trace(\mathcal{S}, v_n)] \\ trace(\mathcal{S}, \ell) & = & trace(\mathcal{S}, \mathcal{S}(\ell)) \end{array}$$

Proof. See Appendix B.3 for proof. □

Note that while we are guaranteed to evaluate to the same result v under both region stacks, we *are not* guaranteed to maintain the same cost. This is due to the fact that even though we perform the same allocations in each instantiation of the region stack, one might need to be resized during an allocation where the other may not.

3.6.3 Semantic Rules

The majority of the rules for the language are quite similar to some of the rules that we saw in Chapter 2 except for the parallel, and region creation rules, so we'll focus on explaining those in this section and leave the rest of the reasoning about the simpler rules to the reader.

Region Creation Creating regions in Λ_{C} is done via a `letregion` expression. Therefore when we encounter such a form during the evaluation of a program we first create a new region of backing memory of size $\mathsf{allocSize}$ via the *newRegion* form and say that creating this region has cost δ_R determined by the function $\Delta^a(|R|)$. We then reduce the body of the `letregion` form with the the region variable ρ mapping to the region name r in E, which then maps to the underlying (backing memory) region in the region stack $\mathcal{S}$. Once we have finished evaluating the body e of the `letregion` form we then remove the region that was allocated.

The functon $\Delta^a : \mathbb{N} \to \mathbb{N}$ determines the cost of allocating a new region. However, how we determine this function presents an interesting problem: since the underlying abstract machine model has no sense of "bulk allocation" like what is being performed here we must instead turn to other means to determine this function. Recent work by Das and Hoffmann [68] has shown that we can infer such costs using machine learning techniques such as linear regression, robust regression, and least squares on an exemplary set of programs.

$$\frac{x \in \mathsf{dom}(E)}{E; \mathcal{S}; x \Downarrow E(x); \mathcal{S}; [\alpha]} \ (\alpha \text{ fresh}) \qquad \frac{}{E; \mathcal{S}; i \Downarrow i; \mathcal{S}; [\alpha]} \ (\alpha \text{ fresh})$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow v_1; \mathcal{S}_1; c_1 \quad \mathcal{S}_1; v_1 \uparrow^{\delta_1} i_1 \\ E; \mathcal{S}_1; e_2 \Downarrow v_2; \mathcal{S}_2; c_2 \quad \mathcal{S}_2; v_2 \uparrow^{\delta_2} i_2 \\ b = i_1 < i_2 \end{array}}{E; \mathcal{S}; e_1 < e_2 \Downarrow b; \mathcal{S}_2; c_1 \oplus_{\delta_1} [\alpha_1] \oplus c_2 \oplus_{\delta_2} [\alpha_2]} \ (\alpha_1, \alpha_2 \text{ fresh})$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow v_1; \mathcal{S}_1; c_1 \quad \mathcal{S}_1; v_1 \uparrow^{\delta_1} i_1 \\ E; \mathcal{S}_1; e_2 \Downarrow v_2; \mathcal{S}_2; c_2 \quad \mathcal{S}_2; v_2 \uparrow^{\delta_2} i_2 \end{array}}{E; \mathcal{S}; e_1 * e_2 \Downarrow i_1 * i_2; \mathcal{S}_2; c_1 \oplus_{\delta_1} [\alpha_1] \oplus c_2 \oplus_{\delta_2} [\alpha_2] \oplus [\alpha_3]} \ (\alpha_1, \alpha_2, \alpha_3 \text{ fresh})$$

$$\frac{}{E; \mathcal{S}; \texttt{True} \Downarrow \texttt{True}; \mathcal{S}; [\alpha]} \ (\alpha \text{ fresh}) \qquad \frac{}{E; \mathcal{S}; \texttt{False} \Downarrow \texttt{False}; \mathcal{S}; [\alpha]} \ (\alpha \text{ fresh})$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow b; \mathcal{S}'; c_1 \quad \mathcal{S}'; b \uparrow^{\delta} \texttt{True} \\ E; \mathcal{S}'; e_2 \Downarrow v; \mathcal{S}''; c_2 \end{array}}{E; \mathcal{S}; \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v; \mathcal{S}''; c_1 \oplus_{\delta} c_2}$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow b; \mathcal{S}'; c_1 \quad \mathcal{S}'; b \uparrow^{\delta} \texttt{False} \\ E; \mathcal{S}'; e_3 \Downarrow v; \mathcal{S}''; c_2 \end{array}}{E; \mathcal{S}; \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v; \mathcal{S}''; c_1 \oplus_{\delta} c_2}$$

$$\frac{\begin{array}{c} r = E(\rho) \quad R = \mathcal{S}(r) \quad R; \texttt{clos}\ (x : \tau, e, E) \downarrow^{\delta} o@R' \\ \mathcal{S}' = \mathcal{S}[R'/R] \quad \ell = (r, o) \end{array}}{E; \mathcal{S}; \lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}'; [\alpha_1] \oplus_{\delta} [\alpha_2]} \ (\alpha_1, \alpha_2 \text{ fresh})$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow \ell; \mathcal{S}_1; c_1 \quad \mathcal{S}_1; \ell \uparrow^{\delta} \texttt{clos}\ (x : \tau, e, E) \\ E; \mathcal{S}_1; e_2 \Downarrow v_1; \mathcal{S}_2; c_2 \quad E[x \mapsto v_1]; \mathcal{S}_2; e \Downarrow v_2; \mathcal{S}_3; c_3 \end{array}}{E; \mathcal{S}; e_1\ e_2 \Downarrow v_2; \mathcal{S}_3; c_1 \oplus_{\delta} c_2 \oplus c_3}$$

$$\frac{\begin{array}{c} E; \mathcal{S}; e_1 \Downarrow v_1; \mathcal{S}_1; c_1 \qquad E; \mathcal{S}_1; e_2 \Downarrow v_2; \mathcal{S}_2; c_2 \\ r = E(\rho) \quad R = \mathcal{S}_2(r) \quad R; (v_1, v_2) \downarrow^{\delta} o@R' \quad \mathcal{S}_3 = \mathcal{S}_2[R'/R] \quad \ell = (r, o) \end{array}}{E; \mathcal{S}; (e_1, e_2)\ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}_3; c_1 \oplus c_2 \oplus_{\delta} [\alpha]} \ (\alpha \text{ fresh})$$

$$\frac{E; \mathcal{S}; e \Downarrow \ell; \mathcal{S}'; c_1 \quad \mathcal{S}'; \ell \uparrow^{\delta} (v_1, v_2)}{E; \mathcal{S}; \texttt{fst}\ e \Downarrow v_1; \mathcal{S}'; c_1 \oplus_{\delta} [\alpha]} \ (\alpha \text{ fresh})$$

$$\frac{E; \mathcal{S}; e \Downarrow \ell; \mathcal{S}'; c_1 \quad \mathcal{S}'; \ell \uparrow^{\delta} (v_1, v_2)}{E; \mathcal{S}; \texttt{snd}\ e \Downarrow v_2; \mathcal{S}'; c_1 \oplus_{\delta} [\alpha]} \ (\alpha \text{ fresh})$$

Figure 3.18: Cost semantics for the serial portion of Λ_{C}

$$\frac{r = E(\rho) \quad R = \mathcal{S}(r) \quad R;\ \texttt{rclos}\ (\rho, \varphi, \varphi', u) \downarrow^{\delta} o@R' \qquad \mathcal{S}' = \mathcal{S}[R'/R] \quad \ell = (r, o)}{E; \mathcal{S}; \lambda\rho \geq \varphi.^{\varphi'} u\ \ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}'; \alpha_1 \oplus_{\delta} [\alpha_2]}\ (\alpha_1, \alpha_2\ \text{fresh})$$

$$\frac{E; \mathcal{S}; e \Downarrow \ell; \mathcal{S}'; c_1 \quad \mathcal{S}'; \ell \uparrow^{\delta}\ \texttt{rclos}\ (\rho', \varphi, \varphi', u) \qquad r = E(\rho) \quad E[\rho' \mapsto r]; \mathcal{S}'; u \Downarrow v; \mathcal{S}''; c_2}{E; \mathcal{S}; e[\rho] \Downarrow v; \mathcal{S}''; c_1 \oplus_{\delta} c_2}$$

$$\frac{r = E(\rho) \quad R = \mathcal{S}(r) \quad R;\ \texttt{clos}\ (f : \tau, u, E[f \mapsto (r, o)]) \downarrow^{\delta} o@R' \qquad \mathcal{S}' = \mathcal{S}[R'/R] \quad E[f \mapsto (r, o)]; \mathcal{S}'; u \Downarrow v'; \mathcal{S}''; c}{E; \mathcal{S};\ \texttt{fix}\ f : (\mu, \rho).u \Downarrow v'; \mathcal{S}''; [\alpha_1] \oplus_{\delta} c}\ (\alpha\ \text{fresh})$$

$$\frac{\mathcal{S} \rightsquigarrow \mathcal{S}_1 \quad \mathcal{S} \rightsquigarrow \mathcal{S}_3 \qquad E; \mathcal{S}_1; e_1 \Downarrow v_1; \mathcal{S}_2; c_1 \quad E; \mathcal{S}_3; e_2 \Downarrow v_2; \mathcal{S}_4; c_2 \qquad \mathcal{S}' = \mathcal{S}_2 \diamond \mathcal{S}_4 \quad r = E(\rho) \quad R = \mathcal{S}'(r) \qquad R; (v_1, v_2) \downarrow^{\delta_a} o@R' \quad \mathcal{S}'' = \mathcal{S}'[R'/R] \quad \ell = (r, o)}{E; \mathcal{S}; e_1 \| e_2\ \ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}''; (c_1 \otimes c_2) \oplus [\alpha_1] \oplus_{\delta_a} [\alpha_2]}\ (\alpha_1, \alpha_2\ \text{fresh})$$

$$\frac{E; \mathcal{S}_{i-1}; e_i \Downarrow v_i; \mathcal{S}_i; c_i \quad r = E(\rho) \quad R = \mathcal{S}_n(r) \qquad R; [v_1, \ldots, v_n] \downarrow^{\delta} o@R' \quad \ell = (r, o) \quad \mathcal{S}_n' = \mathcal{S}_n[R'/R]}{E; \mathcal{S}_0; [e_1, \ldots, e_n]\ \ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}_n'; \left(\bigoplus_i c_i\right) \oplus_{\delta} [\alpha]}\ (\alpha\ \text{fresh})$$

$$\frac{E; \mathcal{S}; f\ e_i \Downarrow v_i; \mathcal{S}'; c_i \quad r = E(\rho) \quad R = \mathcal{S}'(r) \qquad R; [v_1, \ldots, v_n] \downarrow^{\delta_a} o@R' \quad \mathcal{S}'' = \mathcal{S}'[R'/R]}{E; \mathcal{S};\ \texttt{parmap}\ f\ [e_1, \ldots, e_n]\ \ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}''; \left(\bigotimes_i c_i\right) \oplus [\alpha_1] \oplus_{\delta_a} [\alpha_2]}\ (\alpha_1, \alpha_2\ \text{fresh})$$

$$\frac{R = \mathit{newRegion}(\mathsf{allocSize}) \quad r\ \text{fresh} \quad r \notin \mathsf{dom}(\mathcal{S}) \quad \delta_R = \Delta^a(|R|) \qquad E[\rho \mapsto r]; \mathcal{S}[r \mapsto R]; e \Downarrow v; \mathcal{S}'; c \quad \mathcal{S}'' = \mathcal{S}' \setminus r}{E; \mathcal{S};\ \texttt{letregion}\ \rho\ \ \texttt{in}\ e \Downarrow v; \mathcal{S}''; [\alpha_1] \oplus_{\delta_R} [\alpha_2] \oplus c \oplus [\alpha_3]}\ (\alpha_1, \alpha_2, \alpha_3\ \text{fresh})$$

Figure 3.19: Cost Semantics for Λ_{C} (Cont.)

One might now rightly wonder why we don't do this for the other numeric costs in our lookup/allocation rules. While we feel as though learning the numeric costs for the machine model is possible, it also presents a number of challenging problems; the language that Das and Hoffmann [68] look at is a *serial* language, and thus they do not have to worry about schedule orders or similar issues in their learning of the costs for the programs that they train on. However once we add in parallelism, the possible search space increases dramatically, and we therefore leave this to future investigation.

Even in the presence of parallelism, we will still need to be able to determine this cost function. And furthermore, we will need a couple other cost-functions in Chapter 5 in order to determine the cost of transferring computation from one computational device to another. We therefore defer detailing the determination of just how we calculate these *Primitive Cost Providers* (PCPs) and get around the search-space explosion due to the introduction of parallelism until then. The key observation to keep in mind until then however is that all of these PCPs represent rather large—and often serial—processes which makes the learning of these significantly easier in the parallel setting.

Region Abstraction & Instantiation There is an important stratification between the different levels of representation of region variables that makes itself apparent in the operational semantics. In the source language, we can only reference regions by their region variable ρ, and the region stack $\mathcal{S}$ maps region names r to regions in memory R. This separation between the *region variable* for a region, and the *region name* for a region arises from the need to be able to represent a many-to-one mapping of region variable (in the source language) to a singular region in memory due to region abstraction and instantiation. By separating the region names in the region stack from the region variables in the language, we are easily able to

implement this mapping without changing the region stack; we represent region instantiation and abstraction within the operational semantics by updating the environment with a new mapping $\rho \mapsto r$ for evaluation for the body of the expression. We can see this at work in the following rule for region instantiation:

$$\frac{\begin{array}{cc} E; \mathcal{S}; e \Downarrow v; \mathcal{S}'; c_1 & \mathcal{S}'; v \uparrow^{\delta} \texttt{rclos}\,(\rho', \varphi, \varphi', u) \\ r = E(\rho) & E[\rho' \mapsto r]; \mathcal{S}'; u \Downarrow v'; \mathcal{S}''; c_2 \end{array}}{E; \mathcal{S}; e[\rho] \Downarrow v'; \mathcal{S}''; [\alpha] \oplus c_1 \oplus c_2} \ (\alpha \text{ fresh})$$

Expressing Parallelism in the Semantics It is important to note that the semantics shares a global view of the region stack—thus when expressions are evaluated in parallel they are both reading and writing to a shared piece of memory. We can see this being expressed in the semantics in the parallel rules e.g.

$$\frac{\begin{array}{ccc} \mathcal{S} \rightsquigarrow \mathcal{S}_1 & & \mathcal{S} \rightsquigarrow \mathcal{S}_3 \\ E; \mathcal{S}_1; e_1 \Downarrow v_1; \mathcal{S}_2; c_1 & & E; \mathcal{S}_3; e_2 \Downarrow v_2; \mathcal{S}_4; c_2 \\ \mathcal{S}' = \mathcal{S}_2 \diamond \mathcal{S}_4 & r = E(\rho) & R = \mathcal{S}'(r) \\ R; (v_1, v_2) \downarrow^{\delta_a} o@R'; \sigma_1 & \mathcal{S}'' = \mathcal{S}'[R'/R] & \ell = (r, o) \end{array}}{E; \mathcal{S}; e_1 \| e_2 \ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}''; (c_1 \otimes c_2) \oplus [\alpha_1] \oplus_{\delta_a} [\alpha_2]} \ (\alpha_1, \alpha_2 \text{ fresh}) \tag{3.3}$$

where both of the evaluations in red are evaluated in parallel. This, as compared to the semantics for the rule for pairs

$$\frac{\begin{array}{ccccc} & E; \mathcal{S}; e_1 \Downarrow v_1; \mathcal{S}_1; c_1 & & E; \mathcal{S}_1; e_2 \Downarrow v_2; \mathcal{S}_2; c_2 & \\ r = E(\rho) & R = \mathcal{S}_2(r) & R; (v_1, v_2) \downarrow^{\delta} o@R' & \mathcal{S}_3 = \mathcal{S}_2[R'/R] & \ell = (r, o) \end{array}}{E; \mathcal{S}; (e_1, e_2)\ \texttt{at}\ \rho \Downarrow \ell; \mathcal{S}_3; c_1 \oplus c_2 \oplus_{\delta} [\alpha]} \ (\alpha \text{ fresh})$$

where the first evaluation of e_1 in blue evaluates to a region stack $\mathcal{S}_1$, and *then* the evaluation (in green) of the expression e_2 happens *after* the evaluation of e_1 in region stack $\mathcal{S}_1$ and results in a region stack $\mathcal{S}_2$.

Stating that e_1 and e_2 run in parallel does not necessarily mean that they both start and finish at the same time (in fact this would be rare). We therefore need to account for the fact that both the start and end of the evaluations of e_1 and e_2 in Equation (3.3) can be interleaved, non-simultaneous, or even serial as they relate to the region stack.

In order to account for these complexities we use bounding evolutions on the starting region stacks for expressions that are meant to be evaluated in parallel; doing so allows us to express the uncertainty of whether or not they both start evaluation at $\mathcal{S}$, while also allowing us to express the fact that the starting region stacks for both e_1 and e_2 evolved from $\mathcal{S}$. However, this then gives us two separate "handles" to the region stack in the red section of the rule, and we need to be able to determine the final states for the region stack after these parallel reductions have taken place. In order to determine this we need to be able to take the join of these two region stacks. However in order to do so we first need to be able to show that either $\mathcal{S}_2 \rightsquigarrow \mathcal{S}_4$ or $\mathcal{S}_4 \rightsquigarrow \mathcal{S}_2$ in order to apply Theorem 3.6.1 to get that $\mathcal{S}_2 \diamond \mathcal{S}_4$ exists.

Proving that the join exists for region stacks that have evolved in parallel rests on the fact that regions are lexically scoped and that the definition of $\Downarrow$ respects these region scoping rules. The following theorem proves exactly this and, coupled with transitivity of $\rightsquigarrow$, shows that Theorem 3.6.1 can be applied to $\mathcal{S}_2$ and $\mathcal{S}_4$ in Equation (3.3), and therefore the final region can always be determined after a series of parallel steps by taking the join of the ending stack states.

Theorem 3.6.3 (Boundedness of $\Downarrow$)**.** *Let* $E; \mathcal{S}; e \Downarrow v; \mathcal{S}'; c$*. Then* $\mathcal{S} \rightsquigarrow \mathcal{S}'$*.*

Proof. See Appendix B.1. □

3.6.4 Schedules

Now that we have defined the operational semantics for Λ_C and the building of cost graphs, we now turn to factoring in how different scheduling policies can affect which nodes in the cost graph become active and when, this in turn can have a large impact on the space and time usage of the program.

In order to parameterize cost graphs by the scheduling policy that is chosen, we use a similar technique to that of Spoonhower et al. [6] where a *schedule order* $\trianglelefteq$ is

defined over the nodes of the cost graph. However, since the weights of the edges in the cost graphs for Λ_{C} represent the computational cost of traversing that edge, a higher-weighted edge will cause the thread that is assigned the source node to not be schedulable until the number of steps taken since that node has been scheduled has passed the weight of the outgoing edge of that node. Thusly, the next node will not become schedulable until the number of 'ticks' represented by that edge have been performed.

With this in mind, we update the definition of the schedule order that we will be using in order to take into account the weighted edges. This updated schedule order, the *weighted schedule order* $\overline{V^\star}; \alpha_1 \trianglelefteq_{\mathsf{w}} \alpha_2$ adds a set of previous *step contexts* $\overline{V^\star}$. [7] Each step context $V^\star$ in $\overline{V^\star}$ corresponds to a set of simultaneously scheduled nodes V for the schedule, where each node in the step is coupled with the step number(s) in the future at which the thread assigned to that node will finish and become usable again e.g. when allocating memory for a region we lose that thread until the allocation has been performed.[8] Given a step V_k we define the step contextualization $V_k^\star$ of it as follows:

$$\left\{(\alpha, \delta + i) \,\middle|\, \alpha \in V_k, \exists \alpha' \,.\, ((\alpha', \alpha, \delta) \in E) \wedge \alpha' \in V_i \wedge i < k \right\} \tag{3.4}$$

Note the $V^\star$ can, in a sense, be a multiset—we can have a single node mapping to a number of different times since this tracks the cost of reaching the node from its parents that have been scheduled.

Just as before in Section 2.1.1 with schedule orders, a weighted schedule order has the following properties:

Proposition 3.6.1 (Properties of Weighted Schedule Order)**.** *Let $\trianglelefteq_{\mathsf{w}}$ be a weighted schedule order over a cost graph G. Then the following properties hold:*

[7] The schedule order that we saw previously in Section 2.1.1 can be seen as a special case of the weighted schedule order in the case that all the edges have weight 1.

[8] We will assume that step contexts are well formed.

- $\forall \alpha \in G \,.\, \forall \overline{V^\star} \,.\, \overline{V^\star}; \alpha \trianglelefteq_{\mathsf{w}} \alpha$*;*
- $\forall \alpha_1, \alpha_2 \in G \,..\, \forall \overline{V^\star} \,.\, \alpha_1 \prec_g \alpha_2 \Rightarrow \overline{V^\star}; \alpha_1 \triangleleft \alpha_2$*, where* $\prec_g$ *means that* α_1 *is a parent of* α_2*.*

And we can extend this to sets of nodes:

Definition 3.6.2 (Schedule Order for Node Sets)**.**

$$\overline{V^\star}; V_i \trianglelefteq_{\mathsf{w}} V_{i+1} \iff \forall \alpha_1 \in V_1, \alpha_2 \in V_2 \,.\, \overline{V^\star}; \alpha_1 \triangleleft \alpha_2,$$

With the side-condition that $\forall j \leq i \,.\, V_j^\star \in \overline{V^\star}$ *if* $i > 0$*.*

And define simultaneously scheduled nodes and and steps in just the same manner:

Definition 3.6.3 (Simultaneously Scheduled Nodes)**.**

$$\overline{V^\star}; \alpha_1 \bowtie \alpha_2 \iff \overline{V^\star}; \alpha_1 \trianglelefteq_{\mathsf{w}} \alpha_2 \wedge \overline{V^\star}; \alpha_2 \trianglelefteq_{\mathsf{w}} \alpha_1$$

We then use this to partition the graph into sets of simultaneously scheduled nodes or *steps*:

$$\mathsf{steps}(\trianglelefteq_{\mathsf{w}}) = V_1, \ldots, V_k \;\text{ such that }\; \begin{cases} \exists i \,.\, \forall \alpha \in G \,.\, \exists V_i . \alpha \in V_i, \\ \{\}; V_1 \triangleleft V_2, \\ \quad\vdots \\ \{V_1^\star, \ldots, V_{k-1}^\star\}; V_{k-1} \triangleleft V_k, \\ \forall i \,.\, \forall V_i \,.\, \forall \alpha_1, \alpha_2 \in V_i \,.\, \{V_1^\star, \ldots, V_i^\star\}; \alpha_1 \bowtie \alpha_2 \end{cases} \tag{3.5}$$

From this, we can then define the *closure* $\widehat{V_i}$ of the step V_i as

$$\widehat{V_i} = \bigcup_{k=1}^{i} V_k \tag{3.6}$$

Thus while all the nodes in $\widehat{V_i}$ are not necessarily scheduled simultaneously, we do have that

$$\{V_1^\star, \ldots, V_i^\star\}; \widehat{V_i} \triangleleft V_{i+1}$$

With this framework, we can then think of the schedule as defining a wavefront that progresses across the graph (where each "wave" is the next step V_i).

Note that with this definition it is possible to get *empty* steps. In fact, we will use these to our advantage momentarily when calculating the cost for a traversal through the graph. An empty step V_i will be generated when there are nodes left to schedule, yet no nodes schedulable (at the current step):

$$\exists \alpha \in G \,.\, \alpha \notin \widehat{V_{i-1}} \wedge \forall \alpha' \in \widehat{V_{i-1}} \,.\, \{V_1^\star, \ldots, V_{i-1}^\star\}; \alpha' \ntrianglelefteq_{\mathsf{w}} \alpha \tag{3.7}$$

Or in other words, when the node is not able to be scheduled at the current step, but will later become schedulable.

Weighting Steps Since the cost graphs of Λ_{C} are weighted, and since threads can continue running from one step to another, calculating the cost of each step is not entirely straightforward but still rather easy: since another step will start as soon as the *minimum* weight edge from one step to another is finished, the cost of a step is the minimum weight edge between each step, and 1 in the case that the step is empty:

$$\mathsf{step}_w(V_i) = \begin{cases} 0 & i = 1 \\ 1 & i \neq 1, V_i = \emptyset \\ \min\{\delta \mid \alpha_i \in V_i,\ \alpha_{i-1} \in V_{i-1}, (\alpha_{i-1}, \alpha_i, \delta) \in E\} & \text{otherwise} \end{cases} \tag{3.8}$$

and with this the cost of a set of we can define the cost of the computation thus far i.e., the cost of the computation for a given set of steps through the cost graph as follows:

$$\mathsf{step}_T(\overline{V}) = \sum_{V_i \in \overline{V}} \mathsf{step}_w(V_i) \tag{3.9}$$

This will then prove useful for determining when a thread can become runnable again.

Thread Availability In a system with a weighted schedule we lose the invariant that we will always have p threads in a p-thread system; since threads can be utilized for more than one step we will not necessarily always have p threads at each step. We therefore need a way to determine the number of threads available at a given step k in the schedule defined over the graph. We thus arrive at the following definition that can be used in the schedule order to determine how many threads (in a p-thread system) are available based upon the current step context $\overline{V^\star} = \{V_1^\star, \ldots, V_k^\star\}$

$$\mathsf{threads}(\overline{V^\star}) = p - \left|\left\{(\alpha, l) \mid (\alpha, l) \in V_i^\star \in \overline{V^\star}, l > \mathsf{step}_T(\{V_i, \ldots, V_k\}) + 1\right\}\right| \tag{3.10}$$

As an example of a weighted schedule order and how it differs from the schedule order we saw previously, we can define the greedy scheduling policy defined in Equation (2.14) as a weighted schedule order as follows:

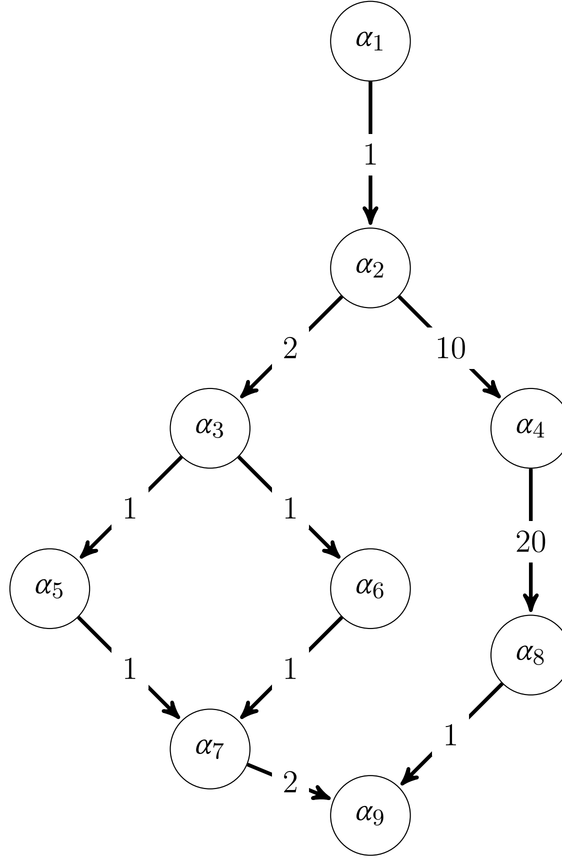
$$\forall \alpha_1, \alpha_2 \,.\, \overline{V^\star}; \alpha_1 \trianglelefteq_{\mathsf{w}} \alpha_2 \Rightarrow \begin{array}{l} \exists \alpha_3 \,.\, \left((\alpha_3 \prec_g \alpha_2) \wedge (\alpha_3, j) \in V_i^\star \in \overline{V^\star} \wedge k < j\right) \\ \vee \left|\{\alpha \mid \overline{V^\star}; \alpha \trianglelefteq_{\mathsf{w}} \alpha_1\}\right| = \mathsf{threads}(\overline{V^\star}) \end{array} \tag{3.11}$$

Where we are saying that a node α_1 is scheduled before α_2 if there exists a parent (α_3) of α_2 that still needs to complete (the greyed area), or if the set of nodes that have been scheduled with α_1 have taken up all of the active threads (the non-greyed section). If we take the cost graph to be unweighted—all of the weights in the graph to be 1—we get that $(\alpha_3, i+1) \in V_i^\star$ for any α_3, and therefore it has always finished by the time we come around—and thus only poses a problem if α_3 and α_1 are scheduled simultaneously. In this case, we arrive back at the definition of a greedy schedule in Equation (2.14):

$$\begin{array}{c} \forall \alpha_1, \alpha_2 \,.\, \overline{V^\star}; \alpha_1 \trianglelefteq_{\mathsf{w}} \alpha_2 \Rightarrow \\ \exists \alpha_3 \,.\, \left((\alpha_3 \prec_g \alpha_2) \wedge (\alpha_3, i+1) \in V_i^\star \in \overline{V^\star} \wedge k < i+1\right) \vee \left|\{\alpha \mid \overline{V^\star}; \alpha \trianglelefteq_{\mathsf{w}} \alpha_1\}\right| = \mathsf{threads}(\overline{V^\star}) \\ \Longleftrightarrow \\ \exists \alpha_3 \,.\, \left((\alpha_3 \prec_g \alpha_2) \wedge \alpha_3 \bowtie \alpha_1\right) \vee \left|\{\alpha \mid \overline{V^\star}; \alpha \trianglelefteq_{\mathsf{w}} \alpha_1\}\right| = p \end{array}$$

Since we have that for all $(\alpha, k) \in V_i^\star \in \overline{V^\star}$ that $k = i + 1$, and therefore that $\mathsf{threads}(\overline{V^\star}) = p$ for all $\overline{V^\star}$.

As an example, of how these weighted schedules work, take the following cost graph:



Then we have for $p \geq 3$ that under the scheduling policy in Equation (3.11), we will get the following steps:

$$\{\alpha_1\}, \{\alpha_2\}, \{\alpha_3, \alpha_4\}, \{\alpha_5, \alpha_6\}, \{\alpha_7\}, \{\}^6, \{\alpha_8\}, \{\}^{19}, \{\alpha_9\}$$

where the superscripts denote "empty steps" in which no new nodes can be scheduled.

Now that we have specified the representation of schedules across the cost graph, we can now turn to specifying how we can determine costs.

3.6.5 Determining Costs

In order to determine the space usage of a program, we utilize techniques similar to that of Spoonhower et al. [6] (and reviewed in Section 2.1.1). However instead

of keeping a heap graph and having a reachability relation on this graph to track the locations that need to be kept alive, we instead look at the region stack that we maintain and build throughout the graph traversal. Moreover, since the operations on the region stack are explicit nodes in the cost graph, the region stack at any given point is completely determined by the closure of the current computational step within the graph. We thus define $\mathcal{S}(V_i, \trianglelefteq_{\mathsf{w}})$ to be the region stack at step i under schedule $\trianglelefteq_{\mathsf{w}}$.

With this definition of a region stack at step i, we then define the space usage at a given step in the computation as the total space used by the locations in the region stack at that step: (i.e., for a given set of vertices in the cost graph):

$$\mathsf{space}(V, \trianglelefteq_{\mathsf{w}}) = |\{\ell \in R \mid R \in \mathcal{S}(V, \trianglelefteq_{\mathsf{w}})\}| = \sum_{R \in \mathcal{S}(V, \trianglelefteq_{\mathsf{w}})} |\ R.v\ | \tag{3.12}$$

We can then say that the space required by a schedule $\trianglelefteq_{\mathsf{w}}$ that computes value v at location ℓ is the maximum space usage required by any step in the schedule over the DAG formed by the cost semantics:

$$\mathsf{space}_v(\trianglelefteq_{\mathsf{w}}) = \max \{\mathsf{space}(V_i, \trianglelefteq_{\mathsf{w}}) \mid V_i \in \mathsf{steps}(\trianglelefteq_{\mathsf{w}})\} \tag{3.13}$$

Determining time costs for a program is relatively straightforward, since we have the $\mathsf{step}_w(V_i)$ function: the total amount of time taken by a program e under schedule $\trianglelefteq_{\mathsf{w}}$ with cost graph $G = (\alpha_1, \alpha_2, V, E)$ is the sum of the time taken for each step given by $\trianglelefteq_{\mathsf{w}}$ through G, or in other words

$$\mathsf{time}(e, \trianglelefteq_{\mathsf{w}}) = \mathsf{step}_T(\mathsf{steps}(\trianglelefteq_{\mathsf{w}})) \tag{3.14}$$

where step_w and step_T were defined in Equations (3.8) and (3.9).

A Brief Aside: Region Lifetime Preservation

Now that the semantics has been developed, we can present the following theorem that details how the type-level outlives relation is preserved at the term- or memory-level by the ancestor relationship (and vice-versa), and by the semantics.

Theorem 3.6.4. *Let $\rho, \rho' \in \mathsf{RVars}$, and e an expressions such that $\{\underline{\rho}\}; \cdot; \cdot \vdash_{exp} e : \tau, \varphi$, and $\rho, \rho' \in \varphi$. Now take the following derivation tree for e:*

$$\cfrac{\cfrac{\cfrac{\vdots}{E; \mathcal{S}_1; e' \Downarrow v'; \mathcal{S}_2; c}\ (\star) \qquad \vdots}{\vdots}}{\{\underline{\rho} \mapsto r_h\}; \{r_h \mapsto \underline{R}\}; e \Downarrow v; \mathcal{S}'; c'} \tag{3.15}$$

such that $\rho, \rho' \in \mathsf{frv}\,(e')$, and $\mathcal{S}_1(E(\rho)) = R_1$, $\mathcal{S}_1(E(\rho')) = R_2$. Then we have that $\mathcal{S}_1 \vdash R_1 :\geq R_2$ iff $\Omega; \Delta \vdash_{rr} \rho \geq \rho'$. Where Ω and Δ are constructed by the following derivation:

$$\cfrac{\cfrac{\cfrac{\vdots}{\Omega; \Delta; \Gamma \vdash_{exp} e' : \tau', \varphi'}\ (\star\star) \qquad \vdots}{\vdots}}{\{\underline{\rho}\}; \cdot; \cdot \vdash_{exp} e : \tau, \varphi} \tag{3.16}$$

Furthermore, we have for E and $\mathcal{S}_1$ as in $(\star)$ and Ω as in $(\star\star)$ that

$$proj(E); \mathcal{S}_1 \vdash_{vreg} \Omega$$

Proof. The proof follows by induction on the structure of the region environment, and region stacks, and by inspection of the the typing rules and operational semantics. □

4

Cost Semantics for GPU Parallelism

Contents

4.1 Introduction **101**
4.2 Restricting Λ_C for the GPU **102**
4.3 The GPU, Abstractly **104**
4.3.1 GPU Memory Model 104
4.3.2 Abstract Machine Model for the GPU 105
4.3.3 The Discrete Memory Machine Model 106
4.3.4 The Unified Memory Machine Model 108
4.3.5 Putting it all Together: the HMM Model 109
4.4 GPU Cost Graph Combinators **111**
4.4.1 Thread Graphs and Meta-edges 112
4.4.2 Warp Graphs 113
4.4.3 Warp Pools 114
4.4.4 Device Boxes 115
4.5 Cost Semantics for Λ^G **115**
4.5.1 Meta Locations 115
4.5.2 Reading and Writing to Memory 117
4.5.3 Thread Semantics 118
4.5.4 Global Cost Semantics 120
4.6 Scheduling GPU Cost Graphs **126**
4.6.1 Thread Schedules 126
4.6.2 Warp Schedules 130
4.6.3 Device Box Schedules 133
4.6.4 Keeping Track of the Region Stack 134

4.7 **Determining Costs from GPU Cost Graphs** 135

4.1 Introduction

Now that we have detailed a theory for profiling region-based parallel languages on the CPU, we turn our attention to defining a cost semantics for profiling region-based parallel languages on the GPU. These two theories will then serve as the constituent parts of the eventual heterogeneous parallel language in Chapter 5. We thus want the GPU part of the language—Λ^{G}—to be as similar as possible to the CPU portion of the language—Λ_{C}—however we need to restrict some of the computational forms permitted in the language that runs on the GPU from what we saw in the previous chapter.

Beyond simply restricting the language forms that can run on the GPU, due to the inherently different parallel architectures of the CPU and GPU the parallel constructs that we encounter in this chapter will have a significant difference computationally compared to the same parallel constructs (syntactically) that we saw in Λ_{C}. However, we will re-use a technique we saw earlier on to our cost semantics in order to lessen the burden when it becomes time to combine these two parts: we saw in Chapter 2 how the description of parallelism from the cost perspective could be pushed from the operational-semantic description into the cost structure that these semantics build. We do the same in this chapter for our GPU language Λ^{G} and reflect the different parallel characteristics of the GPU in the construction of, and subsequent profiling of the cost graphs while altering the operational specification of these forms as little as possible.

The rest of this chapter is structured as follows: we first define the syntax of Λ^{G} (Section 4.2), describe the machine and memory model that we will be using which

is significantly more complex than the CPU version that we saw previously (Section 4.3), introduce our GPU graph operators and combinators (Section 4.4), and define the cost semantics that utilize these combinators to build up the cost graphs that describe the underlying parallel architecture of the GPU (Section 4.5).

We then conclude the chapter in Sections 4.6 and 4.7 by defining the determination of time and space costs from GPU cost graphs. In particular, we define the primary time cost judgement for Λ^{G} in Equation (4.25) as a function parametrized by the starting state of local memory on the GPU, and discuss and define the space consumption for Λ^{G} programs in Equation (4.26).

4.2 Restricting Λ_{C} for the GPU

The core definition of Λ^{G} is given in Figure 4.1, and is almost identical syntactically to the Λ_{C} language that we introduced in Chapter 3. It consists of a standard λ-calculus along with pairs $(e_1, e_2)\ \texttt{at}\ \rho$, parallel composition $e_1 \| e_2\ \texttt{at}\ \rho$, arrays $[e_1, \ldots, e_n]\ \texttt{at}\ \rho$, and parallel traversals of these arrays $\texttt{parmap}\ f\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho$. Each λ is annotated with the set of regions that it uses, and the closure for it must be allocated in the specified region ρ—hence our closures do not contain a region annotation.

The values in Λ^{G} consist of integers i, booleans b, and heap values that can be stored in the regions R of memory. Heap values consist of arrays of values $[v_1, \ldots, v_n]$, function closures $\texttt{clos}\ (x : \tau, e, E)$, locations in memory $\ell = (r, o)$, and finally region closures $\texttt{rclos}\ (\rho, \varphi, \varphi', u)$. Locations ℓ consist of a region name r along with the offset o into that region. Region closures $\texttt{rclos}\ (\rho, \varphi, \varphi', u)$ represent region-polymorphic code and are created via the region abstraction form $\lambda\rho \geq \varphi.^{\varphi'} u\ \texttt{at}\ \rho$; these are then instantiated in much the same way as we would a function closure via the region instantiation form $e[\rho]$.

While the parallel constructs that we see here are the same syntactically as those in Chapter 3, the parallel traversal $\texttt{parmap}\ f\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho$ has a very different computational—and hence cost—interpretation on the GPU: in Λ_{C}, the parallelism introduced by `parmap` consists of fork-join parallelism for each of the e_i that is being mapped across whereas in Λ^{G} (on the GPU) this represents data parallelism across the e_i. We formalize this different parallel and cost structure on the GPU in Sections 4.4 to 4.7. Furthermore, since we do not have fork-join parallelism on the GPU it does not make sense to spark sub-kernels off for a single computation, thus the parallel composition operator $e_1 \| e_2$ does not actually introduce any parallelism on the GPU, and instead is serialized.[1]

While the syntax in Figure 4.1 is almost identical to that of Λ_{C}, there is a crucial omission of the `letregion` form from the language. The precise device properties that impose this restriction on the GPU are discussed in Section 4.3.1. However, it is worth noting now that when it becomes time in Chapter 5 to combine the CPU and GPU portions of the language, this lack of region creation operations in the GPU part of the language will create a natural subtyping relation between CPU and GPU computations that will be crucial for the language—all valid GPU computations are valid CPU computations, but not all CPU computations are valid GPU computations and further, all memory is allocated on the CPU. This will enforce a natural hierarchy on heterogeneous programs; all programs start on the CPU after which computations are then "pushed" over to the GPU and the results "pulled" back to the CPU (from the GPU).

The static semantics for Λ^{G} is the same as that for Λ_{C} with the exception that there is no `letregion` form. We therefore refer the reader to Sections 3.2 and 3.3 for the details of the type system for Λ^{G}.

[1]We could remove this form from the language but we want this language to be as close as possible (syntactically) to that of Λ_{C} so that later on we can combine them more easily.

$$i \in \mathbb{Z} \mid r \in \textsf{RNames} \mid f, x \in \textsf{Vars} \mid \rho \in \textsf{RVars} \mid o \in \mathbb{N}$$

$$\begin{array}{lcll}
\ell & ::= & (r, o) & \text{Region Locations} \\
R & ::= & (\textsf{ap}, s, \textsf{refcnt}, \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\}) & \text{Regions} \\
\mathcal{S} & ::= & \underline{\rho} \mapsto \underline{R} \mid \mathcal{S}, r \mapsto R & \text{Ordered Domain/Region Stack} \\
\varphi & ::= & \{\rho_1, \ldots, \rho_n\} & \text{Effect Sets} \\
\tau & ::= & \texttt{bool} \mid (\mu, \rho) \mid \Pi\rho \geq \varphi.^{\varphi'}\tau & \text{Types} \\
\mu & ::= & \texttt{int} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \texttt{Vec}\ \tau & \text{Boxed Types} \\
\Omega & ::= & \{\overline{\rho}\} \mid \rho, \Omega & \text{World Contexts} \\
\Delta & ::= & \cdot \mid \Delta, \rho \geq \varphi & \text{Region Constraints} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau & \text{Typing Contexts} \\
e & ::= & x \mid i \mid \texttt{True} \mid \texttt{False} & \text{Terms} \\
 & \mid & e_1 < e_2 \mid e_1 * e_2 & \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f & \\
 & \mid & \lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho \mid e_1\ e_2 & \\
 & \mid & \lambda \rho \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho \mid e[\rho] & \\
 & \mid & (e_1, e_2)\ \texttt{at}\ \rho \mid \texttt{fst}\ e \mid \texttt{snd}\ e & \\
 & \mid & \texttt{fix}\ f : \tau.u \mid e_1 \| e_2\ \texttt{at}\ \rho & \\
 & \mid & [e_1, \ldots, e_n]\ \texttt{at}\ \rho \mid \texttt{parmap}\ e\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho & \\
u & ::= & \lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho \mid \lambda \rho \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho & \text{Abstractions} \\
v & ::= & i \mid b \mid (v_1, v_2) \mid [v_1, \ldots, v_n] & \text{Heap Values} \\
 & \mid & \texttt{clos}\ (x : \tau, e, E) \mid \texttt{rclos}\ (\rho, \varphi, \varphi', u) \mid \ell &
\end{array}$$

Figure 4.1: Syntax of Λ^{G}

4.3 The GPU, Abstractly

4.3.1 GPU Memory Model

As we saw in Section 2.4, while CPU and GPU memory are similar in many ways there are crucial differences in how memory accesses take place, and where the data resides in memory affects the cost of access. In particular, in a similar vein to CPU memory there is a hierarchical memory model which consists of private work item memory, local work group memory, and global or constant memory. As opposed to cache memory on the CPU in which sections of memory are brought into or evicted from the cache based upon memory access patterns, on the GPU sections of memory that are local persist throughout the computation. Furthermore, we will

take the same view towards global memory as Harlan [9] in which global accesses are unsynchronizable, but we do have access to atomics.

Put together, these restrictions mean that we cannot resize regions on the GPU since a resize operation would require synchronization operations: resizing a region requires possibly multiple operations to happen atomically in order to avoid race conditions on global memory. This means that we are only able to allocate from the space already available within the region when it is copied over to the GPU. This is also one of the main reasons for restricting region creation to be on the CPU only. Furthermore, as mentioned earlier disallowing resizing of regions on the GPU makes the process of "pulling" and then merging the GPU region back into its parent CPU region later on in Chapter 5 possible.

On the GPU we assume regions are represented using standard memory buffers and hence reside in global memory, except for a specified set of regions that are a priori placed in local memory at the start of the computation. While this leaves open the possibility of optimizing region placement in non-global memory in order to speed-up access from within a kernel based on the work on logical regions by Bauer et al. [69] we do not address this here and leave it to future investigation—however the cost theory we present does take into account local (work group level) memory characteristics in its calculations.

4.3.2 Abstract Machine Model for the GPU

In Chapter 3 the cost semantics depended on an abstract machine model that provided the intensional aspects of the cost semantics—the base costs for memory accesses. However, whereas there are a number of different machine models for the CPU (and the memory hierarchies surrounding it), we have far fewer abstract machine models to choose from for the GPU. This is not just due to the relative newness of GPU computation however: as we saw in Section 2.4 the memory model for the

GPU has a number of interesting characteristics based upon how (and when) various sections of memory are accessed and based upon the threading model and this restricts freedom when designing abstract machines for this architecture.

In this work we will be using the Hierarchical Memory Machine (HMM) model presented by Nakano [70] to model the GPU. This model is based on two other machine models to describe the layout of the GPU as a number of Streaming Multiprocessors (SMPs) with local memory that can read and write to a slower global memory. One—the Discrete Memory Machine (DMM) [71]—is used to describe the characteristics of a single SMP. Each of these DMM models are then connected via another machine—the Unified Machine Model (UMM) [71]. These, together, abstractly represent the memory aspects of the GPU that we are interested in: the GPU is made up of a number of SMPs with local memory (modeled by the DMM) which can then interact with each other and the global memory in certain ways (modeled by the UMM).

The UMM model is based on the Discrete Memory Machine (DMM)—we will therefore first explain the DMM and then derive the UMM from the DMM.

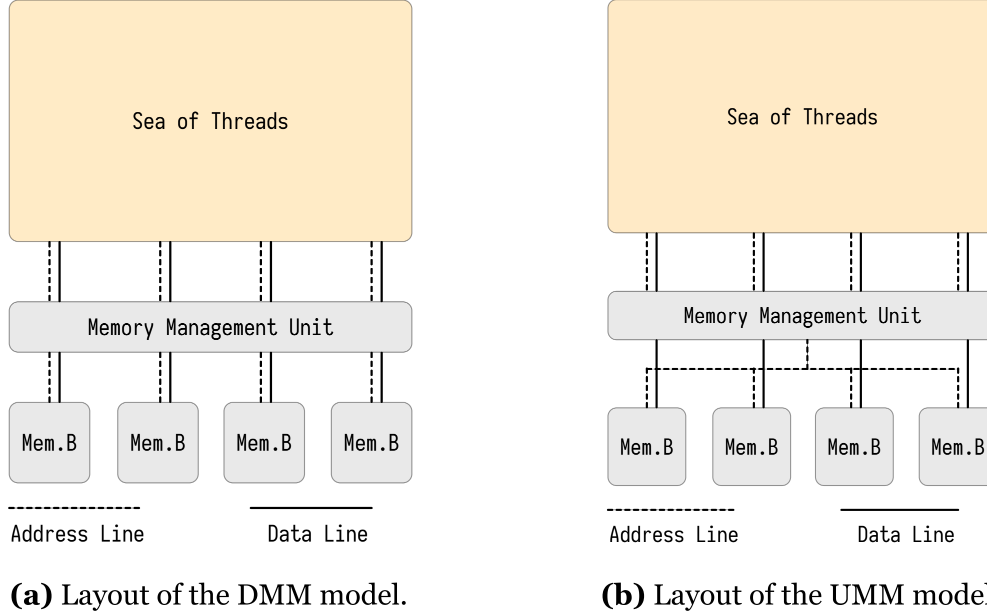
4.3.3 The Discrete Memory Machine Model

The DMM model for the GPU is parameterized by the number of threads p, the width w, and the latency l, which corresponds to memory banks of size w with (global) read and write latency l.[2] The memory in this model is then separated into *banks* B of memory, where

$$B[j] = \{m[j], m[j+w], m[j+2w], \ldots\}$$

denotes the j'th bank of memory (where each $m[j]$ is a memory cell). Memory cells that reside in different banks of memory can be accessed in unit time, and l time units are required in order to complete an access request to global memory, and

[2] And where $w|p$.



(a) Layout of the DMM model.

(b) Layout of the UMM model.

Figure 4.2: Layout of both the DMM and UMM models. Note how the only difference is the layout of the address line.

continuous requests are processed in a pipeline fashion. Thus, it would take $k+l-1$ time units to perform k continuous access requests to global memory.

The machine is assumed to have a number of threads $T(0),...,T(p-1)$. These p threads are then partitioned into p/w *warps* $W(0),...,W(p/w-1)$ such that the i'th warp is defined as

$$W(i) = \{T(i+w), T(i*w+1), ..., T((i+1)*w-1)\}$$

Each warp $W(i)$ is activated for (possible) memory access in a round-robin fashion. Once a warp is activated, each of its w threads then send memory access requests (one request per thread) to the memory bank B. Moreover once a given thread has sent a memory request, it cannot send another memory request until the previous request has been completed i.e., it must wait l time units between memory requests. A diagram of the DMM model is given in Figure 4.2.

Bank Conflicts and Broadcasts If more than one thread requests a given address from a memory bank in the same step this creates a *bank conflict* and the thread must wait until the previous memory request has finished before it can send its request. However, if all the threads request the same address then this creates a *broadcast* and only the cost for one memory request needs to be charged.

4.3.4 The Unified Memory Machine Model

Unlike the DMM model where *discrete* or individual memory addresses can be accessed, the UMM model only allows addressing memory based upon address groups. This is meant to reflect the fact that coalesced accesses to global memory can be combined into a single request, and therefore we want to talk about accessing a contiguous section of memory or *address group* instead of addressing to a given *location* in a given memory *bank* as we did in the DMM model. The UMM model can be seen as being defined in terms of the DMM model by adding *address groups* A where

$$A[j] = \{m[j * w], m[j * w + 1], \ldots, m[(j + 1) * w - 1]\}$$

denotes the j'th address group. To see how these relate to the memory banks that we saw in the DMM model, recall that a memory bank $B[j]$ was defined to be made up of addresses spaced w apart from each other

$$B[j] = \{m[j], m[j + w], m[j + 2w], \ldots\}$$

Thus an address group can be seen as taking a "horizontal slice" across all of the memory banks. This relationship can be seen in Figure 4.3.

Since memory cells in the same address group do not reside in the same memory bank (in the sense of the DMM model), an access to a single address by a warp can be coalesced to access the entire address group simultaneously. Moreover, memory accesses in different address groups cost one time unit each after charging the cost

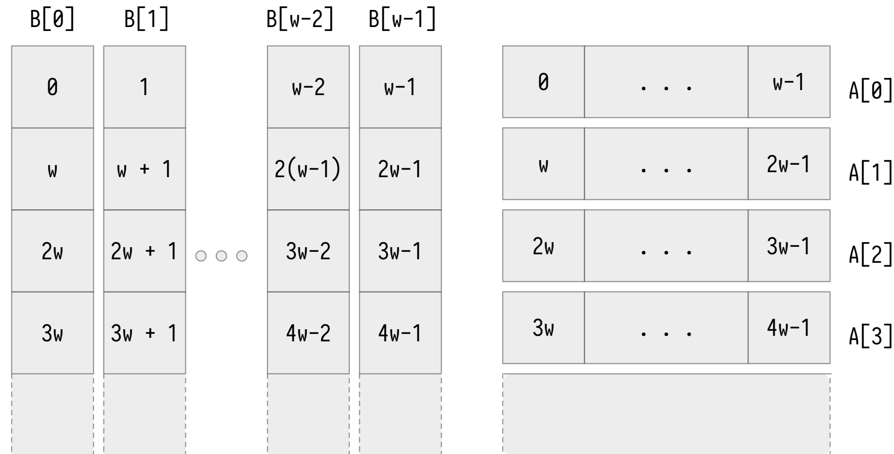


Figure 4.3: Address groups, and how they relate to the memory banks of the DMM model.

l of a memory access: accessing multiple memory locations in the same memory bank possibly causes a latency of l, but since we take into account pipelining of instructions these will only take a single unit of time after the first memory access cost l. We also have the same warp, threading, and warp-access pattern as in the DMM model. A diagram of the UMM model can be found in Figure 4.2.

4.3.5 Putting it all Together: the HMM Model

The HMM consists of d DMMs, along with a single UMM as illustrated in Figure 4.4. Each DMM has w memory banks; the UMM also has w memory banks. The memory banks for each of the DMMs is called the *shared memory*, and those in the UMM are called the *global memory*.

Each DMM in the model works independently from the others. Threads in each DMM are partitioned into warps of w threads, and each warp is dispatched for memory access in turn. Furthermore, threads can access the global memory via the UMM i.e. a warp of w threads in a given DMM can send memory access requests to global memory.

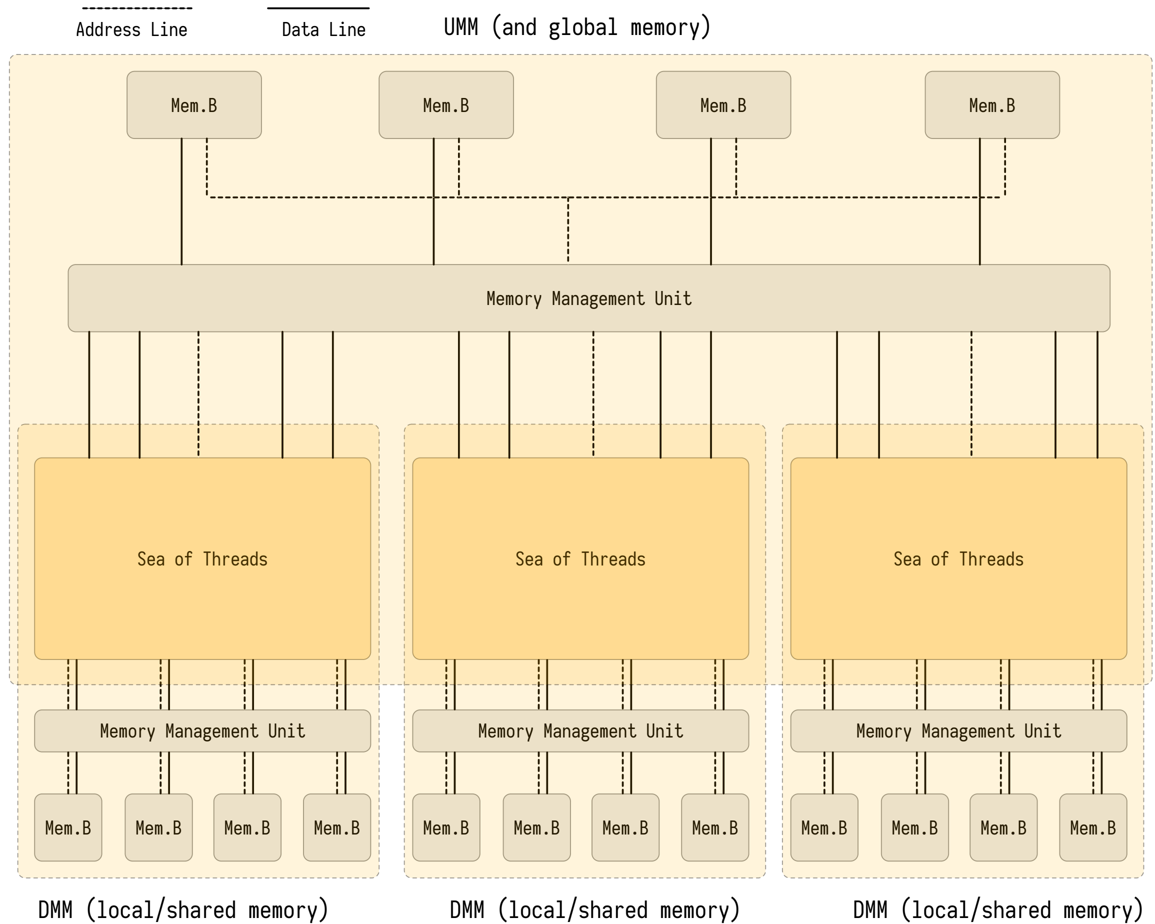


Figure 4.4: Layout of the HMM model.

The shared memory in the DMMs and the global memory in the UMM correspond to the shared memory of each SMP, and the global memory of the GPU. Since the shared memory has a very low latency rate (1 or 2 clock cycles [72]) we shall assume that the memory access latency for each DMM is 1. However, the latency for "global memory" on the GPU is significantly higher, with latencies frequently in the hundreds of clock cycles [72]. We therefore use the latency parameter l for the UMM to be the latency for global memory accesses in the HMM.

The global memory in the UMM can handle requests by warps in the various DMMs in turn, and requests are processed in a pipeline fashion. Since each memory request

$$
\begin{array}{lll}
m & \triangleq & \{i \mapsto v, \ldots\} \\
B_w(i) & \triangleq & \{m[i], m[i+w], \ldots\} \\
A_w(i) & \triangleq & \{m[i*w], m[i*w+1], \ldots, m[(i+1)*w-1]\} \\
DMM^l_w(p) & \triangleq & \left\{\{B_w(i)\}_i, \{W(i)\}_{i=1}^{p/w-1}, l\right\} \\
UMM^l_w(p) & \triangleq & \left\{\{A_w(i)\}_i, \{W(i)\}_{i=1}^{p/w-1}, l\right\} \\
HMM^l_w(p,d) & \triangleq & \left\{\left\{DMM^1_w(p)\right\}_{i=1}^{d}, UMM^l_w(p)\right\}
\end{array}
$$

Figure 4.5: Definitions of the abstract machine states used for the GPU.

to the global memory in the UMM has latency l, it will take l units of time to fulfill the memory access request if all of the requests by the warp are in the same address group.

The specification of the DMM, UMM, and HMM are provided in Figure 4.5.

4.4 GPU Cost Graph Combinators

As we saw in the cost semantics for Λ_C and in the various cost semantics in Section 2.1, cost DAGs are useful for representing whole programs that start, and end, on a single thread. Thus these DAGs have a single source vertex (with in-degree zero), and single sink vertex (with out-degree zero) at all times. This, however, is not the case on the GPU; programs most certainly *do not* start on a single thread. As such, the DAGs for Λ^G will no longer have a single source nor a single sink vertex.

Due to the high amount of parallelism and special memory characteristics of the GPU, the standard approach of analyzing costs via a normal computation graph structure annotated with costs is infeasible. We therefore group GPU computations in the cost graph based on the underlying computational structure of the GPU: we first build up a *thread-DAG* with a single source/sink representing each work item (thread) in a warp; each of these thread-DAGs are then combined to create a *warp graph*, and finally each of these warp graphs are combined to create a *warp pool*

that represent work-groups on the GPU. These warp pools are then put together to form a set of graphs that represent the GPU computation, which we call a *device box*.

Since the thread graphs that are built represent the individual threads running on the GPU, they themselves cannot fork off new threads of computation. Thus the graph combinators for thread graphs are restricted to serial composition $\oplus$ of two graphs. Related to this, since we no longer have fork-join parallelism we no longer have a parallel composition operator $\otimes$ for graphs. Instead we will be composing *sets* of thread-graphs (i.e., a warp pool) with other *sets* of thread-graphs in parallel i.e., composing a warp pool in parallel with another warp pool in a non-fork-join manner. Since the GPU can have many of these work groups running simultaneously for a given kernel, the resulting cost graph built by these combinators will not necessarily be a series-parallel, or rooted, DAG.

4.4.1 Thread Graphs and Meta-edges

The definition of thread graphs is given in Figure 4.6, and it is similar to the definition of the cost graphs in Section 3.6.1 except that we no longer have the parallel composition operator, and the edges in our thread graphs are no longer weighted—almost (more on the exception to this in a second). As we will see shortly, this is due to the fact that we no longer want to reason at the thread level about computational costs but instead at the *warp step* level.

Since it is legal for a thread to spark sub-kernels which may *not* run immediately, but only cause the subsequent kernel to be placed in the “work-queue” of kernels that the GPU needs to schedule—and is therefore subject to a different scheduling policy—we don’t know when kernel calls will complete subject to some other, global, knowledge about the computation graph. Since this scheduling of a sub-kernel amounts to a

de-facto hoisting of the kernel to the top-level, the warp pool created by such a sub-kernel will form another component that will be composed in parallel with the rest of the warp pools in the device box. However, the rest of the computation in the thread that sparked the sub-kernel depends on the value of this sub-kernel, and thus cannot continue until it has finished. Thus, instead of inlining kernel-graphs in the thread graph, we instead mark the edge of the operation that sparked the sub-kernel as a *meta-edge* that is treated abstractly,[3] and will later on be *quiesced* by our global semantics and the scheduling policy once the kernel that the edge depends upon has been run.

It is important to note that the nodes in the DAG that is formed by the thread graphs now matter. Since we need to ensure that the same instruction is executed for each step of the warp, each node tracks the individual operation that is being performed—including memory accesses—so that they can be used later on for profiling. The grammar for graph nodes α is given in Figure 4.6, and each node in the graph is implicitly tagged with a unique identifier; thus there are two types of equality that we will be using over thread graph nodes: *reference equality*—do the nodes have the same unique tag—and *structural equality*—the context should make clear which equality we are using but where it isn't we will mention which version of equality we are using explicitly. Furthermore, as opposed to the cost graphs for Λ_{C} in Chapter 3, thread graphs are unweighted.

4.4.2 Warp Graphs

Once we have a number of thread-graphs $\{c_1, \ldots, c_w\}$ these are combined to create a *warp graph*. This warp graph represents a single warp of width w on the GPU, and is thus made up of a set of threads—the $\{c_1, \ldots, c_w\}$. Thus warp graphs can be seen

[3] As we will see later on these meta-edges will be crucial for restricting scheduling policies to only valid ones over these graphs.

κ	$::=$	uid	Kernel IDs
α	$::=$	$\texttt{less} \mid \texttt{mul} \mid \texttt{reg} \mid \texttt{mem}[\ell] \mid \texttt{kern}[\kappa]$	Labels
e	$::=$	$(\alpha_1,\alpha_2,\mathbb{N}) \mid \blacksquare(\alpha_1,\alpha_2)$	Directed (Meta-)Edges
V	$::=$	$\{\alpha_1,\ldots,\alpha_n\}$	Label Set
E	$::=$	$\{e_1,\ldots,e_n\}$	Edge Set
c	$::=$	(α_1,α_2,V,E)	Thread Graphs

$$
\begin{aligned}
[\alpha] &= (\alpha,\alpha,\{\alpha\},\emptyset)\\
[(\alpha_1,\alpha_2)] &= (\alpha_1,\alpha_2,\{\alpha_1,\alpha_2\},\{(\alpha_1,\alpha_2)\})\\
[\blacksquare(\alpha_1,\alpha_2)] &= (\alpha_1,\alpha_2,\{\alpha_1,\alpha_2\},\{\blacksquare(\alpha_1,\alpha_2)\})\\
(\alpha_1',\alpha_1,V,E) \oplus (\alpha_2',\alpha_2,V',E') &= (\alpha_1',\alpha_2,V\cup V',E\cup E'\cup\{(\alpha_1,\alpha_2')\})\\
(\alpha_1',\alpha_1,V,E) \oplus_{\blacksquare} (\alpha_2',\alpha_2,V',E') &= (\alpha_1',\alpha_2,V\cup V',E\cup E'\cup\{\blacksquare(\alpha_1,\alpha_2')\})
\end{aligned}
$$

Figure 4.6: Thread graph definitions.

as a sort of "administrative redex" of sorts that we use to group threads into a logical chunk:

$$\mathsf{W} = \{c_1,\ldots,c_w\} \tag{4.1}$$

4.4.3 Warp Pools

Each work-group (or SMP) on the GPU is made up of a number of warps, and shares a local memory state with all other warps in that work-group. Thus, once we have a number of warps $\mathsf{W}_1,\ldots,\mathsf{W}_n$, these are grouped together into a set of warps along with a local memory state σ which they all share, the (unique) kernel identifier κ for the kernel represented by the warp pool, something called a meta location $\blacksquare\ell$ (more on this in a second) and location ℓ where the result will be be stored.

$$\mathcal{W} = (\kappa,\blacksquare\ell,\ell,\langle \mathsf{W}_1,\ldots,\mathsf{W}_n\rangle^{\sigma}) \tag{4.2}$$

We call these warp pools, since due to our machine model for work-groups (the UMM), at any given step, only one of these warps can make a step, yet they share the same local (work-group level) memory so it makes sense to group them together to make reasoning about the stepping relation later on easier. Thus, we can view these in some sense as being somewhat similar to thread pools that we might see in a runtime.

$$
\begin{array}{lcl}
a & \in & ValidAddresses \subset \mathbb{N} \\
\Sigma & ::= & \{a_1, \ldots, a_n\} \\
M & ::= & \{B_w(i)\}_i \\
L & ::= & (M, \{[a_1, a_2], \ldots [a_{n-1}, a_n]\}) \\
\sigma & ::= & (L, \Sigma) \\
\mathsf{W}_i & ::= & \{c_{i*w}, \ldots, c_{(i+1)*w-1}\} \\
\mathcal{W}(i) & ::= & (\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_i, \ldots, \mathsf{W}_k \rangle^{\sigma}) \\
\mathsf{K} & ::= & \cdot \mid \kappa \hookrightarrow (f, [e_i], \blacksquare\ell) \Cup \mathsf{K} \\
\mathcal{D} & ::= & \cdot \mid \mathcal{W}(i) \boxtimes \mathcal{D}
\end{array}
$$

Figure 4.7: The various stages and layers that make up a device box. Each c_i is a thread graph, and the GPU is assumed to have warps and memory banks of width w, with p threads per work-group. The semantics of these definitions are explained in subsequent sections.

4.4.4 Device Boxes

Once we have the various warp pools $\mathcal{W}(1), \ldots, \mathcal{W}(n)$ for a given work-group, these are then combined in parallel with the other works-groups for that kernel using the *bulk-parallel* operator $\boxtimes$. $\mathcal{W} \boxtimes \mathcal{D}$ can be seen as placing the work group $\mathcal{W}$ "alongside" the work-groups in $\mathcal{D}$ and indexing them by their kernel id κ. The syntax of the bulk-parallel operator, device boxes, local memory state, warp pools, and warp graphs are given in Figure 4.7. The semantics of each of these are explained in Sections 4.4 and 4.5.

4.5 Cost Semantics for Λ^{G}

4.5.1 Meta Locations

Recall that in a cost graph, each node represents an instruction, with the edges between them representing dependencies between these instructions. Furthermore, as opposed to the cost graphs we saw for Λ_{C}, the edges in our cost graphs no longer consist solely of "normal" edges: we now allow meta-edges that represent control dependencies on promised-to-be-done computation over which the current thread

has no control. Thus the values of these promised computations cannot be known locally in a thread graph. Instead, the computation that produces these values is passed to a warp scheduler that schedules the computations outside of the purview of the thread graph. Due to this, there is no way to *locally* determine both where the result of the computation will be placed, and how much the computation that writes to that location costs. The latter will be handled in Section 4.6, and to handle the former, we need to introduce a way to defer looking at results of a computation until that computation has finished—while still allowing ourselves to reason about this location (at the very least, abstractly).

Since the only form in Λ^{G} that requires us to introduce these "deferred values from computation" are `parmap` (since these are the only forms in the language that can launch kernels), we only have to handle deferred locations as opposed to deferred values in generality. Since these deferred locations are directly tied to meta-edges—the computational dependency that produces a *meta location* is given by a meta-edge—we call these deferred locations meta locations. At the time of creation of these meta locations $\blacksquare\ell$, we know a couple of their key features:

1. We know the region name r (and hence the region) which the meta location will resolve or *quiesce* to;
2. We know the computation that produces the meta location, and thus can associate the quiescence of the edge with the computation that fulfills the dependency represented by that edge.

Since we will need to handle these meta locations abstractly until they have quiesced, meta locations hold both the region name r for the region in which they will be allocated along with a unique identifier that will be used by the global cost semantics to quiesce the edge in the cost graph once the computation has been run. These are then used to pass the local information known in the local cost semantics to the global cost semantics that will later on use this information. It is worthwhile

remarking at this point that our memory judgements *do not* handle meta locations, since we should never encounter them—since meta-edges are the only computations that can produce meta locations, we will restrict allowable schedules over the computation graphs so that meta-edges are “unschedulable” and will only become “schedulable” once the edge has been quiesced. However, we do need to be able to record meta locations in the nodes of thread graphs (these will later be replaced during the quiescence of the edge), and we therefore amend the grammar for graph nodes α so that they are now permitted:

$$\alpha ::= \ldots \mid \texttt{mem}[\ell] \mid \texttt{mem}[\blacksquare\ell] \tag{4.3}$$

Just precisely how these meta-edges and meta locations will be quiesced and scheduled will be explained in Section 4.6.

4.5.2 Reading and Writing to Memory

Since reading and writing to local memory has a significantly different cost than reading and writing to global memory, the rules for reading and writing costs will need to take into account whether a given memory access is to local or global memory. Furthermore, in order to be able to account for bank conflicts and broadcasts, we will need to keep track of all the memory locations accessed in a given step of a warp via a trace Σ.

However, the rules in the cost semantics will not build this trace. Instead, the trace will be built based on the traversal over the cost graph determined by the scheduling policy in Section 4.6. The cost of the step of the warp will then be calculated based upon the (multi-) set of memory locations accumulated in the trace (Section 4.7).

Recall now that each node in the cost graph represents a single instruction, and since we record memory accesses as nodes in the cost graph, this gives rise to the following allocation rules for reading and writing to memory:

$$R; v \downarrow o@R'$$
$$\mathcal{S}; v \uparrow v'; \overline{\ell}$$

These are read respectively as "allocating heap value v in region R results in an updated region R' allocated at offset o", and "reading value v from region stack $\mathcal{S}$ results in heap value v' with possible memory access $\overline{\ell}$," where we use $\overline{\ell}$ to represent either a location ℓ—in the case that we accessed the location ℓ—or non-location $\bullet_\ell$—in the case that v was a non-heap-allocated value. Note that as opposed to the read and write judgements that we saw for the CPU in Section 3.6.1 each allocation judgement is *not annotated with a cost*. Instead, each read and write judgement returns the location accessed by the respective rule. The thread semantics in Section 4.5.3 will then annotate the thread graphs with these locations, which we will then use during traversal of the graph to calculate the cost of the allocation.

We perform this delay of calculating the costs of memory accessed in order to push the determination of the costs from the local—thread—semantics to the more global warp-semantics; due to the memory characteristics of the GPU, we will need the context only available to us at the warp-step level in order to calculate bank-conflicts and broadcasts, and further, we aren't interested in calculating the cost of each individual thread so much as the cost for each step that is taken by the warp.

4.5.3 Thread Semantics

Since we push the determination of costs to the scheduling phase in our cost semantics, the rules for our thread-graphs do not return a trace. Instead, the rules encode the memory accesses and instructions that are performed in the nodes of the cost graph that is built, and these nodes will then be used to build the trace once a given

$$\text{NoExpandWE}\ \frac{|v| + \mathsf{ap} > |R|}{R; v \downarrow \mathtt{err}} \qquad \text{Alloc}\ \frac{|v| + \mathsf{ap} \leq |R| \qquad o = \mathsf{ap} \qquad R' = (\mathsf{ap} + |v| + 1, s, \mathsf{refcnt}, \nu[o \mapsto v])}{R; v \downarrow o@R'} \qquad \text{ReadV}\ \frac{v \neq (r, o)}{\mathcal{S}; v \uparrow v; \bullet_\ell}$$

$$\text{Read}\ \frac{\mathcal{S}(r) = R\{\mathsf{ap}, _, _, _, \nu\} \qquad o < \mathsf{ap} \qquad o \in \mathsf{dom}(\nu) \qquad v = \nu(o)}{\mathcal{S}; (r, o) \uparrow v; (r, o)}$$

Figure 4.8: Read and write judgements for the GPU. Note that the location that has been read or writing is always returned and that no cost is associated with the actual judgement.

set of nodes have been scheduled. Indeed, if we view the trace as being built from a given traversal of the cost graph, if we had the semantic rules build the trace instead of the schedule this would amount to traces of the execution being built from the result of a depth-first traversal of the cost graph (since we evaluate serially until we hit a value in our big-step semantics). Instead, we need the trace to be the result of a type of breadth-first traversal over the graph.

The thread semantics for Λ^{G} are given by the following judgement in Figures 4.9 and 4.10:

$$\mathsf{K}; \mathcal{S} \Big|_E\, e \Downarrow^{\mathsf{t}} v; c \Big| \mathcal{S}'; \mathsf{K}' \tag{4.4}$$

which is read as "expression e in region stack $\mathcal{S}$, with kernel pool K, and environment E reduces to value v with cost graph c, region stack $\mathcal{S}'$, and updated kernel pool K'." A kernel pool K is given by a mapping from unique kernel ids κ to computations to run as a kernel:

$$\mathsf{K} ::= \cdot \ \mid\ \kappa \hookrightarrow (f, [e_i], \blacksquare\ell) \uplus \mathsf{K}$$

Each kernel that is enqueued in the kernel pool is represented as a 3-tuple of the expression f that is to be run, the pieces of data $[e_i]$ on which it acts, and the *meta location* $\blacksquare\ell$ where the result of the kernel should be stored. This enqueuing of a kernel can be seen in the rule for the `parmap` form of the language, where instead of

creating parallelism in the cost graph c, we instead enqueue the kernel in the kernel pool to be scheduled later on, and create a meta-edge that will be quiesced once the kernel has been run.

Further, since we do not allow *any* parallelism to occur at the thread level, we degenerate the parallel composition form $e_1 \| e_2$ so that this results in serial composition of the two expressions. This means that each node in the cost graph that is built by the thread semantics in Figures 4.9 and 4.10 will have in- and out-degree at most 1. The overall strategies in the rules for the thread semantics are fairly similar to the semantic rules we saw for Λ_C in Chapter 3, with the exception that the thread graphs are unweighted, and that we allow meta-edges. Each of these meta-edges will then be quiesced by the scheduling policy once the kernel that the location depends upon has been scheduled by the scheduling policy.

We now move on to the more interesting and subtle portion of the cost semantics for the GPU: the cost semantics for warps, and the global cost semantics.

4.5.4 Global Cost Semantics

Our global semantics make use of an environment which we call a *kernel result pool* ψ that maps (unique) kernel identifiers to the locations in memory where their results are stored.

The grammar for ψ is given by the following:

$$\psi \quad ::= \quad \{\} \mid \psi, \kappa \mapsto \ell \tag{4.5}$$

While the kernel result ψ could be elided in the global rules with some careful bookkeeping, it will prove useful in the next chapter when we start transferring computations between the CPU and GPU to have this mapping of kernel ids to locations in memory where their results are stored. In particular, this allows us to easily

Thread graphs: $\mathsf{K}; \mathcal{S} \,\big|_E\, e \Downarrow^{\mathsf{t}} v; c \,\big|\, \mathcal{S}'; \mathsf{K}'$

$$\frac{x \in \mathsf{dom}(E)}{\mathsf{K}; \mathcal{S} \,\big|_E\, x \Downarrow^{\mathsf{t}} E(x); [\mathtt{reg}] \,\big|\, \mathcal{S}; \mathsf{K}} \qquad \frac{}{\mathsf{K}; \mathcal{S} \,\big|_E\, i \Downarrow^{\mathsf{t}} i; [\mathtt{reg}] \,\big|\, \mathcal{S}'; \mathsf{K}}$$

$$\frac{\begin{array}{c} \mathsf{K}; \mathcal{S} \,\big|_E\, e_1 \Downarrow^{\mathsf{t}} v_1; c_1 \,\big|\, \mathcal{S}_1; \mathsf{K}_1 \qquad \mathcal{S}_1; \sigma; v_1 \uparrow i_1 \\ \mathsf{K}_1; \mathcal{S}_1 \,\big|_E\, e_2 \Downarrow^{\mathsf{t}} v_2; c_2 \,\big|\, \mathcal{S}_2; \mathsf{K}_2 \qquad \mathcal{S}_2; v_2 \uparrow i_2 \\ b = i_1 < i_2 \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, e_1 < e_2 \Downarrow^{\mathsf{t}} b; c_1 \oplus [\mathtt{reg}] \oplus c_2 \oplus [\mathtt{reg}] \oplus [\mathtt{less}] \,\big|\, \mathcal{S}_2; \mathsf{K}_2}$$

$$\frac{\begin{array}{c} \mathsf{K}; \mathcal{S} \,\big|_E\, e_1 \Downarrow^{\mathsf{t}} v_1; c_1 \,\big|\, \mathcal{S}_1; \mathsf{K}_1 \qquad \mathcal{S}_1; v_1 \uparrow i_1 \\ \mathsf{K}_1; \mathcal{S}_1 \,\big|_E\, e_2 \Downarrow^{\mathsf{t}} v_2; c_2 \,\big|\, \mathcal{S}_2; \mathsf{K}_2 \qquad \mathcal{S}_2; v_2 \uparrow i_2 \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, e_1 * e_2 \Downarrow^{\mathsf{t}} i_1 * i_2; c_1 \oplus [\mathtt{reg}] \oplus c_2 \oplus [\mathtt{reg}] \oplus [\mathtt{mul}] \,\big|\, \mathcal{S}_2; \mathsf{K}_2}$$

$$\frac{}{\mathsf{K}; \mathcal{S} \,\big|_E\, \mathtt{True} \Downarrow^{\mathsf{t}} \mathtt{True}; [\mathtt{reg}] \,\big|\, \mathcal{S}; \mathsf{K}} \qquad \frac{}{\mathsf{K}; \mathcal{S} \,\big|_E\, \mathtt{False} \Downarrow^{\mathsf{t}} \mathtt{False}; [\mathtt{reg}] \,\big|\, \mathcal{S}; \mathsf{K}}$$

$$\frac{\begin{array}{c} \mathsf{K}; \mathcal{S} \,\big|_E\, e_1 \Downarrow^{\mathsf{t}} b; c_1 \,\big|\, \mathcal{S}'; \mathsf{K}' \qquad \mathcal{S}'; b \uparrow \mathtt{True} \\ \mathsf{K}; \mathcal{S}' \,\big|_E\, \mathsf{K}' \Downarrow^{\mathsf{t}} e_2; \mathsf{K}'' \,\big|\, v; \mathcal{S}'' c_2 \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow^{\mathsf{t}} v; c_1 \oplus c_2 \,\big|\, \mathcal{S}''; \mathsf{K}''}$$

$$\frac{\begin{array}{c} \mathsf{K}; \mathcal{S} \,\big|_E\, e_1 \Downarrow^{\mathsf{t}} b; c_1 \,\big|\, \mathcal{S}'; \mathsf{K}' \qquad \mathcal{S}'; b \uparrow \mathtt{False} \\ \mathsf{K}'; \mathcal{S}' \,\big|_E\, e_3 \Downarrow^{\mathsf{t}} v; c_2 \,\big|\, \mathcal{S}''; \mathsf{K}'' \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow^{\mathsf{t}} v; c_1 \oplus [\mathtt{reg}] \oplus c_2 \,\big|\, \mathcal{S}''; \mathsf{K}''}$$

$$\frac{\begin{array}{c} r = E(\rho) \qquad R = \mathcal{S}(r) \qquad R; \sigma;\ \mathtt{clos}\ (x : \tau, e, E) \downarrow o@R' \\ \mathcal{S}' = \mathcal{S}[R'/R] \qquad \ell = (r, o) \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, \lambda x : \tau.^{\varphi} e\ \mathtt{at}\ \rho \Downarrow^{\mathsf{t}} \ell; [\mathtt{reg}] \oplus [\mathtt{mem}[\ell]] \,\big|\, \mathcal{S}'; \mathsf{K}}$$

$$\frac{\begin{array}{c} \mathsf{K}; \mathcal{S} \,\big|_E\, e_1 \Downarrow^{\mathsf{t}} \ell; c_1 \,\big|\, \mathcal{S}_1; \mathsf{K}_1 \qquad \mathcal{S}_1; \ell \uparrow \mathtt{clos}\ (x : \tau, e, E) \\ \mathsf{K}_1; \mathcal{S}_1 \,\big|_E\, e_2 \Downarrow^{\mathsf{t}} v_1; c_2 \,\big|\, \mathcal{S}_2; \mathsf{K}_2 \qquad \mathsf{K}_2; \mathcal{S}_2 \,\big|_{E[x \mapsto v_1]}\, e \Downarrow^{\mathsf{t}} v_2; c_3 \,\big|\, \mathcal{S}_3; \mathsf{K}_3 \end{array}}{\mathsf{K}; \mathcal{S} \,\big|_E\, e_1\ e_2 \Downarrow^{\mathsf{t}} v_2; c_1 \oplus c_2 \oplus [\mathtt{mem}[\ell]] \oplus c_3 \,\big|\, \mathcal{S}_3; \mathsf{K}_3}$$

Figure 4.9: Local (thread) cost semantics for the GPU.

$$\frac{\begin{array}{c}\mathsf{K};\mathcal{S}\Big|_E\, e_1 \Downarrow^{\mathsf{t}} v_1; c_1\Big|\,\mathcal{S}_1;\mathsf{K}_1 \qquad \mathsf{K}_1;\mathcal{S}_1\Big|_E\, e_2 \Downarrow^{\mathsf{t}} v_2; c_2\Big|\,\mathcal{S}_2;\mathsf{K}_2\\ r = E(\rho) \quad R = \mathcal{S}_2(r) \quad R;\sigma;(v_1,v_2)\downarrow o@R' \quad \mathcal{S}_3 = \mathcal{S}_2[R'/R] \quad \ell = (r,o)\end{array}}{\mathsf{K};\mathcal{S}\Big|_E\,(e_1,e_2)\ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \ell; c_1 \oplus c_2 \oplus [\mathsf{mem}[\ell]]\Big|\,\mathcal{S}_3;\mathsf{K}_2}$$

$$\frac{\mathsf{K};\mathcal{S}\Big|_E\, e \Downarrow^{\mathsf{t}} \ell; c\Big|\,\mathcal{S}';\mathsf{K}' \qquad \mathcal{S}';\ell \uparrow (v_1,v_2)}{\mathsf{K};\mathcal{S}\Big|_E\ \texttt{fst}\ e \Downarrow^{\mathsf{t}} v_1; [\mathsf{mem}[\ell]] \oplus c \oplus [\mathsf{reg}]\Big|\,\mathcal{S}';\mathsf{K}'}$$

$$\frac{\mathsf{K};\mathcal{S}\Big|_E\, e \Downarrow^{\mathsf{t}} \ell; c\Big|\,\mathcal{S}';\mathsf{K}' \qquad \mathcal{S}';\ell \uparrow (v_1,v_2)}{\mathsf{K};\mathcal{S}\Big|_E\ \texttt{snd}\ e \Downarrow^{\mathsf{t}} v_2; [\mathsf{mem}[\ell]] \oplus c \oplus [\mathsf{reg}]\Big|\,\mathcal{S}';\mathsf{K}'}$$

$$\frac{\begin{array}{c} r = E(\rho) \quad R = \mathcal{S}(r) \quad R;\sigma;\ \texttt{rclos}\ (\rho,\varphi,\varphi',u)\downarrow o@R'\\ \mathcal{S}' = \mathcal{S}[R'/R] \qquad \ell = (r,o)\end{array}}{\mathsf{K};\mathcal{S}\Big|_E\,\lambda\rho \geq \varphi.^{\varphi'} u\ \ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \ell; [\mathsf{reg}] \oplus [\mathsf{mem}[\ell]]\Big|\,\mathcal{S}';\mathsf{K}}$$

$$\frac{\begin{array}{c}\mathsf{K};\mathcal{S}\Big|_E\, e \Downarrow^{\mathsf{t}} \ell; c_1\Big|\,\mathcal{S}';\mathsf{K}' \qquad \mathcal{S}';\ell \uparrow\ \texttt{rclos}\ (\rho',\varphi,\varphi',u)\\ r = E(\rho) \qquad \mathsf{K}';\mathcal{S}'\Big|_{E[\rho'\mapsto r]}\, u \Downarrow^{\mathsf{t}} v; c_2\Big|\,\mathcal{S}'';\mathsf{K}''\end{array}}{\mathsf{K};\mathcal{S}\Big|_E\, e[\rho] \Downarrow^{\mathsf{t}} v; c_1 \oplus [\mathsf{mem}[\ell]] \oplus c_2\Big|\,\mathcal{S}'';\mathsf{K}''}$$

$$\frac{\begin{array}{c} r = E(\rho) \quad R = \mathcal{S}(r) \quad R;\sigma;\ \texttt{clos}\ (f:\tau,u,E[f\mapsto(r,o)])\downarrow o@R'\\ \mathcal{S}' = \mathcal{S}[R'/R] \qquad \mathsf{K};\mathcal{S}'\Big|_{E[f\mapsto(r,o)]}\, u \Downarrow^{\mathsf{t}} v; c\Big|\,\mathcal{S}'';\mathsf{K}''\end{array}}{\mathsf{K};\mathcal{S}\Big|_E\ \texttt{fix}\ f:(\mu,\rho).u \Downarrow^{\mathsf{t}} v; [\mathsf{reg}] \oplus [\mathsf{mem}[(r,o)]] \oplus c\Big|\,\mathcal{S}'';\mathsf{K}''}$$

$$\frac{\begin{array}{c}\mathsf{K}_{i-1};\mathcal{S}_{i-1}\Big|_E\, e_i \Downarrow^{\mathsf{t}} v_i; c_i\Big|\,\mathcal{S}_i;\mathsf{K}_i \quad r = E(\rho) \quad R = \mathcal{S}_n(r)\\ R;\sigma;[v_1,\ldots,v_n]\downarrow o@R' \quad \ell = (r,o) \quad \mathcal{S}'_n = \mathcal{S}_n[R'/R]\end{array}}{\mathsf{K}_0;\mathcal{S}_0\Big|_E\,[e_1,\ldots,e_n]\ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \ell; \left(\bigoplus_i c_i\right) \oplus [\mathsf{reg}] \oplus [\mathsf{mem}[\ell]]\Big|\,\mathcal{S}'_n;\mathsf{K}_n}$$

$$\frac{\begin{array}{c}\mathsf{K};\mathcal{S}\Big|_E\, e_1 \Downarrow^{\mathsf{t}} v_1; c_1\Big|\,\mathcal{S}_1;\mathsf{K}_1 \qquad \mathsf{K}_1;\mathcal{S}_1\Big|_E\, e_2 \Downarrow^{\mathsf{t}} v_2; c_2\Big|\,\mathcal{S}_2;\mathsf{K}_2\\ r = E(\rho) \quad R = \mathcal{S}_2(r) \quad R;\sigma;(v_1,v_2)\downarrow o@R' \quad \mathcal{S}_3 = \mathcal{S}_2[R'/R] \quad \ell = (r,o)\end{array}}{\mathsf{K};\mathcal{S}\Big|_E\, e_1 \| e_2\ \ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \ell; c_1 \oplus c_2 \oplus [\mathsf{reg}] \oplus [\mathsf{mem}[\ell]]\Big|\,\mathcal{S}'';\mathsf{K}_2}$$

$$\frac{r = E(\rho) \qquad (\blacksquare\ell,\kappa\ \text{fresh})}{\mathsf{K};\mathcal{S}\Big|_E\ \texttt{parmap}\ f\ [e_1,\ldots,e_n]\ \ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \blacksquare\ell; [\mathsf{kern}[\kappa]] \oplus_{\blacksquare} [\mathsf{mem}[\blacksquare\ell]]\Big|\,\mathcal{S};\kappa \hookrightarrow (f,[e_i],\blacksquare\ell) \Cup \mathsf{K}}$$

Figure 4.10: Local (thread) cost semantics for the GPU. (Continued)

associate the result of the kernel to the node in the thread graph that sparked that sub-kernel.

4.5.4.1 Warp Semantics

Given a number of thread graphs, the warp semantics serve as a way for us to group computations together in such a way that we can reason about local memory characteristics on the GPU during the scheduling of the graph. Furthermore, this allows tracking the running of a single warp at a time, and determining the trace for the warp at a given step in the schedule.

The definition of the cost semantics for warps is given in Figure 4.11. Since the only expression that can spark a kernel is a call to `parmap`, the warp graph judgement only takes an expression $f\ [e_i]$ representing a kernel in which f is to be applied to the e_i in parallel. Given this expression, and the current region stack $\mathcal{S}$ and environment E, it builds thread graphs for each $f\ e_i$ and groups these together into a number of warp graphs W_i that together constitute a warp pool $(\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^{\sigma})$ representing the kernel $f\ [e_i]$. This is then associated with the kernel identifier κ, and the meta location $\blacksquare\ell$ and location ℓ for the kernel as well.

4.5.4.2 Kernel Semantics

Since each kernel is spawned by first being placed in the kernel pool, the kernel graph rules are responsible for creating the kernel graphs from the kernel pool, and adding them to the device box. In the case that the kernel pool is empty, an empty kernel graph is generated. In the case that there is a kernel $(f, [e_i], \blacksquare\ell)$ in K, we first build the kernel graphs for the rest of the kernel pool:

$$\mathcal{S} | \mathsf{K} \Downarrow^{\mathsf{k}} \psi; \mathcal{D} | \mathcal{S}_1$$

after which we build the warp graph for the kernel under question:

$$\mathcal{S}_1|^r_E\ f\ [e_i] \Downarrow^{\mathsf{W}} \ell; \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^\sigma | \mathcal{S}_2; \mathsf{K}' \tag{4.6}$$

Since the building of the kernel graph may result in further kernels to run which are placed in a kernel pool K' in Equation (4.6), we then build the kernel graphs for these:

$$\mathcal{S}_2|\mathsf{K}' \Downarrow^{\mathsf{k}} \psi; \mathcal{D}''|\mathcal{S}_3 \tag{4.7}$$

After this, we then update the kernel result pool ψ with the mapping of the kernel's unique identifier κ to the location ℓ where the result of the kernel is stored. At this point, we then add the warp pool $(\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^\sigma)$ to our device box $\mathcal{D}'$ along with combining the device box $\mathcal{D}''$ that we got from Equation (4.7), and return back our updated region stack.

Once this rule has completed, and we have no more kernels in our kernel pool, the device box that we are left with represents the cost graph for the computation as a whole on the GPU, and the kernel result pool ψ contains where the result of each kernel that has been run resides.

4.5.4.3 Quiescing Cost Graphs

Since our thread graphs may have meta locations in them that need to be removed before certain nodes in them will become schedulable, we also provide a way to remove meta locations from the graph once the kernel has been run. This will be used by the scheduling policy in the next section to "unblock" and quiesce nodes in the thread graphs once the kernels that they have created have returned their result. The judgement for these takes a quiescence environment γ which is a mapping from meta location $\blacksquare\ell$ to locations ℓ together with a device box $\mathcal{D}$. It then substitutes

Warp graphs: $\mathcal{S}|^r_E\ f\ [e_i]^n_{i=1} \Downarrow^{\mathsf{w}} \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^\sigma | \mathcal{S}'; \mathsf{K}$

$$\frac{\begin{array}{cc} \mathsf{K}; \mathcal{S} \Big|_E\ f\ e_i \Downarrow^{\mathsf{t}} v_i; c_i \Big| \mathcal{S}_i; \mathsf{K}_i & \mathcal{S}' = \mathcal{S}_1 \diamond \ldots \diamond \mathcal{S}_n \\ R = \mathcal{S}'(r) \quad R; [v_i]^n_{i=1} \downarrow o@R' & \mathcal{S}'' = \mathcal{S}'[R'/R] \quad \mathsf{K} = \Cup_i \mathsf{K}_i \\ \mathsf{W}_j = \{c_{jw+1}, \ldots c_{(j+1)w}\} & j = 0, \ldots, \lceil n/w \rceil \end{array}}{\mathcal{S}|^r_E\ f\ [e_i]^n_{i=1} \Downarrow^{\mathsf{w}} (o, r); \langle \mathsf{W}_1, \ldots, \mathsf{W}_{\lceil n/w \rceil} \rangle^\sigma | \mathcal{S}''; \mathsf{K}}$$

Kernel Graphs: $\mathcal{S}|_E\ \mathsf{K} \Downarrow^{\mathsf{k}} \psi; \mathcal{D} | \mathcal{S}'$

$$\frac{}{\mathcal{S}|_E\ \emptyset \Downarrow^{\mathsf{k}} \{\}; \{\} | \mathcal{S}}$$

$$\frac{\begin{array}{c} \mathcal{S}|\mathsf{K} \Downarrow^{\mathsf{k}} \psi; \mathcal{D}|\mathcal{S}_1 \qquad (r, _) = \blacksquare\ell \\ \mathcal{S}_1|^r_E\ f\ [e_i] \Downarrow^{\mathsf{w}} \ell; \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^\sigma | \mathcal{S}_2; \mathsf{K}' \\ \mathcal{S}_2|\mathsf{K}' \Downarrow^{\mathsf{k}} \psi; \mathcal{D}''|\mathcal{S}_3 \end{array}}{\mathcal{S} \Big|_E\ \kappa \hookrightarrow (f, [e_i], \blacksquare\ell) \Cup \mathsf{K} \Downarrow^{\mathsf{k}} \psi[\kappa \mapsto \ell]; (\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^\sigma) \boxtimes \mathcal{D}' \boxtimes \mathcal{D}'' \Big| \mathcal{S}_3}$$

Quiescence: $\gamma \Big| \mathsf{K} \Downarrow^{\mathsf{Q}} \mathcal{D}$

$$\frac{}{\gamma \Big| \{\} \Downarrow^{\mathsf{Q}} \{\}} \qquad \frac{}{\{\} \Big| \mathcal{D} \Downarrow^{\mathsf{Q}} \mathcal{D}} \qquad \frac{\mathcal{D}' = \{c/\gamma \mid c \in \mathcal{D}\}}{\gamma \Big| \mathcal{D} \Downarrow^{\mathsf{Q}} \mathcal{D}'}$$

Figure 4.11: Global cost semantics for the GPU.

ℓ for $\blacksquare\ell$ for each thread graph contained in the device box $\{c/\gamma \mid c \in \mathcal{D}\}$.[4] It is important to note that $\mathcal{D}'$ *need not* be fully quiesced at the end of this judgement—meta locations may still exist in $\mathcal{D}'$.

[4]Note that since our meta locations are each assigned a unique id, we do not have to worry about such things as capture-avoiding substitution, and can perform a simple search and replace operation on the thread graphs.

4.6 Scheduling GPU Cost Graphs

Since the HMM model that we are using as the abstract machine for the GPU imposes a particular schedule, we do not have as many options as we did for the scheduling of CPU cost graphs. However, scheduling GPU cost graphs is not entirely straightforward: while the HMM model provides us a model for the execution of each kernel on the GPU (and hence for the scheduling of each warp/thread), it does not provide us a model of execution in which work-items can enqueue other kernels to be executed on the GPU. In order to handle these device-side enqueuing operations of kernels, we need to impose a further scheduling policy that takes into account meta locations since these represent device-side enqueues of kernels.

4.6.1 Thread Schedules

Since the edges in our thread graphs which represent dependencies on other kernels are given by meta-edges we first formally define what it means for a device box $\mathcal{D}$ to not have any results from kernels outstanding.

Definition 4.6.1. *Given a device box $\mathcal{D}$, we say that $\mathcal{D}$ is* quiescent *and write* $\boxed{\mathcal{D}}$ *if for each edge $e_i \in \mathcal{D}$ e_i is not of the form $\blacksquare(\alpha_1, \alpha_2)$ for any α_1, α_2.*

Now, in order to determine which vertices in a thread graph are schedulable at a given step of a warp, given a set of nodes $\{\alpha_1, \ldots, \alpha_n\}$ we define the *schedulable nodes* of $\{\alpha_1, \ldots, \alpha_n\}$ or $S^{\mathcal{P}}(\overline{\alpha})$ to be given by the following:

$$S^{\mathcal{P}}(\overline{\alpha}) = \{\alpha \mid \alpha \neq \texttt{mem}[\blacksquare\ell], \alpha \in \overline{\alpha}\} \tag{4.8}$$

Recall from Section 2.1.1 that we defined the (*non-weighted*) schedule order $\trianglelefteq$ over the nodes α in a thread graph c as follows:

1. $\forall \alpha \in c \,.\, \alpha \trianglelefteq \alpha$; and

2. $\forall \alpha_1, \alpha_2 \in c \;.\; \alpha_1 \prec_c \alpha_2 \Rightarrow \alpha_1 \lhd \alpha_2$

where $\prec_c$ means that α_1 is a parent of α_2 in the thread graph c. Then given a set of thread graphs c_i in a warp graph W we can extend this relation on nodes to a relation on a set of nodes—the *considerable nodes*—and through this say when a given set of nodes $\overline{\alpha}_1$ can or need to be scheduled before another set of possible nodes $\overline{\alpha}_2$:

$$\overline{\alpha}_1 \trianglelefteq \overline{\alpha}_2 = \forall \alpha_1 \in \overline{\alpha}_1, \alpha_2 \in \overline{\alpha}_2 \;.\; \alpha_1 \trianglelefteq \alpha_2 \tag{4.9}$$

Furthermore, due to the nature of thread graphs and $\prec_c$ we have the following properties relating nodes in our thread graphs and the nodes in these $\overline{\alpha}$ and $S^{\mathcal{P}}(\overline{\alpha})$:

Theorem 4.6.1. *Let* $\mathsf{W} = \{c_1, \ldots, c_n\}$ *be a warp graph. Let* $\overline{\alpha}$ *be a set of considerable nodes, and* $S^{\mathcal{P}}(\overline{\alpha})$ *be its set of schedulable nodes for* W*. Then*

$$\forall c_j \in \mathsf{W} \;.\; \left|\{\alpha \mid \alpha \in \overline{\alpha}, \alpha \in c_j\}\right| \leq 1 \tag{4.10}$$

and therefore by the definition of $S^{\mathcal{P}}(\overline{\alpha})$ *that*

$$\forall c_j \in \mathsf{W} \;.\; \left|\{\alpha \mid \alpha \in S^{\mathcal{P}}(\overline{\alpha}), \alpha \in c_j\}\right| \leq 1 \tag{4.11}$$

This states that we have at most one node from each thread graph in the considerable nodes, and thence the schedulable nodes, at any given time. Importantly, we are not guaranteed to always have a node from every thread graph in these schedulable sets: if there is a meta-edge then the node will not be schedulable until it has been quiesced. From now on we will frequently write $S_i^{\mathcal{P}}$ for the set of schedulable nodes for a set of nodes $S(\overline{\alpha}_i)^{\mathcal{P}}$.[5]

There is a reason why we call these $S_i^{\mathcal{P}}$ *schedulable* nodes as opposed to scheduled nodes: since the GPU is a SIMD architecture, we must be running the same instruction at any given step of a warp W. Therefore when given schedulable nodes $S_i^{\mathcal{P}}$ we partition these into equivalence classes based upon instruction type while ignoring

[5]See Appendix B.4 for the proof of this theorem.

data-specific elements of the instruction: the equivalence relation ignores the kernel id and location for `kern` and `mem` instructions respectively.

Partitioning the nodes in $S_i^{\mathcal{P}}$ by instruction allows us to handle warp divergence easily since the warp needs to perform a step for each equivalence class of $S_i^{\mathcal{P}}$. Thus a set of schedulable nodes $S_i^{\mathcal{P}}$ are partitioned into a set of equivalence classes of *simultaneously scheduled nodes* $[S_i]_{\dot{\alpha}}$. The equivalence classes $[S_i]_{\dot{\alpha}}$ partition $S_i^{\mathcal{P}}$ and each equivalence class contains only nodes with the same instruction (up to the equivalence relation we mentioned above).

Once we have partitioned the schedulable nodes into equivalence classes of simultaneously scheduled nodes, each of these equivalence classes is run as a step of the warp. The order in which these run however, is unimportant: by Theorem 4.6.1 we have that at most one node from any one thread graph will be in $S_i^{\mathcal{P}}$, and hence in any of the equivalence classes. This then means that there are no edges between any of the nodes in $S_i^{\mathcal{P}}$ and hence between any of the equivalence classes. Thus the following equation holds:

$$\forall \alpha_1, \alpha_2 \in S_i^{\mathcal{P}} \ . \ \nexists e \in \mathcal{D} \ . \ e \equiv (\alpha_1, \alpha_2) \tag{4.12}$$

4.6.1.1 Dealing with Kernels & Meta-edges

When a thread graph node enqueues another kernel to execute on the GPU, this causes a meta-edge to be created in the thread graph along with meta vertices, that cause that thread graph to no longer be scheduled until the enqueued kernel has returned its result. Once the kernel has run, the meta location in the device box that corresponds to that kernel is quiesced. After this, the now-quiesced node will be included in the schedulable nodes for the enqueuing kernel.

Providing some more intuition behind this process, let $\texttt{mem}[\blacksquare\ell]$ be a node that is the source of a meta-edge , and let the execution of the nodes in a schedulable node set

$S_i^{\mathcal{P}}$ in a warp pool $\mathcal{W}$ resolve this location; $\blacksquare\ell \mapsto \ell$. Then after the scheduling of the nodes in $S_i^{\mathcal{P}}$, $S_{i+1}^{\mathcal{P}} = \emptyset$ and therefore $\mathcal{W} = (\kappa, \blacksquare\ell, \ell, \emptyset)$. At this point, a quiescence operation is performed on the entire device box $\mathcal{D}$ to replace all occurrences of the meta location $\blacksquare\ell$ with the location that it maps to. At this point the node $\mathtt{mem}[\blacksquare\ell]$ in question has been changed to $\mathtt{mem}[\ell]$, and the corresponding meta-edge has been removed. This node will then be added to the schedulable node set the next time the warp/thread arises in the schedule. The interaction between thread graphs and warp pools is detailed precisely in Section 4.6.3, however, before doing that we first need to make an important distinction between warps that cannot be run yet have more work left to do, and warps that have no more work to do.

4.6.1.2 Unrunnable vs. Finished Warps

Since we do not permit meta vertices to be included in our schedulable node set, the schedulable nodes $S_i^{\mathcal{P}}$ for a warp W at a step i being empty does not necessarily mean that all the threads in that warp have finished. Thus there is an important distinction that we need to make between a warp W being *unrunnable*—there are still nodes left in at least one of the threads graphs of W at that step, but all of them at the current step are meta vertices—and being *finished*—all nodes in all thread graphs in W have been traversed. The latter case is easy to handle: if we encounter a warp W that no longer has *any* nodes in it i.e., $\forall c_i \in \mathsf{W}, c_i \equiv \emptyset$ we mark the warp as a whole as no longer runnable by setting $\mathsf{W} = \emptyset$. In the case that all possibly schedulable nodes would be meta vertices we need to skip trying to run this warp, and try to find another warp in the warp pool to run instead. We can formalize these notions of unrunnable and finished warps with the following definition:

Definition 4.6.2. *Let* W *be a warp with considerable nodes* $\overline{\alpha}$ *and let* $S^{\mathcal{P}}(\overline{\alpha})$ *be its schedulable nodes. Then we say that* W *is:*

- *finished if*

$$\overline{\alpha} \equiv \emptyset$$

  and

- *unrunnable if*

$$(\overline{\alpha} \neq \emptyset) \land S^{\mathcal{P}}(\overline{\alpha}) \equiv \emptyset$$

4.6.2 Warp Schedules

Now that we have defined the scheduling of threads within each warp, we turn to the scheduling of each warp W_i that make up a warp pool $\mathcal{W}$. As opposed to the scheduling of thread graphs, these are much less complicated since we do not have to worry about meta locations, and the scheduling of these is dictated a priori by the HMM model that we are using for the GPU.

Recall that the HMM model requires that each warp W_i is scheduled in a round-robin fashion, thus if $\mathcal{W} = (\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_1, \ldots, \mathsf{W}_k \rangle^{\sigma})$ then we have that at warp-step n, warp W_l where $l = (n \mod k) + 1$ will run.

While it may be tempting to remove warps from the warp pool once they have finished, this is not a good idea from a bookkeeping standpoint. We instead assume the worst-case, and that these finished warps are *not* evicted from the warp pool once they have completed. Thus "running" a finished warp will take up one warp step.[6] Furthermore, since there is no way of determining if a given warp is unrunnable until it has been selected for running we charge the scheduling of an unrunnable or finished warp in a warp pool the same cost as that of a runnable warp.

As was discussed in the previous section, each step of a warp can actually be made up of a number of steps in the case where the warp has diverged. Thus we make a distinction between *thread steps* which are the steps that can run in parallel on the

[6] But there will be no other cost beyond this since there are no memory accesses etc.

warp, and *warp steps* which represent one step of a warp from the perspective of the warp pool. Thus a single warp step can be the result of multiple thread steps. However, since the thread steps that a warp at a given warp step i takes are defined by the number of equivalence classes of scheduled nodes $[S_i]_{\dot{\alpha}}$ for the schedulable nodes $S_i^{\mathcal{P}}$ at that warp step, the number of thread steps that will be made for any given warp step is bounded by the number of instructions that are encoded in the thread graphs.

To encode this relation we first add a "thread graph context" to warps that holds the current set of nodes under consideration for scheduling in each thread graph. These are then used to determine the set of nodes that run to progress from step i to $i+1$. Thus we define a warp W_k at step i as being given by a tuple

$$\mathsf{W}_k^i = (\overline{c}, \overline{\alpha})$$

where $\overline{c}$ are the thread graphs for the warp, and $\overline{\alpha}$ are the nodes in the thread graphs that should be considered for scheduling in the next step of the warp—i.e. the considerable nodes. It is important to recall that the nodes under consideration for the next step $\overline{\alpha}$ *are not* the same as the schedulable nodes $S^{\mathcal{P}}(\overline{\alpha})$: particularly, $\overline{\alpha}$ is permitted to contain meta locations whereas meta locations cannot occur in the schedulable nodes.

We can then introduce a stepping relation for warps based upon this

$$\mathsf{W}_k^i \rightrightarrows \mathsf{W}_k^{i+1}; S_i^{\mathcal{P}} \tag{4.13}$$

Which takes a warp W_k^i at a warp step i and performs thread steps for each thread-/instruction at that step of the warp, and returns back the schedulable nodes $S_i^{\mathcal{P}}$ that made up that warp step along with an updated list of nodes under consideration for scheduling in the next step. The rules for the warp step relation are given in Figure 4.12.

$$\frac{\{\overline{\alpha}\}_i \trianglelefteq \{\overline{\alpha}\}_{i+1} \qquad S_i^{\mathcal{P}} = S^{\mathcal{P}}(\{\overline{\alpha}\}_i) \\ \mathsf{W}_k^{i+1} = (\overline{c}, \overline{\alpha}_{i+1}) \qquad \mathsf{W}_k^i = (\overline{c}, \overline{\alpha}_i)}{\mathsf{W}_k^i \rightrightarrows \mathsf{W}_k^{i+1}; S_i^{\mathcal{P}}}$$

Figure 4.12: Definition of the warp step relation in terms of schedulable and scheduled nodes as defined in Section 4.6.1.

Using this, we then define the stepping relation for warp pools $\mathcal{W}(n)$. As opposed to the relation between warp- and thread steps, there is a one-to-one relation between warp steps and warp pool steps. Recall that a warp pool $\mathcal{W}(n)$ represents a kernel, and thus we need a way to determine when that kernel has completed. This happens precisely when all warps at their current step $\mathsf{W}_k^{i_k}$ in the warp pool are finished i.e.,

$$\forall\, \mathsf{W}_k^{i_k} \in \mathcal{W}\ .\ \mathsf{W}_k^{i_k} \equiv \emptyset$$

The relation takes a warp pool at a given step l along with a boolean flag b that says whether or not all of the warps in the warp box have finished or not, and returns the warp pool at the next step along with the nodes that have been run along with an updated boolean flag that is set to true if the warp that has been "run" is not finished. In the case where all warps in the warp pool $\mathcal{W}(n)$ have completed, $\mathtt{done}(\kappa, \ell, \blacksquare\ell)$ where κ is the kernel identifier, ℓ is the location of the final result, and $\blacksquare\ell$ is the meta location for the kernel is returned. We thus arrive at the following relation for our warp steps:

$$\mathsf{b}; \mathcal{W}(n)^i \Rrightarrow \left(\mathcal{W}(n)^{i+1}; S_i^{\mathcal{P}}; \mathsf{b}'\right) \uplus \mathtt{done}(\kappa, \ell, \blacksquare\ell) \tag{4.14}$$

and which is defined in Figure 4.14.[7]

[7]For ease of reading we will elide the explicit `left` and `right` embeddings in the rules.

$$\frac{\begin{array}{cc} k = n + (i \mod l) & \mathsf{W}_k^{i_k} \in \mathcal{W}(n)^i \\ \mathsf{W}_k^{i_k} \rightrightarrows \mathsf{W}_k^{i_k+1}; S_i^{\mathcal{P}} & b' = b \wedge (\mathsf{W}_k^{i_k+1} \equiv \emptyset) \end{array}}{\mathsf{b}; \mathcal{W}(n)^i \Rrightarrow \mathcal{W}(n)^{i+1}; S_i^{\mathcal{P}}; \mathsf{b}'} \; i \mod l \not\equiv 0$$

$$\frac{\begin{array}{cc} k = n + (i \mod w) & \mathsf{W}_k^{i_k} \in \mathcal{W}(n)^i \\ \mathsf{W}_k^{i_k} \rightrightarrows \mathsf{W}_k^{i_k+1}; S_i^{\mathcal{P}} & b' = \mathsf{W}_k^{i_k+1} \equiv (\overline{c}, \emptyset) \end{array}}{\mathsf{b}; \mathcal{W}(n)^i \Rrightarrow \mathcal{W}(n)^{i+1}; S_i^{\mathcal{P}}; \mathsf{b}'} \; i \mod l \equiv 0$$

$$\frac{\mathcal{W}(n)^i = \left(\kappa, \blacksquare\ell, \ell, \langle \mathsf{W}_n^{i_n}, \ldots, \mathsf{W}_k^{i_k} \rangle^{\sigma}\right)}{\mathsf{b}; \mathcal{W}(n)^i \Rrightarrow \mathtt{done}(\kappa, \ell, \blacksquare\ell)} \; (i \mod l \equiv 0) \wedge (\mathsf{b} \equiv \mathtt{true})$$

Figure 4.13: Definition of the warp pool step relation in terms of warp steps where $\mathcal{W}(n) = (\kappa, \ell, \blacksquare\ell, \langle \mathsf{W}_n, \ldots, \mathsf{W}_{l+n} \rangle^{\sigma})$.

4.6.3 Device Box Schedules

Since a device box consists of a number of different warp pools (one for each kernel currently executing) we also define a stepping relation that relates warp pool steps to device box steps—since these kernels execute in parallel, a device box step consists of one warp pool step for each warp pool currently executing in the device box. We also have to handle warp pool steps enqueuing kernels. This gives rise to the following stepping relation

$$\{\kappa_1, \ldots, \kappa_n\}; \mathcal{D} \twoheadrightarrow \mathcal{D}'; \{S_1^{\mathcal{P}}, \ldots, S_n^{\mathcal{P}}\}; \{\kappa_1, \ldots, \kappa_{n'}\} \tag{4.15}$$

Which takes the set of currently running warps $\{\kappa_1, \ldots, \kappa_n\}$ and the current device box, and returns back an updated device box in which each warp pool $\mathcal{W}(i)$ associated with κ_i in $\mathcal{D}$ is run and has steps given by $S_i^{\mathcal{P}}$. Along with this it returns the set of still active kernels at the end of the step $\{\kappa_i, \ldots, \kappa_{n'}\}$. Furthermore, in the rules we implicitly update the warp pools within the device box from one step to the next. The rules for this relation are given in Figure 4.14.

$$\frac{\begin{array}{c}\forall \kappa \in \{\overline{\kappa}\}, (\mathcal{W}(i)^j, \mathsf{b}) = \mathcal{D}(\kappa)\ .\ \mathsf{b}; \mathcal{W}(i)^j \Rrightarrow \mathcal{W}(i)^{j+1}; S_i^{\mathcal{P}}; \mathsf{b}' \\ \overline{\kappa''}, \gamma = \ \mathtt{unzip}\ \{(\kappa, \blacksquare\ell \mapsto \ell) \mid \mathtt{false}\ ; \mathcal{W}(i)^j \Rrightarrow \mathtt{done}(\kappa, \ell, \blacksquare\ell)\} \\ \overline{\kappa'''} = \{\kappa \mid \mathsf{kern}[\kappa] \in \bigcup_i S_i^{\mathcal{P}}\} \qquad \gamma \mid \mathcal{D} \Downarrow^{\mathsf{Q}} \mathcal{D}' \qquad \overline{\kappa'} = \overline{\kappa} \setminus \overline{\kappa''} \cup \overline{\kappa'''}\end{array}}{\overline{\kappa}; \mathcal{D} \twoheadrightarrow \mathcal{D}'; \{S_{i_1}^{\mathcal{P}}, \ldots, S_{i_n}^{\mathcal{P}}\}; \overline{\kappa'}}$$

Figure 4.14: Definition of the device box step relation in terms of warp pool steps. `unzip :: Set (a,b) -> (Set a, Set b)` takes a set of tuples and returns a tuple of sets.

It is crucial that this stepping relation return an updated device box, since once a kernel has finished—the warp pool that represents it has finished—a quiescence operation needs to be performed on the device box graph in order to unblock any other warps—or more specifically threads within the warps—that depend upon the results of that kernel. Thus once a kernel has finished running and we get a $\mathtt{done}(\kappa, \blacksquare\ell, \ell)$ result, we perform a quiescence operation with the meta location $\blacksquare\ell$ mapping to ℓ on the device box (as defined in Figure 4.11).

4.6.4 Keeping Track of the Region Stack

Recall that both region creation via the `letregion` form, and region expansion are not permitted in Λ^{G} (i.e., on the GPU). Thus, profiling programs written in Λ^{G} for space is a non-issue since we will always be using the same amount of space—the size of the region stack (that is copied over to the GPU) at the start of the computation.

In Section 3.6.5 we used the fact that region stack operations were encoded in the cost graphs that were built to define the region stack at any point in the schedule given by $\trianglelefteq$ and the closure of the current step V_i in the cost graph: $\mathcal{S}(S_i, \trianglelefteq)$. We might be tempted to do the same here. However, since we are not interested in profiling for different schedules on the GPU, and neither do we have to take into account region expansion and creation we do not need to be nearly as complex. Instead, we

can simply take the region stack at the end of building the cost graph, which we will denote by $\dot{\mathcal{S}}$, and use this throughout the cost analysis: since regions cannot be created, and since the `letregion` form is lexically scoped, the final region stack $\dot{\mathcal{S}}$ is a bounded evolution of the starting region stack (Section 3.6.2) and we don't have to worry about referencing non-existent regions in $\dot{\mathcal{S}}$ throughout any traversal of the graph. Furthermore, since regions can't be expanded and since the schedule is fixed, we don't need to think about profiling for space usage with respect to different schedules.

4.7 Determining Costs from GPU Cost Graphs

Now that we have defined the machinery to traverse the graph representing a given schedule of a GPU computation we need to turn our attention to calculating the cost of the computation. In this section we detail how to determine these costs for each step of the GPU computation, and thus the summation of these costs will give us the cost of the computation as a whole.

4.7.0.1 Step Traces

In order to calculate the costs of memory accesses and to take into account bank conflicts and broadcasts, we create a trace for each schedulable step $S_i^{\mathcal{P}}$ that occurs in a given device box step. However, since traces Σ (Figure 4.7) are defined as multisets of *addresses* a as opposed *locations* ℓ, and since the instructions (thread graph nodes) in $S_i^{\mathcal{P}}$ represent memory accesses as *locations* ℓ as opposed to *addresses* a, we need a function that determines the address of a location $\ell = (r, o)$ with respect to the final region stack $\dot{\mathcal{S}}$. We therefore define the function $\mathcal{A}_R(o)$ that, given a region R and an offset o into that region returns the (global) address in memory that the offset into the region corresponds to. This can be easily calculated as the base offset for the region R plus the offset o, and where $R = \dot{\mathcal{S}}(r)$.

We recall now that the schedulable nodes take the following form:

$$S_i^{\mathcal{P}} = \{\alpha_1, \ldots, \alpha_n\}$$

we can define the trace at step $S_i^{\mathcal{P}}$, or *step trace* $\Sigma(S_i^{\mathcal{P}})$ by taking the locations from the nodes in $S_i^{\mathcal{P}}$ that are memory accesses, and convert them to addresses:

$$\Sigma(S_i^{\mathcal{P}}) = \{a \mid \alpha_j \in S_i^{\mathcal{P}}, \alpha_j \equiv \mathtt{mem}(\ell), \ell = (r, o), a = \mathcal{A}_{\dot{\mathcal{S}}(r)}(o)\}$$

It is crucial that meta locations $\blacksquare\ell$ are not encountered during the creation of these step traces, since there is no way to determine whether the location that it would resolve to would lead to a bank conflict, broadcast, or neither. We therefore have the following theorem that ensures that we do not encounter meta locations during the building of traces:

Theorem 4.7.1. *Let $S_i^{\mathcal{P}}$ be a schedulable step. Then the following statement holds:*

$$\forall \alpha \in S_i^{\mathcal{P}} \,.\, \alpha \not\equiv \mathtt{mem}[\blacksquare\ell]$$

Proof. A straightforward consequence of Equation (4.8). □

4.7.0.2 Determining Bank Conflicts & Broadcasts

In order to calculate the cost of a trace, we first define a function $[-]_B$ that gives the memory bank that a given *address* a resides in. We define this function for both addresses $[a]_B$ and locations $[\ell]_B^{\dot{\mathcal{S}}}$ where we recall that w is the bank width for the HMM abstract machine model (and that $\dot{\mathcal{S}}$ is the final region stack for the program):

$$\begin{aligned} [a]_B &= a \mod w \\ [\ell]_B^{\dot{\mathcal{S}}} &= [\mathcal{A}_R(o)]_B \\ &\quad \text{where } \ell = (r, o) \text{ and } R = \dot{\mathcal{S}}(r) \end{aligned} \tag{4.16}$$

In order to determine whether a value is either written to local memory M (Figure 4.7) or global memory in allocation judgements, we also keep track of the *address range*(s) that correspond to the local memory in a *local memory state* L (defined in Figure 4.7). Thus when we seek to calculate the cost of performing an allocation or lookup in a region R, we first determine whether or not the address $\mathcal{A}_R(o)$ is within one of the address ranges for the local memory based upon the local memory state L[8] and charge the access cost of either global memory or local memory. We thus create a function $\mathsf{local}_v^a(L)$ which given a local memory state L, an address a and a value v determines whether or not the value v at address a is a local address with respect to the given local memory state L:

$$\begin{array}{rcl} \mathsf{local}_v^a((M, \{l_0, \ldots, l_k\})) & = & \exists i \ . \ a \in l_i \wedge a + |v| \in l_i \\ \text{where } l_i = [a_i, a_{i+1}] & & \end{array} \tag{4.17}$$

We likewise have a function $\mathsf{localaddr}^a(L)$ that given a local memory state L along with an address returns whether or not that address is local

$$\mathsf{localaddr}^a((M, \{l_0, \ldots, l_k\})) \quad = \quad \exists B_w(i) \in M \ . \ a \in B_w(i) \tag{4.18}$$

and from this we define the set of local addresses of a trace Σ with respect to L as

$$\mathsf{localaddrs}(L, \Sigma) = \{a \mid \mathsf{localaddr}^a(L), a \in \Sigma\} \tag{4.19}$$

We take as an invariant that if an address is not local, then it must be global.

Now that we have these functions, we can define what it means for a trace Σ to have a bank conflict, or a broadcast with respect to a local memory state L. We say that the trace Σ has has a bank conflict of order n (or an n-bank conflict) with respect to L and write $\mathbb{B}^n_{conf}(\Sigma, L)$ if

$$|\{a \mid \exists a' \not\equiv a \in \Sigma \ . \ [a]_B \equiv [a']_B, \mathsf{localaddr}^a(L)\}| = n \tag{4.20}$$

[8] Recall that our regions of memory are *contiguous* sections of memory, and thus can be represented this way.

or, in other words, if the threads in the warp that was run have requested different addresses from the same memory bank in the local memory. We can likewise say that Σ has a broadcast with respect to L and write $\mathbb{B}_{cast}(\Sigma, L)$ if

$$(\forall a_1, a_2 \in \Sigma \, . \, a_1 \equiv a_2) \wedge (\forall a \in \Sigma \, . \, \mathsf{localaddr}^a(L)) \tag{4.21}$$

i.e., if all of the threads have requested the same address from local memory.

Since the local memory state that we have is not like cache-memory on the CPU in that it is not subject to eviction or loading based on access patterns, the costs that we will calculate for a kernel are all with respect to a given starting state for the local memory L, which will be the set of regions that have been loaded into local memory before the kernel runs. This reflects best practice in GPU programming, where one first loads the memory that is needed (or at the very least, will be frequently accessed) into local memory before running the kernel itself. We therefore introduce a function $\mathsf{load}^n_{\mathcal{S}}(\{r_1, \ldots, r_n\})$ that given a region stack $\mathcal{S}$ along with a given set of region-names $\{r_1, \ldots, r_n\}$ attempts to load all of these regions into a local memory of size n. Since local memory is finite $\mathsf{load}^n_{\mathcal{S}}(\{r_1, \ldots, r_n\})$ is not guaranteed to load all of the regions specified, and we therefore make the simplifying assumption that either an *entire* region will be loaded into local memory, or none of the region will be (i.e. the load operates on the granularity of regions).

Once a given set of reads and writes have occurred producing a trace Σ, we then calculate the cost of the operations that resulted in Σ. This can be seen as being similar to the work of Ley-Wild, Acar, and Fluet [4] who define the cost of a self-adjusting computation based upon the trace distance of the heap-trace that is built up by the cost semantics (for an overview of this approach see Section 2.1.3). However, in our case we are not interested in the "distance" between two heap-traces

of the computation,[9] and instead interested in computing the cost based upon the heap-state that is represented by the trace Σ.

We therefore define the function $\mathsf{localcost}_{\mathcal{S}}(\Sigma, L)$ to be the function that given a local memory state L along with a trace Σ, and a region stack $\mathcal{S}$ returns a cost $c \in \mathbb{N}$ for the memory accesses recorded by the trace Σ. In order to calculate this, we first calculate the cost of all local memory accesses in Σ, as well as the cost of any bank conflicts:

$$\mathbb{B}^{n}_{conf}(\Sigma, L) + |\mathsf{localaddrs}(L, \Sigma)|$$

however, in the case where we have a broadcast $\mathbb{B}_{cast}(\Sigma, L)$ this cost is 1—since we have only accessed one local address. We therefore multiply this by 0 if a broadcast has taken place, and by 1 otherwise:

$$(\mathbb{B}^{n}_{conf}(\Sigma, L) + |\mathsf{localaddrs}(L, \Sigma)|)(1_{\{False\}}(\mathbb{B}_{cast}(\Sigma, L)))$$

and we then need to take into account all of the memory accesses to global memory in Σ, and since these are processed in a pipeline manner with latency l, the cost of all of the global memory accesses in Σ will be given by

$$l + |\Sigma \setminus \mathsf{localaddrs}(L, \Sigma)| - 1$$

We thus get the final definition of $\mathsf{localcost}_{\mathcal{S}}(\Sigma, L)$ as being given by the following formula:

$$\mathsf{localcost}_{\mathcal{S}}(\Sigma, L) = (\mathbb{B}^{n}_{conf}(\Sigma, L) + |\mathsf{localaddrs}(L, \Sigma)|)(1_{\{False\}}(\mathbb{B}_{cast}(\Sigma, L))) + l * |\Sigma \setminus \mathsf{localaddrs}(L, \Sigma)| \tag{4.22}$$

---

[9] Although what we are doing here could be considered a degenerate case: our trace operations consist solely of "operate on address a", and we are calculating the "trace" distance between the current heap ("trace") state and the starting heap ("trace").

4.7.0.3 Finalizing Costs

Now that we know how to calculate the cost of memory accesses for each step of a warp we also add in an abstract cost of 1 for each non-memory instruction in the schedulable step $S_i^{\mathcal{P}}$. We thus arrive at the following final definition of the cost for schedulable step $S_i^{\mathcal{P}}$ with respect to a local memory L:

$$\mathsf{cost}_{\Lambda^{\mathsf{G}}}\left(S_i^{\mathcal{P}}, L\right) = \mathsf{localcost}_{\dot{\mathcal{S}}}\left(\Sigma(S_i^{\mathcal{P}}), L\right) + \left(\sum_{\dot{\alpha} \neq \mathsf{mem}[\ell]} |[S_i]_{\dot{\alpha}}|\right) \tag{4.23}$$

From this, we can define the cost of a step of device box as a whole by taking the summation of the costs for each schedulable step $S_i^{\mathcal{P}}$ that is produced by the device box stepping relation, along with an abstract kernel scheduling cost κ_{sched} for each new kernel that is scheduled—or de-scheduled—during the step. Let a device box $\mathcal{D}$ take the following step:

$$\overline{\kappa}; \mathcal{D} \twoheadrightarrow \mathcal{D}'; \{S_1^{\mathcal{P}}, \ldots, S_n^{\mathcal{P}}\}; \overline{\kappa'}$$

and we define the (computational) cost of the step of the device box to be:

$$\begin{aligned} \mathsf{cost}_{\Lambda^{\mathsf{G}}}\left(\{S_1^{\mathcal{P}}, \ldots, S_n^{\mathcal{P}}\}, \mathbb{L}, \overline{\kappa}, \overline{\kappa'}\right) &= \left(\textstyle\sum_{j=1}^{n} \mathsf{cost}_{\Lambda^{\mathsf{G}}}\left(S_j^{\mathcal{P}}, L_j\right)\right) \\ &+ \kappa_{\mathsf{sched}} * \left|\overline{\kappa} \setminus \overline{\kappa'}\right| + \kappa_{\mathsf{sched}} * \left|\overline{\kappa'} \setminus \overline{\kappa}\right| \end{aligned} \tag{4.24}$$

where $\mathbb{L}$ is defined to be a mapping of kernel identifiers κ_i to the (a priori defined) local memory for that kernel L_i

$$\mathbb{L} = \{\kappa_1 \mapsto L_1, \ldots, \kappa_n \mapsto L_n\}$$

Then given the steps of the device box

$$\left(\overline{S_1^{\mathcal{P}}}, \overline{\kappa_1}, \overline{\kappa_1}'\right), \ldots, \left(\overline{S_n^{\mathcal{P}}}, \overline{\kappa_n}, \overline{\kappa_n}'\right),$$

We can define the total cost of the computation with respect to a local memory $\mathbb{L}$ on the GPU to be given by

$$\mathsf{cost}_{\mathsf{G}}(\mathcal{D}, \mathbb{L}) = \left(\sum_{i=1}^{n} \mathsf{cost}_{\Lambda^{\mathsf{G}}}\left(\overline{S_i^{\mathcal{P}}}, \mathbb{L}, \overline{\kappa_i}, \overline{\kappa_i'}\right)\right) \tag{4.25}$$

Determining space cost for the computation is trivial; since regions can neither be created nor expanded on the GPU, the space usage is simply the size of $\dot{\mathcal{S}}$, which is the same as the size of the starting region stack $\mathcal{S}$:

$$|\dot{\mathcal{S}}| = |\mathcal{S}| \tag{4.26}$$

We now move on to combining the cost semantics for both Λ_{C} and Λ^{G} to arrive at a cost semantics for heterogeneous parallel (functional) programming languages in the next chapter (Chapter 5), and then describe the implementation of this profiling framework along with a process for determining the abstract constant for kernel scheduling overhead later on in Chapter 6.

5

Cost Semantics for Heterogeneous Parallel Languages

Contents

5.1 Introduction **143**
5.2 Combining Λ_{C} and Λ^{G} **143**
5.2.1 Computational Worlds 144
5.2.2 Syntax 145
5.2.3 Syntax and Typing 145
5.3 Transferring Memory Between Computational Worlds . . **151**
5.4 Connecting Machines **156**
5.5 Cost Semantics for $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ **157**
5.5.1 Scheduling 159
5.5.2 Determining Costs 162
5.6 Wrapping Things Up **165**
5.6.1 Discussion: Effect Polymorphism 165
5.6.2 Discussion: Costing of Device Boxes 166

5.1 Introduction

In this chapter we turn to combining the two languages that we have defined—Λ_{C} for the CPU, and Λ^{G} for the GPU—to arrive at a parallel functional language $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ for both the CPU and GPU. However, in order to combine these two languages in a safe and reasonable manner we need to update Λ_{C} slightly and introduce the mechanisms with which we can glue together both of these languages statically, operationally, and determine the costs of moving from one computational setting to the other.

We start our journey towards combining these two languages by first defining the language's syntax (Section 5.2.2). After this we present the type system for the combined language and introduce syntactic restrictions in the type system over the operations permitted on the GPU (Section 5.2.3). We then link these two languages by defining memory transferance between the CPU and GPU computational worlds (Sections 5.3 and 5.4).

We then present the main result of this dissertation in Section 5.5: a cost semantics for profiling both time and space usage for programs in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ that use CPU and GPU parallelism and that is parametrized by the scheduling policy on the CPU, and local region allocation on the GPU . The primary definitions that are presented are functions for time- and space profiling using the cost semantics developed, and are defined in Equations (5.10) and (5.15) respectively. We then go on to briefly describe some of the restrictions and design decisions that we have made both for $\Lambda^{\mathsf{C}}_{\mathsf{G}}$, and for our cost graphs in Section 5.6.

5.2 Combining Λ_{C} and Λ^{G}

The syntax of $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ is (almost) exactly that of Λ_{C} on the CPU, and Λ^{G} on the GPU. However, these two languages are not the same: e.g., while Λ_{C} has the `letregion`

form Λ^{G} deliberately does not. We thus need a way to *statically* and syntactically restrict the code that is allowed to run on the GPU, as compared to the CPU. In order to do this we will use the notion of computational *worlds*, one—C—for the CPU and another—G—for the GPU, and use the type system to syntactically enforce these semantic restrictions.

5.2.1 Computational Worlds

Computational worlds are the static representation of the world on which a computation is run. However as it stands Λ_{C} does not contain any forms that would allow us to change either from one computational context to another. And, since our computations start out on the CPU, we need some way of moving computation to the GPU, and storing the result of that computation back on the CPU. This computational capability is provided by the addition of the following form to Λ_{C}:

$$\texttt{on } e \texttt{ at } \rho$$

This runs the expression e on the GPU and stores the result in the region ρ that resides on the CPU. This then allows us to migrate computation from the CPU to the GPU, as long as the expression e is a valid expression in Λ^{G}—which we need to ensure with the static semantics.

However, there is more that needs to be done than simply ensuring that e is a valid expression in Λ^{G} in an `on` expression: since we now have two worlds of computation, both with their own memory, and since we are using a region-typing system, the type system must also ensure that the region ρ into which the result of e will be allocated resides on the correct computational world—C. In order to do this, each region variable is annotated with the computational world on which the backing region resides. And, just as before, each region variable in the program is assumed to be unique, however now this uniqueness invariant is only up-to the world annotation

i.e., all region variables in Λ_{C} are unique, but many will deliberately overlap with region variables in Λ^{G}.

Just as before in Λ_{C} and Λ^{G}, each region resides in a region stack $\mathcal{S}$. Moreover, since region variables are unique up-to computational worlds, we will utilize name shadowing of backing region names when transferring from one computational world to another; all regions copied from one world to another retain their names, and since regions are only 'pushed' from the CPU to the GPU, every region variable on the GPU is unique and has the same region name as its parent region on the CPU. As we will see, this purposeful name shadowing between CPU and GPU region names in the region stacks makes the process of transferring computations from one world to another simpler later on.

5.2.2 Syntax

The syntax of $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ is (almost) exactly that of Λ_{C} on C, with the addition of the $\texttt{on}\ e\ \texttt{at}\ \rho$ form, and exactly Λ^{G} on G. Region variables ρ are now annotated with the world ω on which they reside. These changes are detailed in Figure 5.1. New language expressions are highlighted in gray, and expressions that can only be run on the host are underlined.

5.2.3 Syntax and Typing

We encode the device restrictions statically in the language through the type system. In particular, we statically disallow bad region accesses (i.e., cross-device region accesses and out-of-scope accesses), invalid computations, and invalid computational structures on G by enriching the region-typing system presented previously in Section 3.3, by annotating each region variable with the world on which the backing region must reside, and modifying the typing judgement to also track the current world

$i \in \mathbb{Z} \mid r \in \mathsf{RNames} \mid f, x \in \mathsf{Vars} \mid \zeta, \rho \in \mathsf{RVars} \mid o \in \mathbb{N}$

ρ_ω	$\in$	$RVars$	Annotated Region Variables
ω	$::=$	$\mathsf{C} \mid \mathsf{G}$	Computational Worlds
ℓ	$::=$	(r, o)	Region Locations
R	$::=$	$(\mathsf{ap}, s, \mathsf{refcnt}, \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\})$	Regions
$\mathcal{S}$	$::=$	$\overline{\rho} \mapsto \underline{R} \mid \mathcal{S}, r \mapsto R$	Ordered Domain/Region Stack
φ	$::=$	$\{\rho_\omega^1, \ldots, \rho_\omega^n\}$	Effect Sets
τ	$::=$	$\texttt{bool} \mid \texttt{int} \mid (\mu, \rho_\omega)$	Types
μ	$::=$	$\tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \texttt{Vec}\ \tau \mid \Pi\rho_\omega \geq \varphi.^{\varphi'}\tau$	Boxed Types
Ω	$::=$	$\{\underline{\rho_\omega}\} \mid \rho_\omega, \Omega$	World Contexts
Δ	$::=$	$\cdot \mid \Delta, \rho_\omega \geq \varphi$	Region Constraints
Γ	$::=$	$\cdot \mid \Gamma, x : \tau$	Typing Contexts
e	$::=$	$x \mid i \mid \texttt{True} \mid \texttt{False} \mid e_1 < e_2 \mid e_1 * e_2$	Terms
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f$	
	$\mid$	$\lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho_\omega \mid e_1\ e_2$	
	$\mid$	$\lambda\zeta \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho_\omega \mid e[\rho_\omega]$	
	$\mid$	$(e_1, e_2)\ \texttt{at}\ \rho_\omega \mid \texttt{fst}\ e \mid \texttt{snd}\ e$	
	$\mid$	$\texttt{fix}\ f : \tau.u \mid e_1 \Vert e_2\ \texttt{at}\ \rho_\omega$	
	$\mid$	$[e_1, \ldots, e_n]\ \texttt{at}\ \rho_\omega \mid \texttt{parmap}\ e\ [e_1, \ldots, e_n]\ \texttt{at}\ \rho_\omega$	
	$\mid$	$\underline{\texttt{letregion}\ \rho_{\mathsf{C}}\ \texttt{in}\ e} \mid \underline{\texttt{on}\ e\ \texttt{at}\ \rho_\omega}$	
u	$::=$	$\lambda x : \tau.^{\varphi} e\ \texttt{at}\ \rho_\omega \mid \lambda\zeta \geq \varphi'.^{\varphi} u\ \texttt{at}\ \rho_\omega$	Abstractions
v	$::=$	$i \mid b \mid (v_1, v_2) \mid [v_1, \ldots, v_n]$	Values
	$\mid$	$\texttt{clos}\ (x : \tau, e, E) \mid \texttt{rclos}\ (\rho_\omega, \varphi, \varphi', u) \mid \ell$	

Metafunctions (extended)

$$
\begin{aligned}
\mathsf{fv}\,(e) &= \ldots (\textit{Figure}\ 3.2) \\
\mathsf{fv}\,(\texttt{on}\ e\ \texttt{at}\ \rho_\omega) &= \mathsf{fv}\,(e) \\
\\
\mathsf{frv}\,(e) &= \ldots (\textit{Figure}\ 3.2) \\
\mathsf{frv}\,(\texttt{on}\ e\ \texttt{at}\ \rho_\omega) &= \mathsf{frv}\,(e) \cup \{\rho_\omega\}
\end{aligned}
$$

Figure 5.1: The syntax of $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ with the syntactic changes to Λ_{C} highlighted.

on which an expression is being typechecked. Using world-annotated typing judgements and world-annotated region variables, we disallow both off-current-world region accesses and invalid device-specific computational (world) structures.

The manner in which we amend the type system that we saw earlier in Chapter 3 is important, but largely straightforward. Recall that the typing judgement introduced there for expressions $e \in \Lambda_{\mathsf{C}}$ took the form:

$$\Omega; \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$$

which was read "expression e has type τ with effect φ in contexts Ω, Δ, and Γ." However, since we now need to ensure that regions that are encountered while typechecking reside on the appropriate computational world, we amend this judgement so that it tracks the current computational world:

$$\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e : \tau, \varphi$$

which is read "expression e has type τ and effect φ *on computational world* ω in contexts Ω, Δ, and Γ."[1] As opposed to earlier, each region variable is now explicitly 'colored' with the world on which its backing region resides—ρ_{ω}—and thus we do not have explicit kind judgements in our typing rules.[2]

We can then define the rules for this typing judgement as a straightforward modification of the typing rules in Section 3.3.4 for Λ_{C}. This modified type system is presented in Figures 5.2 and 5.3, and new typing rules are highlighted in grey.[3] The only two non-straightforward forward modifications are the additional rules for the additional `on` expression, and the removal of the rule for `letregion` in the case that ω is G.

---

[1]Recall from Section 3.3.4 that Ω is the current set of valid computational regions, Δ is the current set of region constraints, Γ is a mapping from term variables to types, and φ is a set of bounding effects for e.

[2]This is since we don't treat these 'world colors' as full kinds, but track it as meta-information in the type system.

[3]Since our regions are now annotated with worlds, we will index them by superscripts instead of subscripts so as to avoid multiple subscripts.

$$\boxed{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e : \tau, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} i : \texttt{int}, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 : \texttt{int}, \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_2 : \texttt{int}, \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 * e_2 : \texttt{int}, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 : \texttt{int}, \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_2 : \texttt{int}, \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 < e_2 : \texttt{bool}, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \texttt{False} : \texttt{bool}, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \texttt{True} : \texttt{bool}, \varphi}$$

$$\frac{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_b : \texttt{bool}, \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_t : \tau, \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_f : \tau, \varphi}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \quad x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) = \tau}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} x : \tau, \varphi}$$

$$\frac{\Omega; \Delta; \Gamma, x : \tau_1^{\omega} \vdash_{\text{exp}} e' : \tau_2, \varphi' \quad \Omega \vdash_{\text{place}} \rho_{\omega} \quad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_{\omega}}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \lambda x : \tau_1.^{\varphi'} e'\ \texttt{at}\ \rho_{\omega} : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_{\omega}), \varphi}$$

$$\frac{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_{\omega}), \varphi \quad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_{\omega} \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_2 : \tau_1, \varphi \quad \Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1\ e_2 : \tau_2, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega; \Delta; \Gamma; \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_1 : \tau_1, \varphi \quad \Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e_2 : \tau_2, \varphi \quad \Omega \vdash_{\text{place}} \rho_{\omega} \quad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_{\omega}}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} (e_1, e_2)\ \texttt{at}\ \rho_{\omega} : (\tau_1 \times \tau_2, \rho_{\omega}), \varphi}$$

$$\frac{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho_{\omega}), \varphi \quad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_{\omega}}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \texttt{fst}\ e : \tau_1, \varphi}$$

$$\frac{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho_{\omega}), \varphi \quad \Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho_{\omega}}{\Omega; \Delta; \Gamma^{\omega} \vdash_{\text{exp}} \texttt{snd}\ e : \tau_2, \varphi}$$

Figure 5.2: Modified typing rules for $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ based on the type system for Λ_{C} given in Figure 3.9.

$$\frac{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e_i : \tau, \varphi \qquad \Omega \vdash_{\text{place}} \rho_\omega \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} [e_1,\ldots,e_n] \;\texttt{at}\; \rho_\omega : (\,\texttt{Vec}\;\tau, \rho_\omega), \varphi}$$

$$\frac{\rho_\omega, \Omega;\Delta, \zeta \geq \varphi'';\Gamma^{\omega} \vdash_{\text{exp}} u' : \tau, \varphi' \qquad \Omega \vdash_{\text{place}} \rho_\omega \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} \lambda\,\zeta \geq \varphi''.^{\varphi'} u' \;\texttt{at}\; \rho_\omega : (\Pi \zeta \geq \varphi''.^{\varphi'} \tau, \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e : (\Pi \rho_\omega \geq \varphi''.^{\varphi'} \tau, \rho^1_\omega), \varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho^1_\omega \qquad \Omega \vdash_{\text{place}} \rho^2_\omega \qquad \Omega;\Delta \vdash_{\text{re}} \rho^2_\omega \geq \varphi'' \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi[\rho^2_\omega/\rho_\omega] \supseteq \varphi'[\rho^2_\omega/\rho_\omega]}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e[\rho^2_\omega] : \tau[\rho^2_\omega/\rho_\omega], \varphi}$$

$$\frac{\Omega;\Delta;\Gamma, f : \tau^{\omega} \vdash_{\text{exp}} u : \tau, \varphi}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} \texttt{fix}\, f : \tau.u : \tau, \varphi}$$

$$\frac{\vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\varphi \qquad \Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e_1 : \tau_1, \varphi_1 \qquad \Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e_2 : \tau_2, \varphi_2 \qquad \Omega \vdash_{\text{place}} \rho_\omega \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho_\omega \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_2}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e_1 \| e_2 \;\texttt{at}\; \rho_\omega : (\tau_1 \times \tau_2, \rho_\omega), \varphi}$$

$$\frac{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho'_\omega), \varphi \qquad \Omega \vdash_{\text{place}} \rho'_\omega \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi' \qquad \Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} e_i : \tau_1, \varphi \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho'_\omega \qquad \Omega;\Delta \vdash_{\text{er}} \varphi \ni \rho}{\Omega;\Delta;\Gamma^{\omega} \vdash_{\text{exp}} \texttt{parmap}\, e\, [e_1,\ldots,e_n] \;\texttt{at}\; \rho : (\,\texttt{Vec}\;\tau_2, \rho), \varphi}$$

$$\frac{\Omega;\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Omega;\Delta;\Gamma;\{\rho^1_{\mathsf{C}},\ldots,\rho^n_{\mathsf{C}}\} \qquad \rho_{\mathsf{C}}, \Omega;\Delta, \rho_{\mathsf{C}} \geq \{\rho^1_{\mathsf{C}},\ldots,\rho^n_{\mathsf{C}}\};\Gamma^{\mathsf{C}} \vdash_{\text{exp}} e : \tau, \{\rho^1_{\mathsf{C}},\ldots,\rho^n_{\mathsf{C}}, \rho_{\mathsf{C}}\}}{\Omega;\Delta;\Gamma^{\mathsf{C}} \vdash_{\text{exp}} \texttt{letregion}\, \rho_{\mathsf{C}} \;\texttt{in}\, e : \tau, \{\rho^1_{\mathsf{C}},\ldots,\rho^n_{\mathsf{C}}\}}$$

$$\frac{\{\rho^i_{\mathsf{C}}\} = frv(e) \qquad \Omega \vdash_{\text{place}} \rho_{\mathsf{C}} \qquad \rho_{\mathsf{G}}, \{\rho^i_{\mathsf{G}}\};\rho^i_{\mathsf{G}} \geq \{\rho_{\mathsf{G}}\};\Gamma^{\mathsf{G}} \vdash_{\text{exp}} e : (\tau, \rho_{\mathsf{G}}), \varphi' \qquad \Omega;\Delta \vdash_{\text{er}} \varphi'[\mathsf{C}/\mathsf{G}] \ni \rho_{\mathsf{C}} \qquad \Omega;\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'[\mathsf{C}/\mathsf{G}]}{\Omega;\Delta;\Gamma^{\mathsf{C}} \vdash_{\text{exp}} \texttt{on}\, e \;\texttt{at}\; \rho_{\mathsf{C}} : (\tau, \rho_{\mathsf{C}}), \varphi}$$

Figure 5.3: Modified typing rules for $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ based on the type system given for Λ_{C} in Figure 3.10.

In the typing rule for the `on` expression the substitution $\varphi[\omega'/\omega]$ substitutes each region variable $\rho_\omega \in \varphi$ with the same region variable $\rho_{\omega'}$ with world kind ω'. This rule then enforces that there are no cross-world accesses, and that $e \in \Lambda^{\mathsf{G}}$.

5.2.3.1 Syntactic Inclusion

Due to the fact that we do not allow creation of regions in Λ^{G}, computations that run on the GPU along with the memory those computations require, must be 'pushed' to the GPU from the CPU, and later 'pulled' back to the CPU from the GPU. Furthermore, we have a (trivial) syntactic inclusion of Λ^{G} in Λ_{C}. This leads to the following theorem:

Theorem 5.2.1 (Syntactic Inclusion)**.** *Let e be a closed, well-formed expression in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$. Then the following statements hold:*

- *if $\Omega; \Delta; \Gamma^{\mathsf{G}} \vdash_{exp} e : \tau, \varphi$ then $e \in \Lambda^{\mathsf{G}}$;*
- *if $\Omega; \Delta; \Gamma^{\mathsf{C}} \vdash_{exp} e : \tau, \varphi$ then $e \in \Lambda_{\mathsf{C}}$;*
- *if $e \in \Lambda^{\mathsf{G}} \implies e \in \Lambda_{\mathsf{C}}$ (up-to 're-coloring' of region variables).*

This, along with the computational model that we have built gives rise to a natural subtyping relationship between worlds: any valid program over G is a valid program over C, and this is reflected in (3). We will also abuse terminology a bit and will frequently say Λ_{C} to mean those parts of the computation that run on the CPU, and Λ^{G} to mean those parts of the computation that run on the GPU, and $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ to refer to those portions that reside on both and can be moved "back and forth".[4]

[4]See Appendix B.5 for the proof of this theorem.

5.3 Transferring Memory Between Computational Worlds

In the type system, we modeled memory transfers syntactically by changing the world kind of the region variables within an `on` form. However, this will not work operationally and we need to pin down how memory transfers work in this context. In this section we do exactly this, and detail the handling of memory transfers operationally while laying the ground work for calculating the cost of these operations.

While it may be optimal to lazily transfer regions across as they are needed by the GPU section of code [9], doing so complicates the reasoning for memory transfer costs in our semantics. We therefore take the simpler operational view that data is copied and allocated immediately for each region that is needed by the expression that is to be run on the GPU when an `on` call is encountered. Furthermore, we assume that these regions are immediately deallocated upon completion of the `on` form and after the requisite copies from device to host memory are performed.[5] Since allocating buffers on the GPU is relatively inexpensive as compared to the cost of transferring region contents from host to device memory, we are not overly concerned with coalescing memory transfers and trying to not deallocate data that may just be copied back later. However, this is an interesting and perhaps valuable direction for future work.

We copy regions from the host to the device and vice-versa in a piecewise fashion; if e is an expression that uses (backing) regions R_1, R_2 and R_3, then we will generate a separate copy call[6] for each separate region at the time that the `on` form is encountered. Since this copy is done 'pointwise' with respect to each region in the

[5] Thus we cannot rely on any region pre-existing on the GPU when determining which regions need to be transferred.

[6] Such as `cudaMemPrefetchAsync`.

region stack, we not only have that we will preserve the same lattice structure on the GPU as we had on the CPU between the copied regions, but also that the composition of copies will preserve the region structure that we have from $:\geq$ i.e., $:\geq$ is a bounded join semilattice on regions, and the copy functions create a lattice homomorphism. This preservation of lattice structure throughout the memory transfer process is critical to the safety of our language: if we lost this lattice structure, our regions could no longer reflect the liveness properties that are needed at the type level of the program—in other words, it would allow well-typed programs to (possibly) go wrong. The fact that preservation of the term-level ancestor relationship ($:\geq$) preserves the type-level outlives ($\geq$) relationship that we require and vice-versa was formalized in Theorem 3.6.4, however, we now add the additional following statement to this theorem in order to account for multiple computation worlds:

> *R_1 and R_2 are both regions on the same computational world ω.*

to arrive at the following statement of the theorem in the heterogeneous case:

Theorem 5.3.1. *Let $\rho_\omega, \rho'_\omega \in RVars$, and e an expression such that $\{\underline{\rho}_\omega\}; \cdot; \cdot^\omega \vdash_{exp} e : \tau, \varphi$, and $\rho_\omega, \rho'_\omega \in \varphi$. Now take the following derivation tree for e:*

$$\dfrac{\dfrac{\dfrac{\vdots}{E; \mathcal{S}_1; e' \Downarrow v'; \mathcal{S}_2; c}\ (\star) \qquad \vdots}{\vdots}}{\{\}; \{r_h \mapsto R_H\}; e \Downarrow v; \mathcal{S}'; c'} \tag{5.1}$$

such that $\rho_\omega, \rho'_\omega \in frv(e')$, and $\mathcal{S}_1(E(\rho_\omega)) = R_1$, $\mathcal{S}_1(E(\rho'_\omega)) = R_2$. Then we have that $\mathcal{S}_1 \vdash R_1 :\geq R_2$ iff $\Omega; \Delta \vdash_{rr} \rho \geq \rho'$, and that both R_1 and R_2 reside on ω. And, where Ω and Δ are constructed by the following derivation:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Omega; \Delta; \Gamma^\omega \vdash_{exp} e' : \tau', \varphi'}\ (\star\star) \qquad \vdots}{\vdots}}{\{\underline{\rho}\}; \cdot; \cdot^\omega \vdash_{exp} e : \tau, \varphi} \tag{5.2}$$

Furthermore, we have for E and $\mathcal{S}_1$ as in ($\star$) and Ω as in ($\star\star$) that

$$proj(E); \mathcal{S}_1 \vdash_{vreg} \Omega$$

Now that we are transferring regions of memory to- and from device memory another critical question to ask is whether regions that have possibly evolved in parallel on both CPU and GPU can have 'merge conflicts' when the GPU region is copied back to its parent CPU region. Luckily, due to immutability of the data in our regions, we can prove that conflicts do not arise in device-to-host transfers. In order to formalize the lack of merge conflicts however, we first need to define the process by which regions are merged between the device and host:

Definition 5.3.1 (Region Merging)**.** *Let R be a region on C and R' be a copy of R on G. Take $R_1 = (\mathsf{ap}_1, s_1, \mathsf{refcnt}_1, v_1)$ and $R'_1 = (\mathsf{ap}'_1, s'_1, \mathsf{refcnt}'_1, v'_1)$ to be (possibly empty) evolutions of R and R' respectively. Then we define the* merge *R_2 of R_1 and R'_1 with cost δ, written$R_1; R'_1 \rhd R_2; \delta$ as:*

$$\frac{\delta = \mathsf{cost}_{\mathsf{merge}}(R_1, R'_1) \qquad R_2 = \underset{\mathsf{G}\to\mathsf{C}}{\mathsf{memcpy}}(R'_1 \setminus R_1, \mathsf{ap}_1)}{R_1; R'_1 \rhd R_2; \delta}$$

where $\underset{\mathsf{G}\to\mathsf{C}}{\mathsf{memcpy}}$ is a native way of transferring memory from device to host and $\mathsf{cost}_{\mathsf{merge}}(R_1, R'_1)$ is a primitive cost function that tells us the cost of merging or 'splicing' R'_1 into the R_1 region. How this PCP for region splicing is determined is given in Section 6.4.

Figure 5.4 illustrates the process by which regions are merged. However, it is crucial to note that the cost returned by this merge judgement does *not* take into account the cost of the $\underset{\mathsf{G}\to\mathsf{C}}{\mathsf{memcpy}}$ of the region from the GPU to the CPU—this will be done by a series of functions that are responsible for the movement of a *set* of regions from one world to another since we want to determine the cost of the entire transfer, and not the cost of each individual transfer.

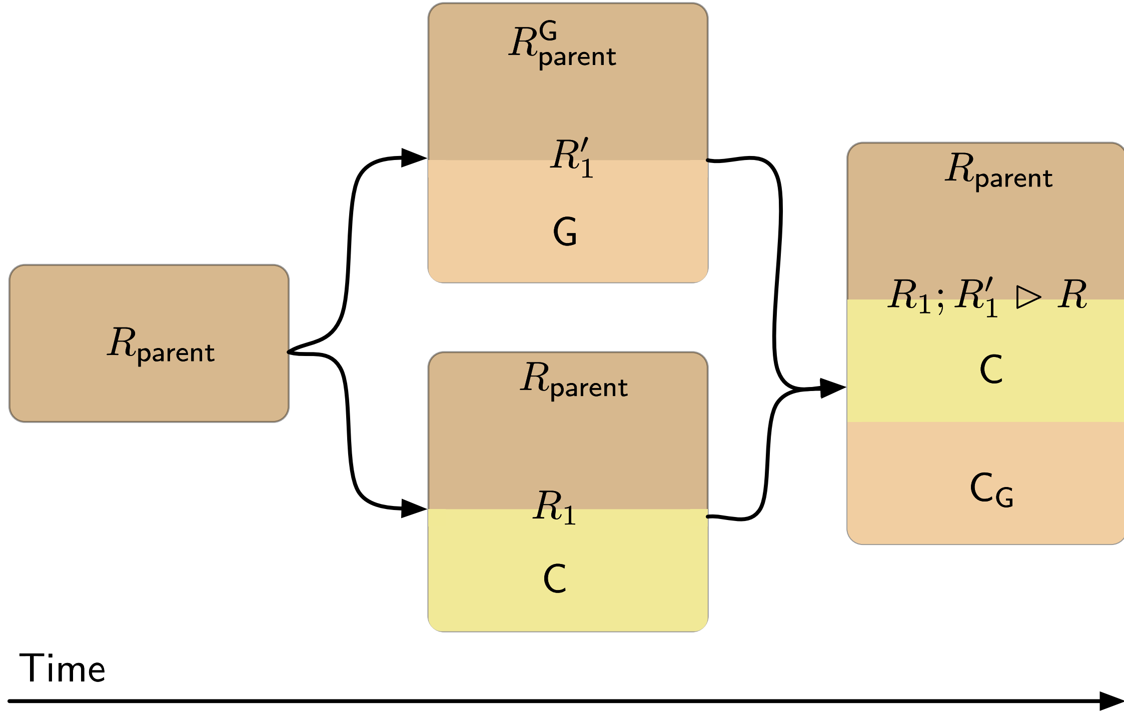


Figure 5.4: The process by which a region (R'_1) that resided on G is merged back in with its parent region (R_1) that resides on C, and where both R_1 and R'_1 evolved from (are bounded evolutions of) a parent region R_{parent} at the time the region was copied to the GPU.

We likewise define a function $\mathsf{traceMerge}_R^{R'}(\ell)$ as:

$$\begin{aligned} &\mathsf{traceMerge}_R^{R'}(\ell) = (o', r) \\ &\quad \text{where } R = (\mathsf{ap}, _, _, _), \ell = (o, r) \\ &\quad\; \mathsf{ap}' \text{ is the allocation pointer at the time the region was copied to the GPU} \\ &\quad\; \text{and } o' = \mathsf{ap} + (o - \mathsf{ap}') \end{aligned} \tag{5.3}$$

that given a location ℓ in the region R' that resides on G, and R''s parent region R on C returns the new location for ℓ in the merged region (residing on C).

Using this definition of region-merging, we can formalize merge-conflict freedom in the device-host transfer process:

Theorem 5.3.2 (Merge Conflict Freedom). *Let R_p be the parent region for R_1 (on C) and R'_1 (on G), and $R_1; R'_1 \triangleright R; \delta$. Then* $\mathsf{range}(R_1) \subseteq \mathsf{range}(R) \wedge \mathsf{range}(R'_1) \subseteq \mathsf{range}(R)$ *and* $\forall o \in \mathsf{dom}(R_1) \cap \mathsf{dom}(R'_1) . v_1(o) \equiv v_2(o)$.

Proof. By definition of $R_1; R'_1 \rhd R; \delta$ and by immutability of region memory. See Appendix B.6 for a detailed proof. □

We now turn to the process of transferring memory. To this end we define two functions: one that allows us to 'push' a set of regions to the GPU, and the other that allows us to pull a set of regions back and merge them with their parent region on the CPU:

$$\begin{array}{l} \mathsf{push}\left(\{r_1 \to R_1, \ldots, r_n \mapsto R_n\}\right) \\ \mathsf{pull}_{\mathcal{S}_\mathsf{C}}\left(\{r_1 \mapsto R'_1, \ldots, r_n \mapsto R'_n\}\right) \end{array}$$

And these can each be defined separately as

$$\mathsf{push}\left(\{r_i \mapsto R_i\}_{i=1}^n\right) = \left(\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\{R_i\}_{i=1}^n), \{r_i \mapsto R'_i \mid R'_i = \underset{\mathsf{C}\to\mathsf{G}}{\mathsf{memcpy}}(R_i, o_i)\}\right) \tag{5.4}$$

where o_i is the base offset for region R_i. We can likewise define $\mathsf{pull}\,()$ to be the following

$$\begin{array}{rl} \mathsf{pull}_{\mathcal{S}_\mathsf{C}}\left(\{r_i \mapsto R'_i\}\right) & = \left(\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\{R'_i\}_{i=1}^n) + \sum_{i=1}^n \delta_i, \mathcal{S}_\mathsf{C}\{r_i \mapsto R''_i\}\right) \\ & \text{where } R_i; R'_i \rhd R''_i; \delta_i, R_i = \mathcal{S}_\mathsf{C}(r_i)\} \end{array} \tag{5.5}$$

where the R'_i reside on G and R_i and R''_i reside on C. For both of these functions, the cost of the memory transfer from one computational world to the other along with a region stack is returned. In the case of $\mathsf{push}\,()$ the resulting region stack is new and resides on G , and in the case of $\mathsf{pull}_{\mathcal{S}_\mathsf{C}}\left(\overline{r \mapsto R'}\right)$, it results in an update to the region stack $\mathcal{S}_\mathsf{C}$ in which each backing region $R_i \in \overline{R}$ (each of which already reside on C) has been merged with its child region $R'_i \in \overline{R'}$ that resided on G.

The functions $\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R})$ and $\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R})$ are both PCPs, and their determination is externally defined, and the costs learned from the implementation.[7] Their definition, and the determination of appropriate costing for them is detailed along with our other PCPs in Section 6.4.

Now that we've defined how to reason about transferring memory between our different computational worlds, we move to connecting the two computational worlds operationally.

[7]i.e., they are implementation dependent.

5.4 Connecting Machines

Connecting the machines is quite straightforward at this point, however we go into the precise details of this connection in the underlying implementation later on in Chapter 6. In particular, since we assume implementation-specific memory transfer functions $\underset{C \to G}{\mathsf{memcpy}}$ and $\underset{G \to C}{\mathsf{memcpy}}$ (e.g., memory transfer functions from either openCL [57], or CUDA [56]), the actual connections between the two machine models are straightforward: both from a semantic and cost standpoint we view the underlying memory transfer functions as extrinsic ways of acting on memory and machines: operationally, they provide us a way to move a set of regions from one computational world to another, and we can observe how much they cost via our (implementation specific) primitive cost providers (Section 6.4) for those transfers. Thus we do not worry about how the two machine models interact with one another beyond the fact that when we transfer memory from C to G (or pull from G to C), the data that is transferred will reside in global memory, some of it will be placed on local memory on the GPU, and we have a way to measure how much this process of memory transference costs. In a sense, each computational world is a black-box to the other, with the only means of communication being through the `on` form—through which a computation is sent, and the result returned.

However, during this process of transferring memory from the CPU to the GPU, the question of determining which regions should be placed in the local memory for the kernel can have a significant impact on the performance of the kernel. In this case, we exploit the fact that regions cannot be expanded once they have been transferred to the GPU—and hence before they are loaded into local memory—and hence cannot outgrow the local memory that is available once they have been placed. Thus once a region has been loaded into local (work-group) memory at the start of a kernel, the entire region remains in local memory for the life of the kernel computation. We go into detail on loading strategies for local regions in Section 5.5.1.1.

5.5 Cost Semantics for $\Lambda_{\mathsf{G}}^{\mathsf{C}}$

Since we have already defined the cost semantics for both the CPU and GPU portions of the language in Chapters 3 and 4 respectively, we only need to detail how these two interact with each other—through the `on` form.

Updating Cost Graphs Since the `on` form pushes the computation performed by e to the GPU, and the rest of the thread of execution must wait until e has finished executing before it can proceed we need a way of representing the device box $\mathcal{D}$ that is built by the `on` form in the cost graphs for the CPU portion of $\Lambda_{\mathsf{G}}^{\mathsf{C}}$. In order to allow this, we amend the grammar for the nodes in the cost graphs from Figure 3.15 with an additional node structure that encodes a device-box computation along with preserving pieces of the semantic context that will be needed later on when scheduling and costing device boxes:

$$\tilde{\alpha} \quad ::= \quad \alpha \;\mid\; (\alpha, \mathcal{D}, \kappa, e, E, \ell) \tag{5.6}$$

and where device box nodes $(\alpha, \mathcal{D}, \kappa, e, E, \ell)$ are treated opaquely by the graph combinators. However, while device-box nodes are treated opaquely by the CPU graph combinators they cannot be treated opaquely by the scheduling policy over which will need to be updated in Section 5.5.1.

Since the GPU cost semantics takes an expression that generates a kernel within the `on` expression, the entry point for creating device boxes is through the cost rules for dealing with enqueued kernels on the GPU (Section 4.5.4.2):

$$\mathcal{S}|\mathsf{K} \Downarrow^{\mathsf{k}} \psi; \mathcal{D}|\mathcal{S}_1$$

thus when we encounter an `on` expression in our cost semantics, we need to build the region stack for the GPU computation $\mathcal{S}$ and add the kernel to the kernel pool

K, at which point we then use the rules from Section 4.5 to generate the device box. With this in mind, we arrive at the following definition for the `on` -rule:

$$\frac{\begin{array}{c}\overline{\rho_{\mathsf{C}}} = frv(e) \qquad \overline{r_{\mathsf{C}}} = E\left(\overline{\rho_{\mathsf{C}}}\right) \qquad \overline{R_{\mathsf{C}}} = \mathcal{S}\left(\overline{r_{\mathsf{C}}}\right) \\ (\delta_{\mathsf{push}}, \mathcal{S}_{\mathsf{G}}) = \mathsf{push}\left(\overline{r_{\mathsf{C}} \mapsto R_{\mathsf{C}}}\right) \\ \mathcal{S}_{\mathsf{G}}|\kappa \hookrightarrow (f, [e_i], \blacksquare\ell) \Downarrow^{\mathsf{k}} \psi; \mathcal{D}|\{\overline{r_{\mathsf{C}} \to R'}\} \\ \mathcal{S} \rightsquigarrow \mathcal{S}' \qquad (\delta_{\mathsf{pull}}, \mathcal{S}'_{\mathsf{C}}) = \mathsf{pull}_{\mathcal{S}'}\left(\overline{r_{\mathsf{C}} \mapsto R'}\right) \\ \ell_{\mathsf{G}} = (o, r) = \psi(\kappa) \qquad \ell = \mathsf{traceMerge}^{\{\overline{r_{\mathsf{C}} \to R'}\}(r)}_{\mathcal{S}'(r)}(\ell_{\mathsf{G}})\end{array}}{E; \mathcal{S};\ \mathtt{on}\ e\ \mathtt{at}\ \rho_{\mathsf{C}} \Downarrow \ell; \mathcal{S}'_{\mathsf{C}}; \alpha_1 \oplus_{\delta_{\mathsf{push}}} (\alpha_2, \mathcal{D}, \kappa, e, E, \ell) \oplus \alpha_3 \oplus_{\delta_{\mathsf{pull}}} \alpha_4} \ (\kappa, \alpha_i \text{ fresh}) \tag{5.7}$$

In this rule we first collect and transfer all the regions that the expression e that is to run on the GPU requires (line 1) after which we build the device box for the computation (line 2), and then merge the region stack on the GPU at the end of the computation with a bounded evolution of the region stack on the CPU—since some other thread on the CPU could be modifying the CPU region stack while the GPU computation is running (line 3). Finally, on line 4, the location on the CPU where the result of the kernel is stored is calculated using the $\mathsf{traceMerge}(_)$ function that we defined in Equation (5.3).

The costing structure for this computation is rather interesting: since the cost rule for the GPU section of the code does not return a cost for us to use to weight the edge in the CPU cost graph, we instead need to factor this in in the cost derivation of the code in our scheduling regimen. In particular, since the cost of a cost graph is determined via the scheduling policy, instead of having a static edge-weight in the CPU cost graph connecting the sink of the device box to the rest of the graph, this cost will instead be determined dynamically—once the computation within the device box has been costed, this will cost will then be used as the weight for the unweighted edge between the device box and α_3 in Equation (5.7) while calculating the cost of the computation as a whole.

5.5.1 Scheduling

While the scheduling of the cost graph remains largely unchanged from what we saw earlier in Section 3.6.4, we need to update parts of the scheduling policy to take into account device boxes in the graph and the scheduling restrictions around them, and define the way we cost these device boxes in the cost graph.

A Note on Parametrization In the CPU cost graph, we are interested in parameterizing the computation by the traversal order of the graph, or in other words, the schedule in which the computation is run. However, for GPU computations, we have a different dimension of parameterization; while the schedule is fixed, we instead parameterize the costing of the computation by the regions that are loaded into (work group) local memory on the GPU. Since the cost analysis of a device box is dependent upon which regions (and hence which sets of locations) reside in local work-group memory when costing traces, we can load regions into local memory abstractly during scheduling—we don't need to physically load regions into local memory, instead simply treating addresses within a region as local addresses during the cost analysis of the trace depending on whether or not we have 'loaded' that region into local memory (Section 4.7). However, we need a way of determining which regions should be placed in local memory for the execution of each individual kernel that we encounter while traversing the cost graph.

5.5.1.1 Loading Regions Locally

Because of the desire to parameterize memory placement on the GPU, just as we have a schedule order $\trianglelefteq_{\mathsf{w}}$ for abstractly representing schedules for the program represented by the cost graph, we also want to have an abstract way of determining which regions should be placed in local memory for the execution of a given kernel. However, as opposed to the schedule order which defines a preorder over the cost

graph, the function that we need to have for determining region placement needs to be more ad-hoc in nature: it needs to be able to examine both the kernel and the region stack at the point of the call, and to have access to the environment (in order to connect region variables in the program to region names in the region stack) and the size of the local memory on the GPU.

With this in mind, we parameterize the cost model by a function $\mathsf{determineLocal}^n(e, \mathcal{S}, E)$ that, given a program e, a region stack $\mathcal{S}$, the size available in local work group memory n, and the current environment E, returns a set of *ranked regions* where each region name that occurs free in e is associated with a score that will be used to determine if it will be loaded into local memory or not. As an example of such a ranking heuristic function for determining the placement of regions into local memory on the GPU consider the one defined in Figure 5.5 where `unionWith` $(+)$ a b unions the two maps a and b, and combines the value of duplicate keys by addition.

In this ranking function, given an expression $e \in \Lambda^{\mathsf{G}}$ that is to be run on the GPU, we take the *ranked region variables* of e, $\widetilde{\mathsf{frv}}(E, e)$. $\widetilde{\mathsf{frv}}(E, e)$ performs a similar function to the free region variable metafunction $frv(e)$ that was defined in Figure 3.2, however now each region variable is ranked by the number of read and write operations to the backing region. Once this set of ranked region names for the expression has been determined we take the region stack $\mathcal{S}$ in which every region resides in global (GPU) memory and abstractly load these into the local memory on the GPU in a ranked order. This is done using the function $\mathsf{load}^n_{\mathcal{S}}(\{r_1, \ldots, r_n\})$ that was introduced in Section 4.7.0.2, and that we will detail in Section 6.3.3.2.

One could also use more sophisticated methods based on Logical Regions [69], or indeed by using the cost semantics that we are describing here as a scoring function to decide which regions should be loaded into local memory.

$$
\begin{array}{rcl}
a \,\dot{\cup}\, b & = & \texttt{unionWith}\,(+)\;a\;b \\
a \,\dot{\setminus}\, r & = & \texttt{delete}\;r\;a
\end{array}
$$

$$
\begin{array}{lcl}
\widetilde{\mathsf{frv}}(E, x) & = & \{\} \\
\widetilde{\mathsf{frv}}(E, \texttt{True}) & = & \{\} \\
\widetilde{\mathsf{frv}}(E, \texttt{False}) & = & \{\} \\
\widetilde{\mathsf{frv}}(E, i\;\texttt{at}\;\rho_\omega) & = & \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, \texttt{if}\;e_1\;\texttt{then}\;e_2\;\texttt{else}\;e_3) & = & \widetilde{\mathsf{frv}}(E, e_1) \,\dot{\cup}\, \widetilde{\mathsf{frv}}(E, e_2) \,\dot{\cup}\, \widetilde{\mathsf{frv}}(E, e_3) \\
\widetilde{\mathsf{frv}}(E, \lambda x : \tau.^{\varphi}\, e\;\texttt{at}\;\rho_\omega) & = & \widetilde{\mathsf{frv}}(E, e) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, e_1\;e_2) & = & \widetilde{\mathsf{frv}}(E, e_1) \,\dot{\cup}\, \widetilde{\mathsf{frv}}(E, e_2) \\
\widetilde{\mathsf{frv}}(E, \lambda \zeta_\omega \geq \varphi'.^{\varphi}\, u\;\texttt{at}\;\rho_\omega) & = & \left(\widetilde{\mathsf{frv}}(E, u) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\}\right) \dot{\setminus} \{\zeta_\omega\} \\
\widetilde{\mathsf{frv}}(E, e[\rho_\omega]) & = & \widetilde{\mathsf{frv}}(E, e) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, (e_1, e_2)\;\texttt{at}\;\rho_\omega) & = & \widetilde{\mathsf{frv}}(E, e_1) \,\dot{\cup}\, \widetilde{\mathsf{frv}}(E, e_2) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, \texttt{fst}\;e) & = & \widetilde{\mathsf{frv}}(E, e) \\
\widetilde{\mathsf{frv}}(E, \texttt{snd}\;e) & = & \widetilde{\mathsf{frv}}(E, e) \\
\widetilde{\mathsf{frv}}(E, \texttt{fix}\;f : \tau.u) & = & \widetilde{\mathsf{frv}}(E, u) \\
\widetilde{\mathsf{frv}}(E, e_1 || e_2\;\texttt{at}\;\rho_\omega) & = & \widetilde{\mathsf{frv}}(E, e_1) \,\dot{\cup}\, \widetilde{\mathsf{frv}}(E, e_2) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, [e_1, \ldots, e_n]\;\texttt{at}\;\rho) & = & \left(\dot{\bigcup}_{i=1}^{n} \widetilde{\mathsf{frv}}(E, e_i)\right) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\} \\
\widetilde{\mathsf{frv}}(E, \texttt{parmap}\;e\;[e_1, \ldots, e_n]\;\texttt{at}\;\rho_\omega) & = & \widetilde{\mathsf{frv}}(E, e) \,\dot{\cup}\, \left(\dot{\bigcup}_{i=1}^{n} \widetilde{\mathsf{frv}}(E, e_i)\right) \,\dot{\cup}\, \{(E(\rho_\omega), 1)\}
\end{array}
$$

Figure 5.5: Definition of $\widetilde{\mathsf{frv}}(E, e)$. Note that since $e \in \Lambda^{\mathsf{G}}$ we do not handle the `letregion` nor the `on` forms.

To continue on with the example, an example of how we might then use the $\widetilde{\mathsf{frv}}(E, e)$ function in conjunction with $\mathsf{load}^n_{\mathcal{S}}(\bar{r})$ is given below

$$
\begin{array}{c}
\mathsf{determineLocal}^n(e, \mathcal{S}, E) = \mathsf{load}^n_{\mathcal{S}}(\{r_1, \ldots, r_k\}) \\
\text{where } \{r_1, \ldots, r_k\} = \mathtt{trimRVars}(\widetilde{\mathsf{frv}}(E, e), n, \mathcal{S})
\end{array}
$$

where `trimRVars` can be defined as the set of the highest-ranked regions whose total size is less than n:

$$
\begin{array}{l}
\mathtt{trimRVars}(\{(r_1, n_1), \ldots, (r_k, n_k)\}, n, \mathcal{S}) = \\
\quad \{r_1, \ldots, r_t \mid \sum_{i \leq t} |\mathcal{S}(r_i)| \leq n, \forall n_i, 1 \leq i \leq k \,.\, \forall n_j > k \,.\, n_i \geq n_j\}
\end{array}
\tag{5.8}
$$

5.5.2 Determining Costs

With this additional $\mathsf{determineLocal}^n(e,\mathcal{S},E)$ function for parameterizing the loading of regions into local memory when we encounter a device box, we can define how to cost device boxes in the larger graph that is built-up by the cost semantics for $\Lambda_{\mathsf{G}}^{\mathsf{C}}$.

Recall from Section 3.6.4 that a weighted schedule over the graph $\trianglelefteq_{\mathsf{w}}$ produces a series of steps

$$\{V_1,\ldots,V_k\} = \mathsf{steps}(\trianglelefteq_{\mathsf{w}})$$

and from this the cost of the computation from one step to the next could be defined as the maximum weight edge between each step of the computation across the graph, as defined by $\trianglelefteq_{\mathsf{w}}$:

$$\mathsf{step}_w(V_i) = \begin{cases} 0 & i = 1 \\ \max\{\delta \mid \alpha_i \in V_i,\ \alpha_{i-1} \in V_{i-1}, (\alpha_{i-1},\alpha_i,\delta) \in E\} & \text{otherwise} \end{cases}$$

However while the definition of $\mathsf{steps}(\trianglelefteq_{\mathsf{w}})$ still works over these graphs, the way we calculate the cost from one step of the computation to the next no longer works in step_w. In particular, since the cost of the device box is based upon the loading policy that has been chosen, we need to use the unweighted edge from the sink of the device box to dynamically insert the cost of the device box with respect to the given local load policy $\mathsf{determineLocal}^n(e,\mathcal{S},E)$ chosen; once a local memory state $\mathbb{L}$ has been chosen for the kernel the (computational) cost of the kernel is then calculated via $\mathsf{cost}_{\mathsf{G}}(\mathcal{D},\mathbb{L})$ defined in Equation (4.25) and this is then used as the weight for the edge connecting the device box to the next node in the graph.

We thus arrive at the following updated definition of step_w, $\mathsf{step}_w^{\mathsf{H}}$, for cost graphs that include device boxes:

$$\mathsf{step}_w^{\mathsf{H}}(V_i, \mathsf{determineLocal}^n(_,_,_)) = \begin{cases} 0 & \text{if } i = 1 \\ \max \Delta_i & \text{otherwise} \end{cases} \tag{5.9}$$

where

$$\Delta_i = \left\{ \delta \;\middle|\; \begin{array}{l} \tilde{\alpha_i} \in V_i, \\ \tilde{\alpha_{i-1}} \in V_{i-1}, \\ \tilde{\alpha_{i-1}} \in VVars, \\ (\tilde{\alpha_{i-1}}, \tilde{\alpha_i}, \delta) \in E \end{array} \right\} \bigcup \left\{ \mathsf{cost}_{\mathsf{G}}(\mathcal{D}, \mathbb{L}) \;\middle|\; \begin{array}{l} \tilde{\alpha_i} \in V_i, \\ \tilde{\alpha_{i-1}} \in V_{i-1}, \\ \tilde{\alpha_{i-1}} = (\alpha, \mathcal{D}, \kappa, e, E, \ell), \\ \mathbb{L} = \mathsf{determineLocal}^n(e, \mathcal{S}(V_i, \trianglelefteq_{\mathsf{w}}), E) \end{array} \right\}$$

The left-hand side of this union is the same as the set that we calculated the step weight over for the normal CPU cost graphs, whereas the right-hand side of the union performs the dynamic costing of the GPU computation in the case that the edge that we are looking at is from the sink of the GPU node.

With this new function for calculating the cost of each step in the cost graph, we arrive at the following definition for the time cost of a program in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$:

$$\mathsf{time}^{\mathsf{H}}(e, \trianglelefteq_{\mathsf{w}}, \mathsf{determineLocal}^n(_,_,_)) = \sum_{V_i \in \mathsf{steps}(\trianglelefteq_{\mathsf{w}})} \mathsf{step}^{\mathsf{H}}_w(V_i, \mathsf{determineLocal}^n(_,_,_)) \tag{5.10}$$

Updating Weighted Schedule Orders This dynamic costing leads to the following extension of $\trianglelefteq_{\mathsf{w}}$ to take into account device boxes in the cost graph. While in general the schedule order does not need to be changed, the way we build the step contexts does need to change in order to handle the dynamic costing that we perform on outgoing edges from a device box.

Recall from Equation (3.4) that the contextualization $V_i^{\star}$ of a step V_i was defined as

$$\left\{(\alpha, \delta + i) \;\middle|\; \alpha \in V_k, \exists \alpha' \,.\, ((\alpha', \alpha, \delta) \in E) \wedge \alpha' \in V_i \wedge i < k \right\}$$

However, with the dynamic assignment of costs for the outgoing edge from a device box, we need to update this definition of contextualization; we need to assign the

cost of the GPU computation to the edge from the device box just as we did for the weighted step function that we defined previously:

$$\begin{array}{c}\left\{ (\tilde{\alpha}, \delta + i) \;\middle|\; \begin{array}{l} \tilde{\alpha} \in V_k, \\ \exists \tilde{\alpha'} \,.\, \left((\tilde{\alpha'}, \tilde{\alpha}, \delta) \in E \right), \\ \tilde{\alpha'} \in V_i, \\ i < k, \tilde{\alpha}, \\ \tilde{\alpha'} \in VVars \end{array} \right\} \\ \bigcup \\ \left\{ (\tilde{\alpha}, i + \mathsf{cost}_{\mathsf{G}}(\mathcal{D}, \mathbb{L})) \;\middle|\; \begin{array}{l} \tilde{\alpha} \in V_k, \\ \exists \tilde{\alpha'} \,.\, \left((\tilde{\alpha'}, \tilde{\alpha}, \delta) \in E \right), \\ \tilde{\alpha'} \in V_i, \\ \tilde{\alpha'} = (\alpha_1, \mathcal{D}, \kappa, e, E, \ell), \\ \mathbb{L} = \mathsf{determineLocal}^n(e, \mathcal{S}(V_i, \trianglelefteq_{\mathsf{w}}), E) \end{array} \right\}\end{array} \tag{5.11}$$

5.5.2.1 Space Costs

Calculating the space cost of a $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ program however—while straightforward—is slightly more nuanced than what we saw for both Λ_{C} and Λ^{G}: we want to return both the total space used for the CPU portion of the computation, along with the space used for the GPU portion of the computation. Thus the space cost function returns a tuple of costs—one for the CPU and the other for the GPU—instead of a single cost like we have seen thus far.

The CPU space cost is determined just as before, being defined as the high water mark of the space taken by the region stack (which resides on the CPU):

$$\mathsf{space}^{\mathsf{C}}(\trianglelefteq_{\mathsf{w}}) = \max \{ \mathsf{space}(V_i, \trianglelefteq_{\mathsf{w}}) \mid V_i \in \mathsf{steps}(\trianglelefteq_{\mathsf{w}}) \} \tag{5.12}$$

Since regions cannot be resized on the GPU, we can define the space cost of the GPU portion of the computation by taking the maximum of the memory transferred to the GPU at the start of each device box. This can easily be determined by taking the

free region variables in the expression to be run, and summing up the size of the backing regions:

$$\mathsf{space}\mathcal{D}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}}) = \begin{cases} 0 & \text{if } \tilde{\alpha} \in VVars \\ \sum\limits_{\rho \in frv(e)} |\mathcal{S}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}})(E(\rho))| & \tilde{\alpha} = (\alpha, \mathcal{D}, \kappa, e, E, \ell) \end{cases} \tag{5.13}$$

where we define

$$\mathcal{S}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}}) = \mathcal{S}(V, \trianglelefteq_{\mathsf{w}}), \text{ such that } \tilde{\alpha} \in V, \text{ and } V \in \mathsf{steps}(\trianglelefteq_{\mathsf{w}})$$

With this we can then define the space usage of the GPU portion of the code to be the maximum value taken by $\mathsf{space}\mathcal{D}$ for all nodes in the computational graph G:

$$\mathsf{space}_{\mathsf{G}}(\trianglelefteq_{\mathsf{w}}) = \max\{\mathsf{space}\mathcal{D}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}}) \mid \tilde{\alpha} \in G\} \tag{5.14}$$

We then define the space cost function for the entire computation to be the tuple of both $\mathsf{space}^{\mathsf{C}}$ and $\mathsf{space}_{\mathsf{G}}$ over the computation graph under a schedule $\trianglelefteq_{\mathsf{w}}$:

$$\mathsf{space}^{\mathsf{C}}_{\mathsf{G}}(\trianglelefteq_{\mathsf{w}}) = \left(\mathsf{space}^{\mathsf{C}}(\trianglelefteq_{\mathsf{w}}), \mathsf{space}_{\mathsf{G}}(\trianglelefteq_{\mathsf{w}})\right) \tag{5.15}$$

Note that for determining space costs, the local load policy chosen for device boxes does not factor into the equation(s): no matter which way we may load regions into local memory on the GPU (i.e., what we use for $\mathsf{determineLocal}^{-}(_,_,_)$) we will always need to load the same regions, and hence, the same amount of memory onto the GPU.

5.6 Wrapping Things Up

5.6.1 Discussion: Effect Polymorphism

As we mentioned previously we do not have full effect polymorphism in Λ_{C}, and hence in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$. This is because when we transfer functions from the CPU to the GPU, we need to be able to determine the set of regions that also need to be transferred

based upon a *term-level* function, as opposed to one that also operates over types. Thus when we perform effect truncation in Figure 3.5, we are not only performing a type-level transformation, but also a term-level one; for each region abstraction that we introduce at the type-level, we also introduce a term-level region abstraction (and instantiation). Due to this transformation we can gather the free region variables as a purely term-level function, as opposed to one that must operate over both types and terms, and ensures that we will not need to instantiate any effect polymorphic code with regions that we have not determined as needed by the computation that runs on the GPU.

5.6.2 Discussion: Costing of Device Boxes

The dynamic costing of device boxes in the cost graph, along with the calculation of the memory transfer costs to- and from- the GPU, requires that we have at least one vertex of the cost graph that is 'on' the CPU preceding the device box, along with at least *two* that succeed it. Thus, in order to properly cost device boxes, there is a crucial well-formedness property that we require: cost graphs must always start, and finish, on the CPU. Since the only nodes that are in $VVars$ are normal computational nodes that arise from Λ_{C}—and hence represent computation performed on the CPU—we arrive at the following well-formedness theorem for cost graphs in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$.

Theorem 5.6.1 (Well Formedness of Cost Graphs)**.** *Let $G = (\tilde{\alpha_1}, \tilde{\alpha_2}, V, E)$ be a cost graph for $\Lambda^{\mathsf{C}}_{\mathsf{G}}$. Then*

$$\tilde{\alpha_1} \in VVars \wedge \tilde{\alpha_2} \in VVars$$

i.e., both the source and sink nodes of any cost graph will be in $VVars$, and hence 'reside' on the CPU.

Proof. This is easily shown by inspection of the rules for the cost semantics—in particular that for every rule the cost graph created always has source and sink in $VVars$. We then induct on the construction of the cost graphs by showing for any two cost graphs G_1 and G_2 for which the theorem holds, that the theorem also holds for $G_1 \oplus G_2$ and $G_1 \otimes G_2$. □

6

Implementation and Evaluation

Contents

6.1 Introduction **169**
6.2 The Compilation Pipeline **169**
6.2.1 Typechecking 169
6.2.2 Profiling 170
6.2.3 Normalization 170
6.2.4 Closure Conversion and λ-Lifting 171
6.2.5 Retyping 173
6.2.6 Explicitly Passing the Heap 173
6.2.7 Threading 177
6.2.8 Lowering & Code Generation 180
6.3 Profiling **181**
6.3.1 Representing Memory 181
6.3.2 CPU 182
6.3.3 GPU 190
6.4 Primitive Cost Providers **197**
6.4.1 Deriving the ATPR 199
6.5 Evaluation **204**
6.5.1 Comparing to Real-World Runtimes 209

6.1 Introduction

In this chapter we describe a proof of concept implementation of Λ_G^C, and provide a brief evaluation of the cost framework that we have developed. We first give an overview of the compiler for Λ_G^C and the compilation techniques that we utilize (Section 6.2) after which we provide an in-depth discussion Section 6.3 of the implementation and details of the profiling theory and system that were presented in Chapters 3 to 5. We then discuss the determination of our primitive cost providers (Section 6.4), and conclude by evaluating the efficacy of the cost framework that we have developed against the known real-world costs of these programs in Section 6.5. Further, since the theory that we have built determines the *abstract* cost of a program, we examine the efficacy of our framework for determining real-world behavior by measuring the *ratio* of the abstract costs inferred for programs running on the CPU and the GPU to the real-world runtimes of these same programs in Sections 6.5 and 6.5.1.

6.2 The Compilation Pipeline

The compilation for Λ_G^C, is largely straightforward and does not present any particularly difficult, or previously unencountered implementation challenges. The compilation consists of the following passes:

6.2.1 Typechecking

This pass verifies the typing and effect properties for Λ_G^C based on the type system in Figures 5.2 and 5.3. The implementation is straightforward and follows readily from the typing rules that have been given for Λ_G^C. Beyond checking the types for the language, this pass also annotates each expression in the AST with the expression's

type. Thus, after this pass each expression e will be annotated with a type τ. Notationally, we refer to the expression e along with its type as e^τ, and in the case where we are just referring to the expression (without regard to its type) as e. This typing information will then be used later on by other compilation phases.

6.2.2 Profiling

In this pass we perform semantic profiling based upon the theory described in Chapters 3 to 5. We go into detail on the implementation of this in Section 6.3.

6.2.3 Normalization

In this pass we perform A-normalization [73][1] which defines each intermediate result of a computation as a variable. For example if we had the following expression before this pass:

$$e_1^{\tau_1} + e_2^{\tau_2} - e_3^{\tau_3}$$

Then after performing normalization of this term, we would get the following expression

```
let x₁ : τ₁ = e₁ in
let x₂ : τ₂ = e₂ in
let x₃ : τ₃ = e₃ in
x₁ + x₂ - x₃
```

where the x_i are fresh, and where we utilize the typing annotations that we have placed on the AST from the previous pass to determine the τ_i when binding to the expression e_i.

[1] Which is similar to, and is sometimes called K-normalization [74].

6.2.3.1 Constant Folding

In this pass we compute basic arithmetic expressions that are already known at compile time. For example, if we had the following expression

```
let x : int = 1 in
let y : int = 2 in
x + y
```

then after constant folding we would get `3` for this expression. More can be read about constant folding in Muchnick et al. [75].

Furthermore, when we encounter a parallel tuple form $e_1 || e_2$ `at` ρ we convert this into a parallel let, or `plet` form which takes two binders—one for each field of the tuple—and binds these to the normalization of e_1, and e_2 respectively. Thus we transform a parallel tuple $e_1^{\tau_1} || e_2^{\tau_2}$ `at` ρ into the following expression

```
plet x : τ1 = normalize e1
parwith y : τ2 = normalize e2 in
[makeTuple(x : τ1, y : τ2)] @ ρ
```

and where we differentiate primitive function calls to the runtime, from other function calls by wrapping them in square brackets.

6.2.4 Closure Conversion and λ-Lifting

This pass performs both closure conversion and λ-lifting. While the implementation of this can sometimes be non-trivial, the literature and algorithms for both of these are quite extensive, so we refer readers interested on reading more about this to e.g. Minamide, Morrisett, and Harper [76] and Johnsson [77].

However, while the implementation is relatively straightforward, it is important to highlight that at this point in the compilation process we have already verified the safety of the region abstractions and instantiations in the typechecking pass. Due to this, we can erase the region bounds for abstractions and treat them as normal

abstraction and instantiation forms binding region variables. Thus after this pass we no longer have region abstraction or instantiation forms, and instead replace these with normal function closures (over the region abstraction variable) and applications.

The above transformation correlates to the eventual runtime representation of region abstractions and instantiations in our backend. However, copying regions at each call-site is both highly inefficient, and violates our cost semantics for the language. Thus we mark the closures that arise from region abstractions so that later on we can ensure that these parameters are passed by reference in the resulting code.[2]

However, we are not done with transforming region abstractions just yet. Recall that in the definition of $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ that we have the following grammar productions:

$$\begin{array}{lcl} e & ::= & \ldots \mid \lambda\zeta \geq \varphi'.^{\varphi}u \ \texttt{at}\ \rho \mid \ldots \\ u & ::= & \lambda x : \tau.^{\varphi}e \ \texttt{at}\ \rho \mid \lambda\zeta \geq \varphi'.^{\varphi}u \ \texttt{at}\ \rho \end{array}$$

Importantly, this means that within each region abstraction, the only things that can occur immediately are either another region abstraction, or a normal λ-abstraction. Furthermore, we must eventually bottom out at a λ-abstraction. Using this, and since we convert region abstractions into closures, we can fuse the closures that we would generate for any region abstractions with the underlying term-abstraction that the body of the abstraction resolves to. Thus, for example, given the following program (where ρ' is a valid region variable)

```
1   λ ρ1 ... .
2    λ ρ2 ... .
3      λ x : ... .
4        e
5       at ρ' at ρ' at ρ'
```

[2]Furthermore, knowing which arguments need to be passed by reference will also be important when performing threading, since we need to explicitly pass by reference at the call-site.

we fuse region abstractions on line 1 and 2 into the underlying lambda term on line 3, removing the bounding effects, and typing the region variables as *references* to the region type `region` which we introduce at this point in the compilation process. We thus arrive at the following for the above example[3]

```
λ (ρ1 : ref(Region)) (ρ2 : ref(Region)) (x : ..) .
e at ρ′
```

and create the closure for this term.

6.2.5 Retyping

After we have converted the closures for our program, the resulting AST may be ill-typed—function types now need to be converted to closure types in the case that they close over free variables, and others may remain primitive function types in the case that the function does not close over any free variables; since we can use a function pointer (direct-call) instead of creating a closure in the generated code. Performing retyping is a straightforward pass in the compiler, and consists of keeping track of the function symbols that we have converted to closures and those that can be represented as primitive functions in our backend. We then traverse the AST and possibly change the type of that variable based upon the the typing environment that we have kept from closure conversion.

6.2.6 Explicitly Passing the Heap

Recall from Section 3.3 that we have a special single region $\underline{\rho}$ that is restricted in size, and that is accessible from all parts of the program—the "heap". One option for representing this in the generated code is through a region that is globally scoped. However, doing so presents two problems: firstly, this would mean having a globally scoped variable that may have parallel reads and writes in the generated code; and

[3] Note that at this point, we also start allowing multi-argument functions in the AST.

secondly having such a global variable presents problems when running a heterogeneous program—we need this shared heap variable to be both a device *and* host variable, however, having such a globally scoped variable is not supported in most versions of CUDA. To get around this problem, for any function that accesses $\underline{\rho}$, we add an extra "heap" parameter, and then transitively close the call graph to bind the heap.

As an example, take the following source program in $\Lambda_{\mathsf{G}}^{\mathsf{C}}$:

```
(λ x : int {heap} (x,2) at heap) at heap
```

this then compiles to the following snippet of code, in a `C`-like AST just before this pass runs:

```
def f_temp : ((Int, Int), _heap_) (x : Int) {
  [makeTuple(x:Int, 2:Int)] @ _heap_
} FREE ()[_heap_]

def main : ... () {
  let fresh_id_2 : ... = {
    Clos{f_temp FREE(_heap_)} @ _heap_
  } in {
    fresh_id_2
  }
}
```

However, a problem arises in this transformation: the heap variable `_heap_` is not defined within the body of `f_temp`; indeed, it shows up in the free region variables that we annotate each function with during the compilation process, we therefore add an extra parameter `_heap_` to `f_temp` to make this code valid.[4] We then record this additional parameter, and traverse the AST, changing each call site to `f_temp` to also pass through the `_heap_` parameter. Taking this into account, we will arrive at the following code after this pass runs:

```
def f_temp : ((Int, Int), _heap_) (x : Int, _heap_ : ref(Region)) {
  [makeTuple(x:Int, 2:Int)] @ _heap_
} FREE ()[]
```

[4]Note that we we are no longer restricted by the source AST at this point in the compilation process—in particular we are allowed to have multi-argument functions.

```
def main () {
  let fresh_id_2 : ... = {
    Clos{f_temp FREE()} @ _heap_
  } in {
    fresh_id_2
  }
}
```

It is important to note that we special-case the "main" function, since we will create the heap region at the very beginning of this function. Thus we do not need to add a parameter for the heap region.

While the above method based upon free variables works for many cases, simply adding a parameter to bind the heap region in each function in which `_heap_` occurs free is not good enough. Indeed, consider the following source program:

```
let f : int → (((int, int), heap), {heap}), heap =
  λ x : int {heap} (x,2) at heap at heap
in
let h : int → (((int, int), heap), {heap}), heap =
  λ x : int {heap} f x at heap
in
h 1
```

and its corresponding compilation before this pass:

```
 def h_temp : ((Int, Int), _heap_) (x : Int) {
  f_temp(x)
} FREE ()[]

def f_temp : ((Int, Int), _heap_) (x : Int) {
  [makeTuple(x:Int, 2:Int)] @ _heap_
} FREE ()[_heap_]

def main : ((Int, Int), _heap_) () {
  h_temp(1)
}
```

In this case we have one function—`h_temp`—that, while `_heap_` does not occur free within its body, after the transformation to both `f_temp` and its call-sites, it does indeed occur free. Indeed, we would get the following invalid code using the above policy for determining when to add heap-region parameters:

```
def  h_temp : ((Int, Int), _heap_) (x : Int) {
  f_temp(x, _heap_)
} FREE ()[]

def f_temp : ((Int, Int), _heap_) (x : Int, _heap_ : ref(Region)) {
  [makeTuple(x:Int, 2:Int)] @ _heap_
} FREE () [_heap_]

def main ((Int, Int), _heap_) () {
  h_temp(1)
}
```

We could solve this problem by iteratively re-traversing and rewriting call-sites until we reach a fixed point (with respect to the transformed AST), but there is a much easier and more efficient solution to this problem: we amend the criteria to determine when to add a heap argument to the function f to be either that `_heap_` occurs free within the body of f, or, there exists a call-site $h(x_1, \ldots, x_n)$ within the body of f such that the call-site of h is rewritten. This can be seen as pre-empting the fact that in the rewriting of the call to h that we are then creating a free-variable with the heap variable, we then thread this through to all calling functions (and their calling functions etc. until we hit `main` in which `_heap_` is in scope). Using this method, we arrive at the following valid code:

```
def  h_temp : ((Int, Int), _heap_) (x : Int, _heap_ : ref(Region)) {
  f_temp(x, _heap_)
} FREE ()[]

def f_temp : ((Int, Int), _heap_) (x : Int, _heap_ : ref(Region)) {
  [makeTuple(x:Int, 2:Int)] @ _heap_
} FREE ()[]

def main ((Int, Int), _heap_) () {
  h_temp(1, _heap_)
}
```

Indeed, by eagerly rewriting functions that we encounter as applications as we traverse the AST, we arrive at a one-pass, linear-time solution to this problem.

6.2.7 Threading

This pass maps the parallel operators in $\Lambda_{\mathsf{G}}^{\mathsf{C}}$—$e_1 \| e_2$ and `parfor`—to their underlying parallel primitives. However, we need to also take into account the fact that these will map differently based upon whether or not we are within an `on` from in the language. We therefore separate our handling into two cases.

Furthermore, in this pass we move to an imperative AST, and we therefore *stratify* the AST into *statements* (that have no return value), and *expressions* that return a value. Furthermore, during this pass we will need to not only just "return" expression values, but may need to instead return it through setting some variable. Thus when we encounter an expression in a statement context we need a way to determine just how that expression's value should be returned. We do so by passing a function from the calling context that when applied to an expression produces the correct return statement. Thus the pass—$[\![e, ret]\!]_{\mathsf{S}}$—for converting from the previous AST to this new stratified one takes two arguments; the first being the expression that we are processing, and the second the function that we apply to an expression that we encounter in a statement context.

Outside an `on` form — on the CPU For the parallel tuple operator, this is the spawning of two tasks, coupled with a syncing (or joining) of the two, followed by creating a tuple. We thus perform the following transformation. Given the following program structure

```
plet x : τ1 = e1
parwith y : τ2 = e2 in
[makeTuple(x:τ1, y:τ2)] @ ρ
```

we then pull e_1 and e_2 into their own top-level functions by wrapping them in a function whose parameters close over both the free term and region variables in each of the terms, and name the function based upon the name (x or y) that is passed in. Thus given an expression e along with the name for the function f, and the return

type τ we generate a top-level function based on the following (recall that $[\![e, ret]\!]_S$ is recurrence to the top-level of the pass):

```
def f(fv₁ : τ₁, ..., frv₁ :  ref(Region), ..., ret_var : ptr(τ)) {
      [[e, λ x : τ → *ret_var = x]]_S
}
```

Listing 6.1: Hoisting of a thread expression to the top-level.

where $\{\texttt{fv}_i\} = fv([\![e, _]\!]_S)$ and $\{\texttt{frv}_i\} = frv([\![e, _]\!]_S$. Note that we are passing the result of the thread computation in as the last parameter to these functions—and is due to the underlying thread(/task) functions that we use in the parallel runtime.

After having hoisted both e_1 and e_2 into their own top-level functions—f_x and f_y respectively—we transform the original program into the following

```
x : τ₁;
y : τ₂;
x' : Thread = thread(f_x, fv^x_1, ..., ref(frv^x_1), ..., addr(x));
y' : Thread = thread(f_y, fv^y_1, ..., ref(frv^y_1), ..., addr(y));
join(x', y');
[makeTuple(x:τ₁, y:τ₂)] @ ρ
```

where `x'` and `y'` are fresh.

We perform a similar transformation when encountering a **`parmap`** term, however instead of creating only two threads—one for each component of the pair—we create a thread for each element in the array.[5] Thus given the following source expression

```
parmap e [e_i]_{i=1}^n at ρ
```

Listing 6.2: Parallel map expression before transformation.

we first transform e in the same way that we did for the parallel-tuple form, and then us this newly-created function—`f_task`—within the loop body that we generate. This loop traverses the array, and sparks off a task for each element after which we

[5]Note that we do not worry about granularity controls around the amount of parallelism, and leave this instead to the number of underlying threads that we allow in our task runtime to control the "true" amount of parallelism that is present.

re-traverse the array joining all the tasks that were sparked. We thus arrive at the following:

```
threads : [Thread];
ret : [Int] = alloc[Int, ρ](n);
for(Int index = 0; index < n; ++index) {
  ret_local : ref(Int) = ret[index];
  in_local  : Int = arr[index];
  threads[index] = thread(f_task, in_local, ref(ρ), addr(ret_local));
};
for(int index = 0; index < n; ++index) {
  join(threads[index];
}
```

Within an `on` form Recall from Section 6.2.3, that when we encounter a parallel tuple form $e_1 \| e_2$ `at` ρ during normalization, we simply elide the transformation to a `plet`, and instead transform this into a normal tuple form. Thus we only need to handle the **`parmap`** case when on the GPU—particularly, generating the kernel and kernel call. Thus when we encounter a **`parmap`** form as in Listing 6.2 while within an `on` form, we perform two operations. The first, is hoisting the function e that is being mapped into a valid kernel function. This is easily done by transforming the above **`parmap`** form into the following:

```
defkern f_tmp(arr_in : ref([Int]), ret : ref([Int]), arr_len : Int,
              fv1 : τ1, ..., frv1 :  ref(Region), ...) {
  tid : Int = threadIdx.x;
  stride : Int = blockDim.x;
  for (int index = tid; index < arr_len; index += stride){
    f_th(arr_in[index], fv1, ..., frv1, ..., addr(ret[index]));
  }
}
```

and where $\texttt{f}_{th}$ is generated from e as in Listing 6.1, and where $\{\texttt{fv}_i\} = fv([\![e, _]\!]_S)$ and $\{\texttt{frv}_i\} = frv([\![e, _]\!]_S$ as before.

Once we have transformed the expression into a kernel in this manner, we then replace the **`parmap`** with the kernel call to this kernel, passing along the appropriate free term, and region, variables.

```
ret : [Int] = alloc[Int, ρ](n);
f_tmp<<<numB,numTh>>>([e]_{i=1}^{n}, ref(ret), n, fv_1, ..., ref(frv_1), ...);
synchronize;
```

Where we recall from Section 2.4.1 that `numB` specifies the number of thread blocks, and `numTh` specifies the number of threads per thread thread block. Furthermore, recall that at the start of an `on` form that we transfer over all regions needed for the computation within the `on` form, and merge these regions back into their parent (host) regions at the end of the scope of the `on` form. We therefore do not need to worry about transferring memory to and from the kernel at the start and end of the kernel call.

6.2.8 Lowering & Code Generation

After we have converted the code into an imperative AST, we then perform a straightforward translation of this imperative AST into a restricted subset of the `C++11` AST along with kernels. During this lowering, we pre-declare the fields and methods of structs before generating the actual method definitions—this is to allow (mutually) recursive functions and struct definitions in the generated code.[6]

Beyond this, we also remove **`letregion`** forms at this point, and convert them to allocations *within a block* in the generated code—thus we do not need to worry about manually inserting a destructor at the end of the block, and instead the C++ memory manager will automatically use the region-allocator's destructor when the region variable leaves scope.

```
                            {
letregion ρ in e              Region ρ = create_region();
                    ⟹         e';
                            }
```

[6]Since we use structs with methods to represent closures in the generated code, getting into the situation in which we generate mutually recursive struct definitions is in-fact quite easy.

While in this section we've provided an overview of most of the passes in the compiler, we have not described the implementation of the profiling system. In the next two sections, we will dive into the implementation of the semantic profiling system (based upon the cost semantics described in the previous three chapters), after which we'll move on to determination of PCPs.

6.3 Profiling

Since the compiler serves as a test-bed for the profiling theory that we have developed in Chapters 3 to 5, these passes are rightly the most complicated and thought-intensive of the whole compilation pipeline. We first take a look at how we (abstractly) represent memory in this system.

6.3.1 Representing Memory

Since $\Lambda_{\mathsf{G}}^{\mathsf{C}}$ is a region-based language, we represent the abstract memory in the profiling system as a stack of regions, with each region holding the current allocation pointer, size, and underlying memory (as defined in Definition 3.4.1). Each piece of memory is a mapping of offsets (`int`s) to values v (as defined in Figure 3.1) which we represent as `Val.t` in the implementation. Thus we have the following types for the memory system:

```
type memory = (int, Val.t) Hashtbl.t

type region = {
  allocation_ptr : int;
  size           : int;
  memory         : memory;
}
```

Using this, we represent the region stack as a class in OCaml with the following methods and types:

```
type region_stack_internal = (Reg.name, region) Hashtbl.t
class reg_stack = object
  val mutable stk : region_stack_internal  = ...
  val mutable name_stack : Reg.name Stack.t = ...
  val alloc_amt = ...
  (* Push a region/name onto the region stack *)
  method push (rname: Reg.name) (reg: region): unit = ...
  (* Update the underlying region. Used for CPU/GPU merging *)
  method update_underlying_region (rname: Reg.name) (reg: region): unit = ...
  (* Pop region off region stack *)
  method pop () : unit = ...
  (* Membership query of region stack *)
  method has (rname: Reg.name): bool = ...
  (* Get underlying region for region name *)
  method get_rg (rname: Reg.name): region = ...
  (* Get value stored at a location *)
  method get ((rname, index): Val.loc): Val.t = ...
  (* Get current allocation ptr for backing region *)
  method alloc_ptr (rname: Reg.name): int = ...
  (* Allocate value v in region rname in the region stack *)
  method alloc (v: Val.t) (rname: Reg.name): int = ...
end
```

Note how internal representation of the region stack is using a hashtable from region names to regions instead of an actual stack—this is to allow us to search and lookup regions by their name when allocating or looking up values. However, in order to emulate the FIFO behavior that we need of the region stack, we also keep a stack of region names internally—thus when pushing a region onto the stack we both push the name on to the name stack, and then insert that name along with its backing region into the hashtable.

With this definition of the region stack, and the methods available to us, we now move on to the CPU profiling implementation.

6.3.2 CPU

The CPU implementation of the profiling framework follows in a relatively straightforward manner from the cost rules detailed in Chapter 3. Thus this pass consists primarily of the function to build the cost graph based upon the cost semantics:

```
let rec build_cost_graph (env: env) (e: Ast.t): cost_graph * Val.t
```

Where a cost graph type is defined as a product type consisting of a single source and sink vertex along with a set of vertices and edges that make up the graph:

```
type cost_graph = {
  src      : vertex;
  sink     : vertex;
  vertices : VertexSet.t;
  edges    : EdgeSet.t;
}
```

And, just as in the rules for the cost semantics that we presented in Section 3.6.3 in which the environment that mapped variables to values, and *region variables* to their underlying *region names* the environment here consists of two sets of mappings:

```
type env = {
  regions   : Reg.name RMap.t;
  variables : Val.t SMap.t;
}
```

To see how the implementation of the rules relates to the cost semantics, take as an example the following two rules from Figure 3.19—the first for the `parmap` operation, and the second for region allocation/creation:

$$\frac{\begin{array}{c} E;\mathcal{S};f\ e_i \Downarrow v_i;\mathcal{S}';c_i \qquad r = E(\rho) \qquad R = \mathcal{S}'(r) \\ R;[v_1,\ldots,v_n] \downarrow^{\delta_a} o@R' \qquad \mathcal{S}'' = \mathcal{S}'[R'/R] \end{array}}{E;\mathcal{S};\ \texttt{parmap}\ f\ [e_1,\ldots,e_n]\ \texttt{at}\ \rho \Downarrow \ell;\mathcal{S}'';\left(\bigotimes_i c_i\right) \oplus [\alpha_1] \oplus_{\delta_a} [\alpha_2]}\ (\alpha_1,\alpha_2\ \text{fresh}) \quad (6.1)$$

$$\frac{\begin{array}{c} R = \mathit{newRegion}(\mathsf{allocSize}) \qquad r\ \text{fresh} \qquad r \notin \mathsf{dom}(\mathcal{S}) \\ \delta_R = \Delta^a(|R|) \qquad E[\rho \mapsto r];\mathcal{S}[r \mapsto R];e \Downarrow v;\mathcal{S}';c \qquad \mathcal{S}'' = \mathcal{S}' \setminus r \end{array}}{E;\mathcal{S};\ \texttt{letregion}\ \rho\ \ \texttt{in}\ e \Downarrow v;\mathcal{S}'';[\alpha_1] \oplus_{\delta_R} [\alpha_2] \oplus c \oplus [\alpha_3]}\ (\alpha_i\ \text{fresh}) \quad (6.2)$$

The rule for region creation in `build_cost_graph` is implemented by the following snippet of code. Compare this to the rule for `letregion`. in Equation (6.2)

```
| A.LetReg(reg, e) → (* letregion r in e *)
(* Create a fresh backing region name *)
  let rname = Reg.fresh_rname () in
  (* Allocate new region and push it on to region stack *)
  let new_region = Memory.new_region () in
  (* region_stack is global *)
  region_stack#push rname new_region;
  (* Calculate the cost of the region allocation using PCP *)
  let cost = PrimCosts.reg_cost new_region in
  (* Extend region environment with mapping of reg → rname
     and build the cost graph. *)
  let g, v = build_cost_graph (extend_renv reg rname env) e in
  (* End of scope, so pop off region stack *)
  region_stack#pop;
  (fresh_single_edge cost) +.+ (g +.+ fresh_empty_graph), v
```

As another—more complicated—example, the implementation of the `parmap` rule in `build_cost_graph` is implemented by the following snippet of code, and it too, can largely be read off from the **parmap** rule in Equation (6.1).

```
| A.ParMap(e, arr, reg) → (* parmap e arr at reg *)
  (* build the cost graphs for e and the array *)
  let gclos, closv = build_cost_graph env e in
  let g, v = build_cost_graph env arr in
  (* Get the underlying array *)
  begin match lookup_val v with
    | cost1, Val.VArr vs →
    (* Make sure the thing being mapped is a function (closure) *)
      begin match lookup_val closv with
        | cost2, Val.VClos(x, ty, e, env') →
          let env' = {env with variables = env'} in
          let gs, vs = List.split begin
          (* Build a cost graph for each thread in the parray *)
              List.map begin fun v →
                build_cost_graph (extend_venv x v env') e
              end vs
            end in
          let vl = Val.VArr vs in
          (* Annotate the vertex with its values size *)
          let alloc_edge_nm = annotate_vertex_w_size vl in
          (* Allocate the value *)
          let acost, loc = alloc env vl reg in
          (* Cost graph before running *)
          let f_edge = fresh_single_edge cost1 in
          let pre_graph = gclos +.+ (scompose_w f_edge fresh_empty_graph cost2) in
          (* Cost graph for the running of the parmap *)
          let parmap_graph = List.fold_left begin fun acc grph →
            acc *.* grph
          end fresh_empty_graph gs in
```

```
            (* Cost graph for after the parmap *)
            let cont_edge = fresh_single_edge alloc_edge_nm acost in
            pre_graph +.+ (parmap_graph +.+ cont_edge), Val.VLoc loc
          | _, v → failwith ...
        end
      | _,v → failwith ...
    end
```

Where `scompose_w g1 g2 weight` takes two cost graphs `g1` and `g2` and returns their weighted serial composition as defined in Figure 3.15. Furthermore, it is crucial to note that we annotate allocation vertices in the graph with the size of the value being allocated as well—this will prove crucial later on during scheduling when we want to determine the amount of space allocated at a given step in the schedule across the graph.

6.3.2.1 Representing Schedules

Having discussed the building of the cost graphs, we move to the implementation of schedules (schedule orders) over the cost graphs.

Recall from Section 3.6.4 that a step context is a set of vertex and integer-multiset pairs—where each integer represents the step in the future that the thread assigned to the parent of that vertex will finish its computation. Thus when the integer multiset assigned to the vertex is empty, or when the current step is greater than all integers in the multiset, that vertex becomes schedulable. Furthermore recall that the cost of a step with respect to a schedule is calculated by taking the maximum over the weights of the edges traversed in that step:

$$\mathsf{step}_T(\overline{V}) = \sum_{V_i \in \overline{V}} \mathsf{step}_w(V_i)$$

We represent both the step contexts and the pairing of finished vertices with their weights by the following types

```
type cost = int
type vc = vertex * cost
type step_context = (int list) VMap.t
```

Since the step context represents at which step the threads needed to schedule the vertex complete, the scheduling of a cost graph can be seen as similar to topological sorting of weighted DAGs. However there are some crucial differences. Particularly, topological sorting is not concerned with sets of simultaneously scheduled nodes—however with our scheduling policy this is given by the schedule order which tells us how to determine which nodes should be chosen from a (possibly) $p + k$ number of available nodes in a p-thread system i.e. this tells us among other things how to break ties between nodes when traversing the cost graph.

With this knowledge we can detail the function that traverses the graph and builds the sets of simultaneously scheduled nodes:

```
1  let schedule_graph (g: graph) (num_threads: int)
2    (schedule: vc list → int → step_context → vc list * vc list):
3    (vertex list * cost) list =
4    let rec step (step_ctxt: step_context) (step_acc: (vertex list * cost) list)
5        (scheduled_nodes: vc list) (remaining_nodes: vc list) (curr_step: int) =
6      (* Remove threads that finished at this step *)
7      let pstep_ctxt = prune_step_context step_ctxt curr_step in
8      (* If there are no more scheduled nodes, or any nodes left to schedule *)
9      if scheduled_nodes = [] && remaining_nodes = [] &&
10        (* And we don't have any outstanding threads *)
11        VMap.is_empty pstep_ctxt then
12     (* Then we are done scheduling the graph, so return the steps *)
13     then step_acc
14     else
15      (* Advance the frontier by looking at all vertices that could be next *)
16      let next_step = List.map fst scheduled_nodes
17                   |> fun x → get_schedulable_nodes g x pstep_ctxt
18      in
19      (* Put the new nodes in with the previously unscheduled_nodes *)
20      let schedulable_nodes = remaining_nodes @ next_step in
21      (* schedule the nodes to run. May not be able to schedule all of them *)
22      let new_scheduled_nodes, unscheduled_nodes =
23        schedule schedulable_nodes num_threads pstep_ctxt
24      in
25      (* Calculate the step cost for the step that we performed here. *)
26      let step_cost = List.fold_left max 0 (List.map snd scheduled_nodes) in
27      (* Get the minimum cost of advancing this step *)
28      let min_step_cost = List.fold_left min max_int (map snd scheduled_nodes) in
29      (* Put all the simultaneously scheduled vertices together. *)
30      let scheduled_vertices = (map fst scheduled_nodes, step_cost)::step_acc in
31      (* Recurse to keep on scheduling *)
32      let step_ctxt = add_nodes new_scheduled_nodes step_ctxt curr_step in
```

```
33      let step_cost = curr_step + min_step_cost in
34      step step_ctxt scheduled_vertices new_scheduled_nodes unscheduled_nodes
      step_cost
35    in
36    step (VMap.singleton g.src [0]) [] [(g.src, 0)] [] 1
```

Even though we can think of the scheduling of the graph as being similar to a topological order, we actually write in a similar fashion to a breadth-first search where we keep track of a frontier as we progress across the function. And, this can be seen as having quite a number of similarities to search strategies for graphs (such as $A^\star$-search) where we use a "scoring" function—the schedule in this case—to determine which nodes to progress to next.

Going through this function in more detail, on line 9, we prune the step context via the `prune_step_context` function—since the context contains the set of threads that need to finish before a given vertex can run, we remove these threads from the thread context if the current step is greater than or equal the step at which the thread is supposed to finish:

```
let prune_step_context (ctxt: step_context) (step: int): step_context =
  VMap.fold begin fun vtx counts acc →
  match List.filter (fun i → step < i) counts with
    | [] → acc (* No more outstanding threads *)
    | l → VMap.add vtx l acc (* Still outstanding *)
  end ctxt VMap.empty
```

After we have pruned our step context, we then determine whether or not we are finished scheduling the graph on lines 11 – 16. If there still remain nodes in the graph left to schedule, we first try to advance the frontier of the graph based upon the nodes that were scheduled in the previous step (given by `scheduled_nodes`) on line 19, and where `get_schedulable_nodes` is defined as follows

```
let get_schedulable_nodes (g: graph) (vs: vertex list)
  (step_ctxt: step_context): (vertex * cost) list =
  (* The vertex is schedulable if there are no parents pointing to it *)
  let is_schedulable vtx = VMap.mem vtx step_ctxt |> not in
  ESet.fold begin fun (v1, v2, weight) acc →
  (* v1 has already been scheduled, but can v2 now run? *)
    if List.mem v1 vs && (is_schedulable v2) then (v2, weight)::acc else acc
```

```
  end g.edges []
```

Once we have advanced the frontier of the graph based upon what nodes have now possibly become schedulable after the previous step runs, we add this to the set of *schedulable* nodes—which are these plus the nodes that were able to be scheduled on the previous step, yet were not scheduled (line 21).

Once we have this final set of schedulable nodes, we pass this to the scheduling function that will determine which nodes will be scheduled for the next step on line 23—this is the function that is the schedule order in Section 3.6.4. As an example of what one of these schedule orders would look like, recall the greedy schedule that we defined in Equation (3.11). However, as opposed to there where we needed to include the fact that no dependencies remained in order for a node to be schedulable, when implementing a schedule we do not need to include this logic; the greedy p-traversal schedule order can be defined thusly:

```
let greedy_p (vs: vc list) (num_threads: int) (v_star: step_context): vc list * vc
    list =
  let num_threads = get_num_avail_threads_pruned num_threads v_star in
  list_split_at num_threads vs
```

After this we calculate the minimum step cost, and add the now scheduled threads into the step context (line 31):

```
let add_nodes (newly_scheduled: vc list) (step_ctxt: step_context) (curr_step: int)
    : step_context =
  List.fold_left begin fun acc (v, c) →
    match VMap.find_opt v step_ctxt with
    | None → VMap.add v [c + curr_step] acc
    | Some cs → VMap.add v ((c + curr_step)::cs) acc
  end step_ctxt newly_scheduled
```

6.3.2.2 Determining costs

Once we have created the steps across the cost graph for a given schedule, determining the cost is straightforward—consisting of summing the cost for each step:

```
let calc_total_time (steps: (vertex list * cost) list): cost =
    List.fold_left (fun total_cost (_, cost) → total_cost + cost) 0 steps
```

When profiling a program there are two space metrics that we report: the first *used space* is the amount of space used by the *values* that have been allocated up to that point in the program; the second *allocated space* is the space that is allocated for regions (regardless of whether or not that space contains values) up to that point in the program. Thus a freshly allocated (and unused) region would have used space 32[7] and allocated space $32 + \mathsf{allocSize}$. Put another way, recalling the definition of space at a step V with respect to a schedule $\trianglelefteq_{\mathsf{w}}$ from Equation (3.12):

$$\mathsf{space}(V, \trianglelefteq_{\mathsf{w}}) = |\{\ell \in R \mid R \in \mathcal{S}(V, \trianglelefteq_{\mathsf{w}})\}| = \sum_{R \in \mathcal{S}(V, \trianglelefteq_{\mathsf{w}})} |\, R.v \,| \tag{6.3}$$

which takes the used space across a schedule. We can likewise easily define the allocated space at a step to be given by

$$\sum_{R \in \mathcal{S}(V, \trianglelefteq_{\mathsf{w}})} |\, R \,| \tag{6.4}$$

Since we annotate vertices that involve allocation in the cost graph with the size of the value being allocated, we can easily determine the amount of space taken as we traverse the cost graph, by accumulating these allocation sizes by the region in which they are allocated, and then removing them when the region is deallocated—which recall from our cost semantics is also represented as a node in the cost graph. Determining the space used at a given step across the schedule is given by the following piece of code:

```
let calc_space_at_step ((vertices, _): vertex list * cost)
  (curr_regions: cost SMap.t): cost * cost SMap.t =
  let new_reg_nodes = add_regions vertices curr_regions in
  let deallocated_nodes = remove_regions vertices new_reg_nodes in
  let new_step = add_allocs vertices deallocated_nodes in
  let cost = Map.sum_values new_step in
  cost, new_step
```

[7] The size of the bookkeeping fields for the region.

with which we can easily define the space of the schedule over the cost graph to be the maximum (computed recursively) of the set of steps in the traversal:

```
let calc_total_space (vc: (vertex list * cost) list): cost =
 let rec calc (vc: (vertex list * cost) list) (step: cost SMap.t) (curr_cost: cost)
    : cost =
    match vc with
    | [] → curr_cost
    | v::vs →
      let cost, new_step = calc_space_at_step v step in
      max cost curr_cost |> calc vs new_step
  in calc vc (SMap.singleton (Reg.print_nm heap_rname) PrimCosts.base_alloc_size) 0
```

Computing the allocated space for a schedule is likewise quite easy and far more straightforward than this.

6.3.3 GPU

In this section we detail the implementation of the GPU cost semantics. In order to do this, we start with the types that we create. We first define the set of instructions that make up the thread graphs:

```
type instr =  Less | Mul | Add | Minus | Eq
           | Kern of kern_id
           | Put of Val.loc
           | Get of Val.loc
           | MetaPut of Val.meta_location
           | MetaGet of Val.meta_location
           | Annotated of int * instr
           | NoMemAccess
```

and where we define a `meta_location` to be a unique id coupled with a region name:

```
type meta_location = {
  uid: string;
  region_name: Reg.name;
}
```

After this, we define the graph structures and partitions based upon the definition of GPU thread graphs in Section 4.4:

```
1  (* Type of a kernel ID *)
2  type kern_id = string
3  (* Represents an entry in the kernel pool (see Section 4.5.3) *)
```

```
 4 type kern_pool_entry = {
 5   func: Ast.t;
 6   arr : Ast.t;
 7   mloc: Val.meta_location;
 8 }
 9 type thread_graph = instr list
10 type kernel_pool  = kern_pool_entry SMap.t
11 type warp_graph   = { threads: thread_graph list; }
12 type warp_pool    = {
13   warps: (warp_graph ref) list;
14   (* Tracks the current focusing strategy for warp
15      schedules. Used during the scheduling portion. *)
16   current_foc : int;
17   (* The local memory that we will be using, along with the memory trace
18      that we will accumulate. Used for scheduling. *)
19   local_memory: ISet.t * Trace.t;
20   kern_id     : kern_id;
21   meta_loc    : Val.meta_location;
22   resolve_loc : Val.loc;
23 }
24 type device_box = {
25   (* The set of warps that make up this device box *)
26   warps: (warp_pool ref) SMap.t;
27   (* Substitution from meta locations to locations that we in scheduling *)
28   qenv : Val.loc SMap.t;
29   kerns: SSet.t;
30 }
```

With these type definitions, we can then define the function that builds the thread graphs for a Λ^{G} expression:

```
let rec build_thread_graph (env: env) (e: Ast.t): thread_graph * Val.t
```

Which just as with the function that builds the cost graphs for Λ_{C} returns a cost graph—in this case a *thread* graph—along with the value.

As a simple example on how we build these thread graphs, recall the multiplication rule from Figure 4.9

$$\frac{\begin{array}{cc} \mathsf{K};\mathcal{S}\Big|_E\, e_1 \Downarrow^{\mathsf{t}} v_1; c_1\Big|\,\mathcal{S}_1;\mathsf{K}_1 & \mathcal{S}_1; v_1 \uparrow i_1 \\ \mathsf{K}_1;\mathcal{S}_1\Big|_E\, e_2 \Downarrow^{\mathsf{t}} v_2; c_2\Big|\,\mathcal{S}_2;\mathsf{K}_2 & \mathcal{S}_2; v_2 \uparrow i_2 \end{array}}{\mathsf{K};\mathcal{S}\Big|_E\, e_1 * e_2 \Downarrow^{\mathsf{t}} i_1 * i_2; c_1 \oplus [\mathtt{reg}] \oplus c_2 \oplus [\mathtt{reg}] \oplus [\mathtt{mul}]\Big|\,\mathcal{S}_2;\mathsf{K}_2}$$

which corresponds to the following pattern match clause in the definition of `build_thread_graph`

```
| A.Mul(e1, e2) →
  let g1, v1 = build_thread_graph env e1 in
  let g2, v2 = build_thread_graph env e2 in
  let cost, v = v_mul v1 v2 in
  g1 @ nomem :: g2 @ nomem :: [Mul], v
```

When dealing with `parmaps`—i.e. kernels—however, we need to have access to a (global) kernel pool `kern_pool` that we can add to, that is a mapping from (unique) `kern_id`s to `kern_pool_entry`s. We then add kernels effectfully to this global kernel pool via an `add_kernel` function. With this knowledge we can present the rule for `parmaps` in `build_thread_graph`;

```
| A.ParMap(e, arr, (Reg.R reg)) →
  let kid = new_kern_id () in
  let uid = Reg.fresh () in
  let mloc = Val.{ uid; region_name = Reg.RName reg; } in
  let kern_entry = { func = e; arr = arr; mloc = mloc; } in
  add_kernel kid kern_entry;
  [Kern kid; MetaPut mloc], Val.VMLoc mloc
```

Which can be readily read off from the `parmap` rule in the cost semantics:

$$\frac{r = E(\rho) \qquad (\blacksquare\ell, \kappa \text{ fresh})}{\mathsf{K}; \mathcal{S} \Big|_E \ \texttt{parmap}\ f\ [e_1, \ldots, e_n]\ \ \texttt{at}\ \rho \Downarrow^{\mathsf{t}} \blacksquare\ell; [\mathsf{kern}[\kappa]] \oplus_{\blacksquare} [\mathsf{mem}[\blacksquare\ell]] \Big| \mathcal{S}; \kappa \hookrightarrow (f, [e_i], \blacksquare\ell) \uplus \mathsf{K}}$$

Similarly to the cost semantics where we had a rule for creating warp pools in Figure 4.11, we also have a function that reflects this rule—that takes an entry in the kernel pool and builds a warp pool for the kernel pool entry:

```
let build_warp_pool (env: env) (kern_id: kern_id) (kern: kern_pool_entry):
    warp_pool * Val.t
```

Since the kernel pool is global, we also have a function that takes any kernels that have been added to the kernel pool and generates a warp pool for each kernel entry in the kernel pool—and iteratively keeps doing so until no more entries remain in the kernel pool. This function is defined by the rule for device boxes in Figure 4.11 that takes an environment and keeps generating warp pools until no more entries remain.

Note that this iterative process is required since kernels can spark sub-kernels—thus the building of one warp pool (which represents a kernel) could then cause the creation of another warp pool (another kernel). This is given by the following function:

```
let rec build_device_box (env: env): device_box * Val.t =
  match SMap.min_binding_opt !kern_pool with
  | None → {warps = SMap.empty; qenv = SMap.empty; kerns = SSet.empty;}, Val.VInt
    1
  | Some (kern_id, kern_entry) →
    (* Remove the kernel from the kernel pool *)
    kern_pool := SMap.remove kern_id !kern_pool;
    (* Build the rest of the warp graph *)
    let db, _ = build_device_box env in
    let warp_graph, resolved_loc = build_warp_pool env kern_id kern_entry in
    { warps = SMap.add kern_id (ref warp_graph) db.warps;
      qenv = SMap.add kern_entry.mloc.Val.uid (loc_of_val resolved_loc) db.qenv;
      kerns = SSet.empty;
    }, resolved_loc
```

Importantly, this function does not take a kernel pool entry to construct the device box for the kernel—recall that the kernel pool is a global mutable pool that is effected.

6.3.3.1 Scheduling

Once a device box, or device boxes have been built, we can schedule and profile the device box based upon the theory in Section 4.6.

In order to perform the scheduling of a set of thread graphs as defined in Section 4.6.1 we create a module with the following signature:

```
1  module ThreadSched = struct
2    (* We distinguish the type of schedulable v.s. considerable nodes even
3       though they have the same underlying representation. *)
4    type sched_step = instr list
5    type cons_step = instr list
6    (* Subst for quiescence over a thread graph *)
7    let subst qenv tgraph =  ...
8    (* Quiesce a thread graph *)
9    let quiesce (w: warp_graph) (qenv: Val.loc SMap.t): warp_graph = ...
10   let considerable_nodes (w: warp_graph): cons_step = ...
11   let schedulable_nodes (w: warp_graph): sched_step = ...
```

```
12   let schedulable_nodes_from (considerable_nodes: cons_step): sched_step = ...
13   let is_finished (w: warp_graph): bool = ...
14   (* Def 4.6.2 *)
15   let is_unrunnable (w: warp_graph): bool = ...
16   let perform_step_w (w: warp_graph) (nodes: sched_step): warp_graph * sched_step =
       ...
17   let perform_step (w: warp_graph): warp_graph * sched_step = ...
18 end
```

The above module can be seen as accomplishing the task described as "given a warp graph, perform one step of that warp graph." However, since we can have many warps that need to run in a specific order in order to agree with our abstract machine model, we also need to have a module that handles warp pools. With this viewpoint, we implement the scheduling of these warps in the context of a warp pool in the following module. It takes care to move the focus to the correct warp after performing a warp step, and then calls out to the `ThreadSched`uler to perform the actual step for each thread within that warp graph.[8]

```
1 (* Scheduling of a warp pool *)
2 module WarpPoolSched = struct
3   let is_finished (w: warp_pool): bool = ..
4   let quiesce (wp: warp_pool) (qenv: Val.loc SMap.t): warp_pool = ...
5   let perform_step (w: warp_pool): warp_pool * ThreadSched.sched_step = ...
6 end
```

Since a kernel is represented as a warp pool, this then provides us a way to run a *single* kernel. However, we also need to be able to deal with multiple kernels, and particularly sub-kernels. This is performed by the `KernSched` module that handles the scheduling of multiple kernels based upon the theory in Section 4.6.1:

```
1 module KSched = struct
2   (* Determine if all computation for a device box has run. *)
3   let is_finished (d: device_box): bool = ...
4   (* We first update the qenv for the DB, and then we perform quiescence
5    * operations on the rest. *)
6   let quiesce_db (d: device_box) (finished_kerns: warp_pool list): device_box = ...
7   let perform_step (d: device_box): device_box * ThreadSched.sched_step list = ...
8   let run (d: device_box): (ThreadSched.sched_step list * SSet.t * SSet.t) list =
      ...
9 end
```

[8] Recall that the `warps` field within the definition of a `warp_pool` are mutable.

With the above definitions on how to schedule device boxes, we now turn to the implementation of memory on the GPU—both representing the memory, and the different ways that regions can be loaded into work-group local memory.

6.3.3.2 Representing GPU Memory

Unlike CPU memory in the cost model we have built, in which we don't have to worry about that many factors that can affect the overall cost of the computation, with GPU memory we need to factor in such things as global versus local memory latency, bank width, and word width. Since the costs that we calculate depend upon bank conflicts or broadcasts, the latter two points can play an important factor in the calculation of costs. In order to do this, we implement GPU memory as we did CPU memory in Section 6.3.1 via an OCaml class with the following signature.

```
1  class gpu_memory = object
2    val mutable local_regions: SSet.t
3    val mutable offsets : int SMap.t
4    val mutable gpu_mem: Memory.region_stack
5    val mutable bank_width : int
6    val mutable global_latency : int
7    val word_width
8
9    method set_bank_width (w: int): unit
10   method set_global_latency (l: int): unit
11   method get_bank_width: int
12   method get_global_latency: int
13   method load_mem_local (regs: Reg.name list) (rstack: Memory.reg_stack): unit
14   method load_mem (regs: Reg.name list) (rstack: Memory.reg_stack): unit
15   method is_local_loc ((Reg.RName rname, offset): Val.loc): bool
16   method is_local (instr: instr): bool
17   method calc_offset ((Reg.RName rname, offset): Val.loc): int
18 end
```

This class serves as a wrapper around a region stack (`gpu_mem`) where we abstractly keep track of which regions reside in local/shared memory (via the `local_regions` field) without loading these regions into a separate of section of memory—we can then query whether or not instructions (particularly `puts` or `gets`) access local memory or not, and whether given locations are local too. With this representation of the

local memory, representing different loading strategies becomes fairly straightforward—all it requires is changing the regions that are added to the `local_regions` set, and we don't actually have to change the memory. This aspect is crucial, since with this setup we can build up a number of traces with the schedules that we defined above, and then vary which regions are loaded into local memory. Because of this when it comes to determining the cost of a trace we can simply ask which loading configurations are best for that computation easily and relatively efficiently—all we have to do is change the set of local regions and re-analyze and re-cost the traces.

6.3.3.3 Traces and Costs

Recall from Section 4.7 that the actual determination of the cost of the computation is done by analyzing the memory traces that are built up from scheduling the device box. So before we can detail the implementation of the cost analysis for these traces, we first need to detail the implementation of the traces based upon those we saw in Sections 4.5.2 and 4.7.0.1.

```
module Trace = struct
  type t = instr list
  let put (trace: t) (l: Val.loc) : t = ...
  let get (trace: t) (l: Val.loc) : t = ...
  let has_put (trace: t) (l: Val.loc): bool = ...
  let has_get (trace: t) (l: Val.loc): bool = ...
  let empty: t = []
  let weak_eq (instr1: instr) (instr2: instr): bool =
    match instr1, instr2 with
    ...
    | Kern _      , Kern _
    | Put _       , Put _
    | MetaGet _   , MetaGet _
    | Annotated _, Annotated _ → true
    | _ → false
end
```

With this definition of a trace, we can then implement the cost analysis over these traces with the following module that is parameterized by the set of local regions—since the costing of traces in Section 4.7 is parameterized by the set of regions that

are loaded into local memory. The cost analysis of these memory traces is implemented by the following module definition:

```
module AnalyzeTrace = struct
  let locs_of_trace (trace: Trace.t): Val.loc list =
  let is_bcast (mem: gpu_memory) (locs: Val.loc list): bool =
  (* We return 0 here to signify that there are no bank conflicts *)
  let is_bconf (mem: gpu_memory) (locs: Val.loc list): int =
  let local_addrs (mem: gpu_memory) (trace: Val.loc list): Val.loc list =
  let global_addrs (mem: gpu_memory) (trace: Val.loc list): Val.loc list =
  (* get ops and form the equiv classes (cf. simultaneously scheduled nodes in
    4.6.1) *)
  let ops_of_trace (trace: Trace.t): int =
  (* EQ. 4.20 *)
  let trace_cost (mem: gpu_memory) (trace: Val.loc list): int =
  (* EQ. 4.21 *)
  let step_cost (mem: gpu_memory) (step: WSched.sched_step): int =
  let device_box_cost (mem: gpu_memory) (l: (WSched.sched_step list * SSet.t * SSet.
    t) list): int =
end
```

6.4 Primitive Cost Providers

While this dissertation is concerned with inferring abstract costs for programs, there is an important distinction that needs to be made between internal and external costs. A parallel can be drawn between these external costs—that are given by PCPs—and the IO monad in Haskell: the IO monad allows a pure program to interact with the outside world, and the PCPs allow our costs to interact with actions outside the scope or world of the program itself. Further, these external costs are ones that are implementation dependent, and therefore need to be 'lifted' from the implementation (this chapter) into the theory that we have developed in the last two chapters. As opposed to internal costs which are not (as) dependent upon the outside world.

This parallel of PCPs and interaction can be seen in the PCPs that we have used throughout this dissertation: the Δ^a PCP for allocation of regions—*interaction* with the memory allocator outside of our computational world—the PCPs $\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R})$ and $\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R})$ for transferring regions between the host and device—*interaction* with the MMU for

the GPU and the CPU, along with the hardware memory bus—and finally the PCP $\mathsf{cost}_{\mathsf{merge}}(R, R')$ for merging regions together when transferring a region from device to host.

There are a number of ways to determine how much a primitive operation should cost—and hence what the PCP should be for the primitive—and there are a number of considerations that need to be taken into account in determining just how we should derive the PCP cost since each of them has various upsides and downsides. The three main possibilities for determining them are the following:

- **Analytic:** We know the time taken by a given operation such as the rate at which memory can be transferred across the memory bus, or the time it takes to perform one add or multiplication—call it T—and we could then define the cost of the PCP based upon this e.g. in the case of memory transfers the cost would be the sum of the sizes of the regions being transferred divided by the rate of transfer $\sum_i |R_i| / \mathsf{T}_{\mathsf{memTfer}}$. However, using such a method does not allow us to take into account such things as that transferring a number of non-contiguous segments of memory is slower than transferring a number of contiguous segments.

- **Empirical:** In this case we learn the cost of these transfers using a small and simple set of cost semantic rules and the methods presented by Das and Hoffmann [68] to perform linear regression and fill in the cost holes of the cost semantics for the PCPs in order to give an accurate costing method. However this has the downside of both being quite heavyweight, and suffers from not being abstract enough; since the other non-PCP generated costs are abstract we need to make sure that we do not overweight the inferred costs with respect to PCPs which may very well happen if we determine PCPs in this manner, or even if we determine them analytically.

- **Fully Abstract:** In this setting, we do not seek to tie the cost of these PCPs to the cost of the underlying operation at all—just as we said that things such as addition or the like have an abstract cost of 1, we could likewise say that a PCP for e.g. memory allocation will have abstract cost 0.1 times the number of bytes in the allocation. However, this has the problem of being *too* abstract—since we lose all track of reality in the process.

As we see there are a number of possibilities that each have their own advantages, and disadvantages. In order to get the best of everything—a PCP that is abstract enough to not skew the cost analysis of the programs too much towards memory allocation and transfers (in either direction), yet is accurate enough to be useful—we take a hybrid approach to the determination of PCPs: we first scale the analytic cost of primitive operations for which we don't need a PCP–such as addition and multiplication—and then divide this analytic cost by the abstract cost as defined by our cost semantics to arrive at *abstract to primitive ratio* (ATPR) or scale factor. Since the analytic cost for things such as memory allocation or memory transfers is not always readily available, we will then use an empirical approach to gather the costs for memory allocation and transfers. However, in the case where constants are already known these could be used just as well too.

In the next portion of this section we will detail the exact methods we use to determine each PCP that we have needed throughout this dissertation: Δ^a for region allocations; $\mathsf{cost}_{\mathsf{merge}}(R, R')$ for region merging; and finally $\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R})$ and $\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R})$ for transferring regions between computational worlds. But first, we detail the derivation of an ATPR.

6.4.1 Deriving the ATPR

The ATPR is a way for us to relate the real-world costs of an operation to the inferred abstract cost by measuring the ratio between the inferred abstract costs from our

cost semantics, and the actual operations for primitives in our language that do not rely on PCPs—thus we must determine the ATPR from primitive operations that do not involve any memory operations such as e.g. $+$ and $*$.

Since the analytic cost for such primitive operations are generally well known for most, if not all, hardware configurations we can safely assume the existence of constants for our primitive operations: $\mathsf{T}_{\mathsf{mul}}, \mathsf{T}_{\mathsf{add}}, \mathsf{T}_{\mathsf{less}}$ etc. Furthermore since we assign the abstract cost of 1 to each of these primitives in our cost semantics, the ATPR for each primitive is quite easily determined once we have these analytic costs; it is simply the reciprocal of the analytic cost $\frac{1}{\mathsf{T}_{\mathsf{op}}}$.[9]

Once we have the ATPR for each primitive operation in the language, we then take the average of these operations to arrive at the ATPR that we will use for the second part of the process—scaling the cost constants inferred from the empirical cost semantics. However, before doing this we need to detail just how we determine the empirical costs.

Allocation Costs The measurement of the empirical costs for allocations is inferred by taking the average of 100 allocations of two types: the first being allocation of a new region of memory Δ^a, and the second, expansion of regions for Δ^s. Since we always allocate and expand our regions by a minimum size given by $\mathsf{allocSize}$, the time for each allocation is measured when allocating or expanding by $\mathsf{allocSize}$. The definitions of both of these are given by the following underlying cost rules[10]

$$\frac{\mathsf{alloc}\ (\mathsf{allocSize})}{\mathit{newRegion}(\mathsf{allocSize}) \Downarrow^{l} \tilde{c}^{\ \mathsf{alloc}}} \qquad \frac{\mathsf{realloc}\ (R, \mathsf{allocSize})}{\mathsf{expand}\ (R, \mathsf{allocSize}) \Downarrow^{l} \tilde{c}^{\ \mathsf{realloc}}} \tag{6.5}$$

[9]Since the cost of an operation cannot be zero, we don't need to worry about division by zero issues in the taking of the reciprocal.

[10]We denote cost semantic rules in which we need to learn the costs based upon linear regression with the operator $\Downarrow^{l}$.

With these we can then define both delta Δ^a and Δ^s as follows—since we always allocate and expand in allocSize chunks.

$$\begin{aligned} \Delta^a(n) &\triangleq i * \tilde{c}^{\,\mathsf{alloc}} && \text{where } i = \lceil n/\mathsf{allocSize} \rceil \\ \Delta^s(n) &\triangleq i * \tilde{c}^{\,\mathsf{realloc}} && \text{where } i = \lceil n/\mathsf{allocSize} \rceil \end{aligned} \tag{6.6}$$

While the determination of the empirical costs for allocation are quite straightforward, the determination for memory transfers is a bit more complicated. We can formally state what we need easily: given a set of region $\{R_1, \ldots, R_n\}$ on the host, we want to determine the cost of transferring these regions to the device. And, similarly, the other direction from device to host. The problem arises in the non-compositionality of the costs for this cost—the cost of transferring a number of regions is not necessarily the sum of the cost of transferring each individual region. We therefore need to get a bit more nuanced in our inference of the cost for this operation.

Naively, we could state that if transferring a region has a cost c, then transferring a set of regions $\{R_1, \ldots, R_n\}$ would have the cost

$$\left\lceil \sum_i |R_i| * c \right\rceil$$

however, when transferring multiple regions, the entire transfer can result in a lower cost per region. Since we do not want the cost function or determination to be very complicated, this problem can be seen as asking the question: given a per-region cost c and a set of regions to transfer, derive a suitable constant c' such that the cost of the transfer of the regions is given by the following sum:

$$\left(\left\lceil \sum_i |R_i| * c \right\rceil \right) + c'$$

With this in mind we arrive at the following cost semantics to infer the cost of these transfers (in the 'style' of Das and Hoffmann [68]). The first, for determining the

cost of a single memory transfer from host to device, and the second for transferring memory from device to host:

$$\frac{\texttt{memTransferToDevice}(\mathsf{R},\mathsf{R}')}{\mathsf{transfer}^{\mathsf{G}}(R) \Downarrow^{\mathsf{l}} \tilde{c}^{\,\mathsf{mdev}}} \qquad \frac{\texttt{memTransferToHost}(\mathsf{R}',\mathsf{R})}{\mathsf{transfer}^{\mathsf{C}}(R) \Downarrow^{\mathsf{l}} \tilde{c}^{\,\mathsf{mhost}}} \tag{6.7}$$

With this, we can then determine the cost for the transfer of the set of regions

$$\frac{\mathsf{transfer}^{\mathsf{G}}(R_i) \Downarrow^{\mathsf{l}} c_i \qquad R_i \in \overline{R}}{\mathsf{transfer}^{\mathsf{G}}\left(\overline{R}\right) \Downarrow \left(\sum_i c_i\right) + \tilde{c}^{\,\mathsf{tdev}}} \qquad \frac{\mathsf{transfer}^{\mathsf{C}}(R_i) \Downarrow^{\mathsf{l}} c_i \qquad R_i \in \overline{R}}{\mathsf{transfer}^{\mathsf{C}}\left(\overline{R}\right) \Downarrow \left(\sum_i c_i\right) + \tilde{c}^{\,\mathsf{thost}}} \tag{6.8}$$

where both $\tilde{c}^{\,\mathsf{tdev}}$ and $\tilde{c}^{\,\mathsf{thost}}$ are cost constants that need to be learned.

In order for the cost functions to take into account such things as transferring two regions that are next to each other in memory, before using both $\mathsf{transfer}^{\mathsf{C}}\left(\overline{R}\right)$ and $\mathsf{transfer}^{\mathsf{G}}\left(\overline{R}\right)$ to determine the cost of the $\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R})$ and $\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R})$ PCPs respectively, we first need to determine any regions that reside next to each other in memory, and treat both of regions as a single region when determining the cost from the underlying transfer cost function. To do this, we first define a predicate $\mathsf{isNbr}(R,R')$ that given two regions determines if the two constitute a contiguous section of memory.

Definition 6.4.1. *Given two regions R and R' we define the predicate $\mathsf{isNbr}(R,R')$ to be the following*

$$\mathsf{isNbr}(R,R') = (base + s + \mathsf{word_size} \equiv base') \vee (base' + s' + \mathsf{word_size} \equiv base)$$

where $R = (base, \mathsf{ap}, s, \mathsf{refcnt}, v)$, $R' = (base', \mathsf{ap}', s', \mathsf{refcnt}', v')$, and $\mathsf{word_size}$ is the alignment being used.

and in the case that two regions of memory R_1 and R_2 neighbor each other, we denote this by $R_1 \boxplus R_2$ and if the two regions are not neighbors then nothing is returned.

With these definitions, we can now define the PCPs for transferring memory. Given a set of regions $\{R_1, \ldots, R_n\}$ to transfer, we first combine all regions that are neighbors:

$$\mathsf{coalesce}\left(\overline{R}\right) = \{R \mid R, R' \in \overline{R}\ .\ R \boxplus R'\} \cup \{R \mid R \in \overline{R}\ .\ \nexists R' \in \overline{R}\ .\ \mathsf{isNbr}(R,R')\} \tag{6.9}$$

After we have coalesced neighboring regions in this manner, we then call the respective underlying cost functions. We thus have the following definitions for transfer functions

$$\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R}) = \mathsf{transfer}^{\mathsf{C}}\left(\mathsf{coalesce}\left(\overline{R}\right)\right) \tag{6.10}$$

and

$$\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R}) = \mathsf{transfer}^{\mathsf{G}}\left(\mathsf{coalesce}\left(\overline{R}\right)\right) \tag{6.11}$$

Merge Costs The merging of two regions R_1 and R_1' can be seen as a coupling of two operations—first possibly expanding R_1 to take into account the "delta" from the original parent region, and then copying this delta to the parent region.[11] Since in reality the copies from the GPU to the CPU are "sharded" in the sense that only the new portion of memory is copied back from device, and since we are already taking into account the cost of transferring the memory, the merge cost is simply the cost of a possible reallocation. Since the memory transfer places the memory directly into the preallocated memory, the cost of any copying that needs to be done is is covered by the transference cost. Thus the final costing of this is simply that of possible reallocation. [12]

We thus arrive at the following definition for the $\mathsf{cost}_{\mathsf{merge}}(R, R')$ PCP, where we by convention place the device region on the RHS.

$$\frac{v = R.v \setminus R'.v \qquad i = \mathsf{calcExpand}\,(R, v) \qquad \mathsf{expand}\,(R, i) \Downarrow^{\mathsf{l}} c}{\mathsf{cost}_{\mathsf{merge}}(R, R') \Downarrow^{\mathsf{l}} c} \tag{6.12}$$

and where $\mathsf{calcExpand}\,(R, v)$ is defined as follows

$$\mathsf{calcExpand}\,(R, v) \triangleq \begin{cases} i \,.\, i | \mathsf{allocSize} \wedge (i > |v| - (|R| - R.\mathsf{ap})) & \text{if } |v| > |R| - R.\mathsf{ap} \\ 0 & \text{otherwise} \end{cases}$$

[11] We may be tempted to argue that reallocation is not needed, however while the region cannot expand on the GPU, since we allow parallel access (and allocation) to the parent regions while the GPU is running, this may be allocated into and there may not be space left when transferring back to the host.

[12] Note that it is *crucial* that we are merging regions in which one is a parent of the other.

that returns the amount that the region needs to be expanded in order to handle the new memory that needs to be placed in it at the granularity of $\mathsf{allocSize}$.

Kernel Scheduling Costs We also need to determine the abstract cost κ_{sched} for scheduling a kernel to run on the GPU. Since we already take into account the memory transfer overhead between host and device, this PCP is meant to represent only the *fundamental overhead* of launching a kernel. In order to measure this overhead we measure the overhead of launching an empty kernel with no memory transfers or computation on the GPU, and which we will write as $\mathsf{kernLaunch}()$. The rule for determining κ_{sched} is given by the following rule:

$$\frac{}{\mathsf{kernLaunch}() \Downarrow^{l} \kappa_{\mathsf{sched}}} \tag{6.13}$$

Since all of the semantic rules that we have presented for determining our PCPs are straightforward, and since the implementation and learning of costs $\tilde{c}$ follows immediately from Das and Hoffmann [68], we do not go into detail on the precise implementation of these rules. We now turn our attention to evaluating the efficacy of the cost theory by evaluating it on some example programs.

6.5 Evaluation

Since the costs that we are dealing with are *abstract*, evaluating the efficacy of the cost semantics in terms of predicting exact runtime costs of programs or space usage would be incorrect. Instead, we are interested in measuring the asymptotic costs or bounds of programs using these semantics, and how these asymptotic bounds and costs differ between using the CPU and GPU to perform the computation. To this end we demonstrate the efficacy of our cost semantics as they are implemented in the compiler on a small set of example programs: parallel and serial Fibonacci; and vector addition of various sizes. Each of the programs that we have used is designed

to demonstrate a particular aspect of the cost semantic system that we have developed. In particular, that we are able to determine the following key characteristics of input programs:

- We are able to determine when the input is too small to effectively utilize all of the available task parallelism on the CPU.
- We are are able to determine that programs which on their surface may seem amenable to GPU acceleration, actually will be more costly to run on the GPU compared to computing locally on the CPU. Further to this, we are also able to calculate at which stage the input becomes large enough for the movement of the computation to the GPU to be useful.
- We are able to accurately determine the space tradeoffs with GPU acceleration.

Each program in the evaluation below was run on a Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz CPU and NVIDIA GeForce GT 750M with 2048MB VRAM GPU.

Figure 6.1 shows the calculated runtimes and space usages of two different versions of the parallel Fibonacci function: one using only one region for the entirety of the computation—and thus allocating all closures in one region of memory for the entire computation; and the other using a new region to allocate the closure at each recursive call. We see the following computational properties from these graphs: first, we see that even though the allocation overhead of creating and deleting a region for each recursive call imposes a significant overhead to the computation, as the number of threads grows, this overhead becomes less apparent—which makes sense since allocation latency can be masked on one thread while other threads perform useful computation. We also see that as the thread numbers grow, the size of the single region grows as the number of threads grow—this also holds with reality; the frontier of the the computation grows considerably in its possible width as the number of threads grows, and therefore the number of active nodes in the graph grows–along with the space that these thread contexts take up. Furthermore, this

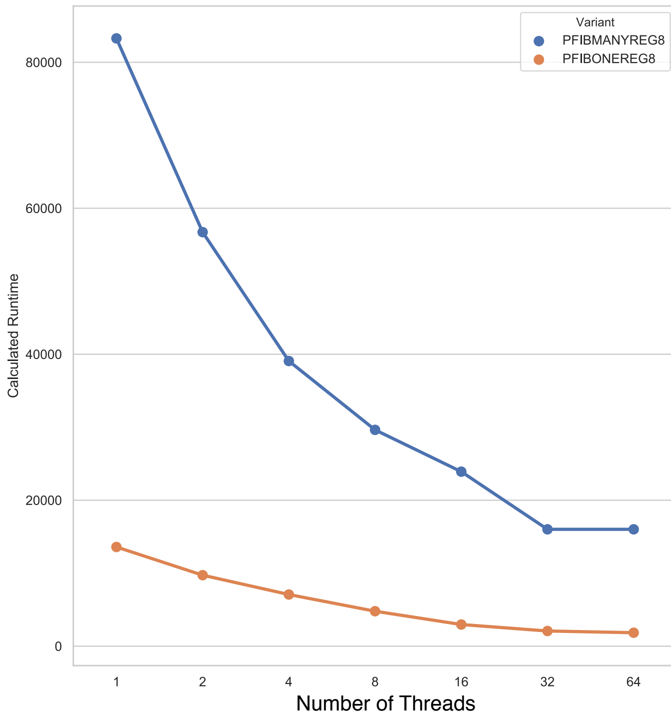
is compounded by the fact that the memory is never deallocated, and thus even if the amount of memory used may only grow a little bit at each step, over the course of the computation this amounts to a considerable amount of memory; the size of the memory is not just the “high water mark” but the entirety of the memory allocated.

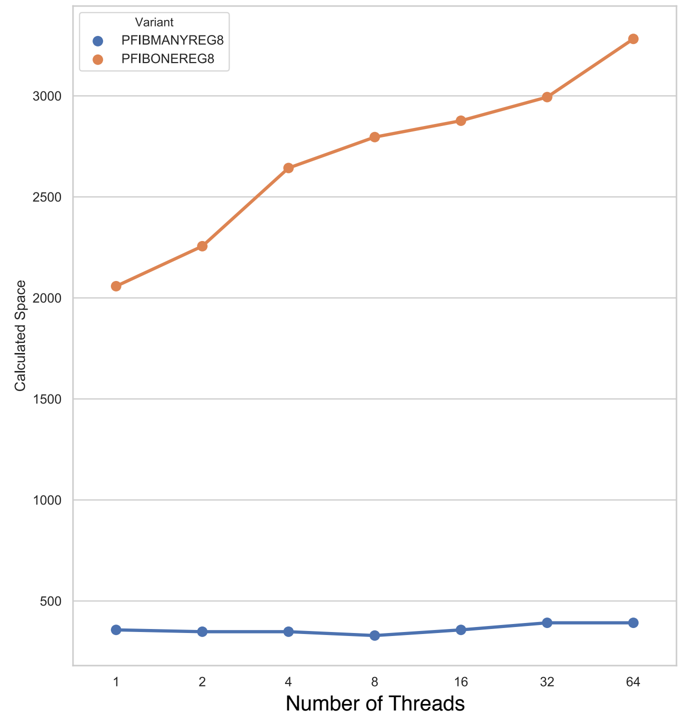
As compared to the single-region function, the size of the memory used in the many-region version grows only slightly—while the width of possible threads may increase, we also remove old memory as function calls return. Thus the space used is not compounded as significantly as in the single region version of the code; the memory usage here is only the “high water mark” of the memory used during the computation.

Figures 6.2 and 6.3 compare the use of `parmap` on both a fixed size array with a varying number of threads (Figure 6.2) and a varying-size array with a fixed number of threads. We also see a couple other key properties from these graphs, in particular: in Figure 6.2 we are able to determine that the array is too small for thread-based (CPU-task-based) parallelism to be effective—showing a speedup of less than 19% from 1 thread to 64 threads; and, the system is also able to determine that the array is too small to be amenable to GPU acceleration—and in particular that the transfer costs would far outweigh any computational benefits.

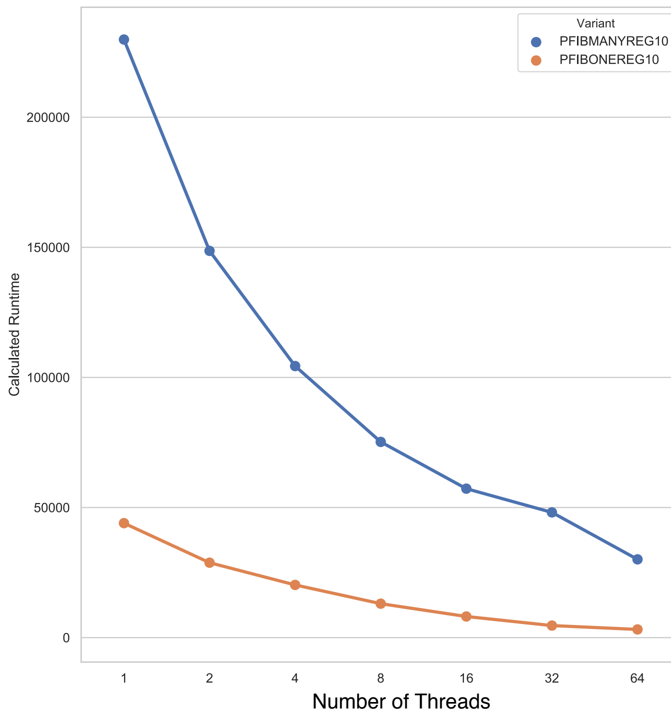
In the second of these graphs in which we vary the size of the input array as opposed to the number of threads, we also see that the asymptotic growth of the computation grows significantly slower for the GPU addition as compared to the CPU-based computation. In particular, while the CPU computation grows in a linear manner with the size of the input the predicted GPU performance for this code is sub-linear. We also see in this code, that the additional memory required by the GPU to store and compute over the vector that the size of the GPU-based program approximately doubles in size to that of the CPU-based program.

Parallel Fibonacci with N = 8





Parallel Fibonacci with N = 10



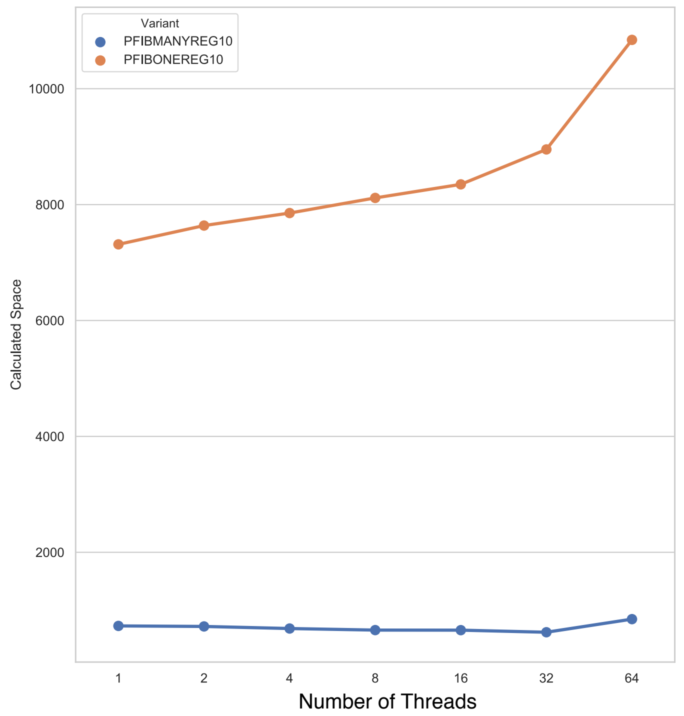


Figure 6.1: Calculated computational and space usage for the parallel Fibonacci functions. One using only one region, and the other using a new region at each recursive step.

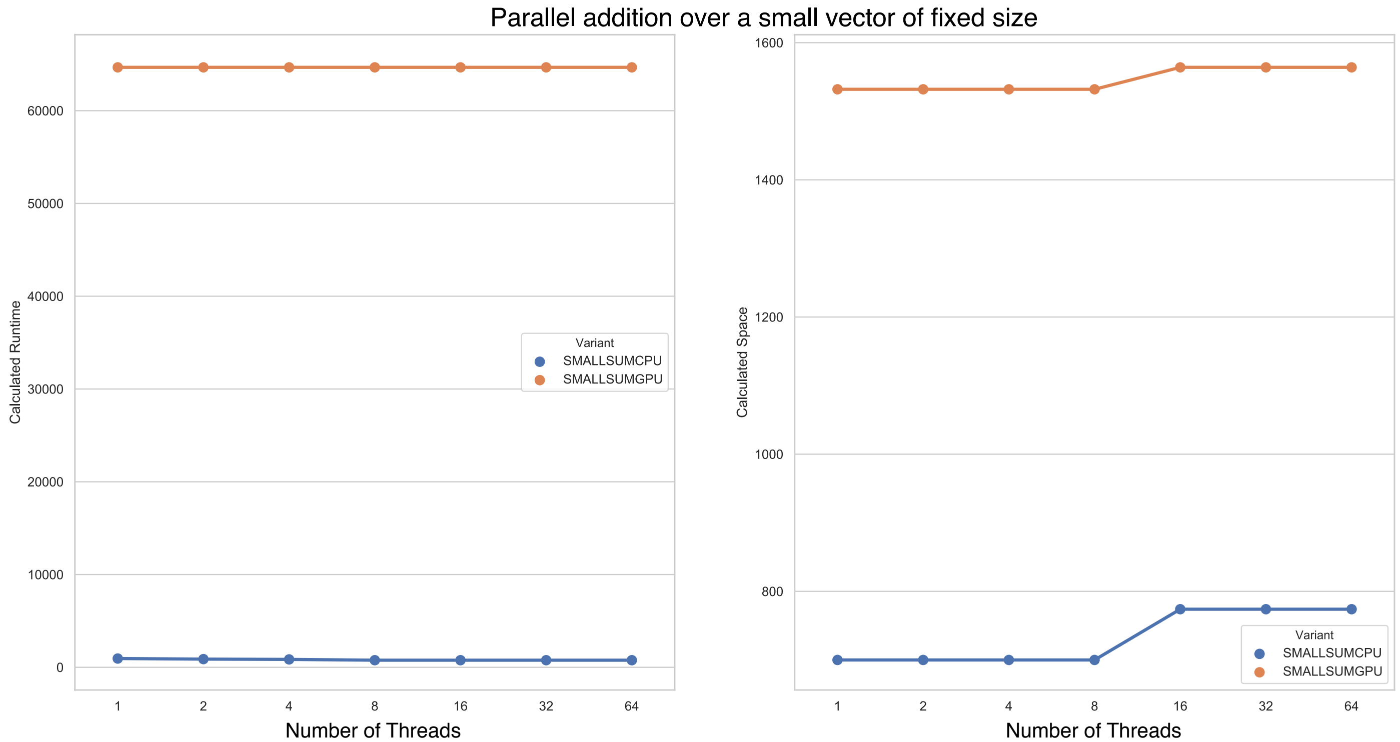


Figure 6.2: Calculated computational and space usage for parallel addition over a small fixed-size array. Notice that we rightly determine that the cost of running on the GPU is much more costly, and similarly, that the useful amount of parallelism available to us on the CPU as we use more thread does not help due to the computational overhead of these threads.

We also compare two different types of vector addition on the GPU; one in which we simply add the two numbers, and another in which we perform a simple branching operation based upon the data from the vector–this will then cause a warp-divergence on the GPU code. We can see the results of this in Figure 6.4. As we see while the space is exactly the same for the two, their runtimes differ by a noticeable amount; about 6%. While this amount is not too great, this divergence amount is to be expected; the length of any given divergent path on the GPU has only length one and thus we at most would need to perform twice as many warp steps for any given warp step.

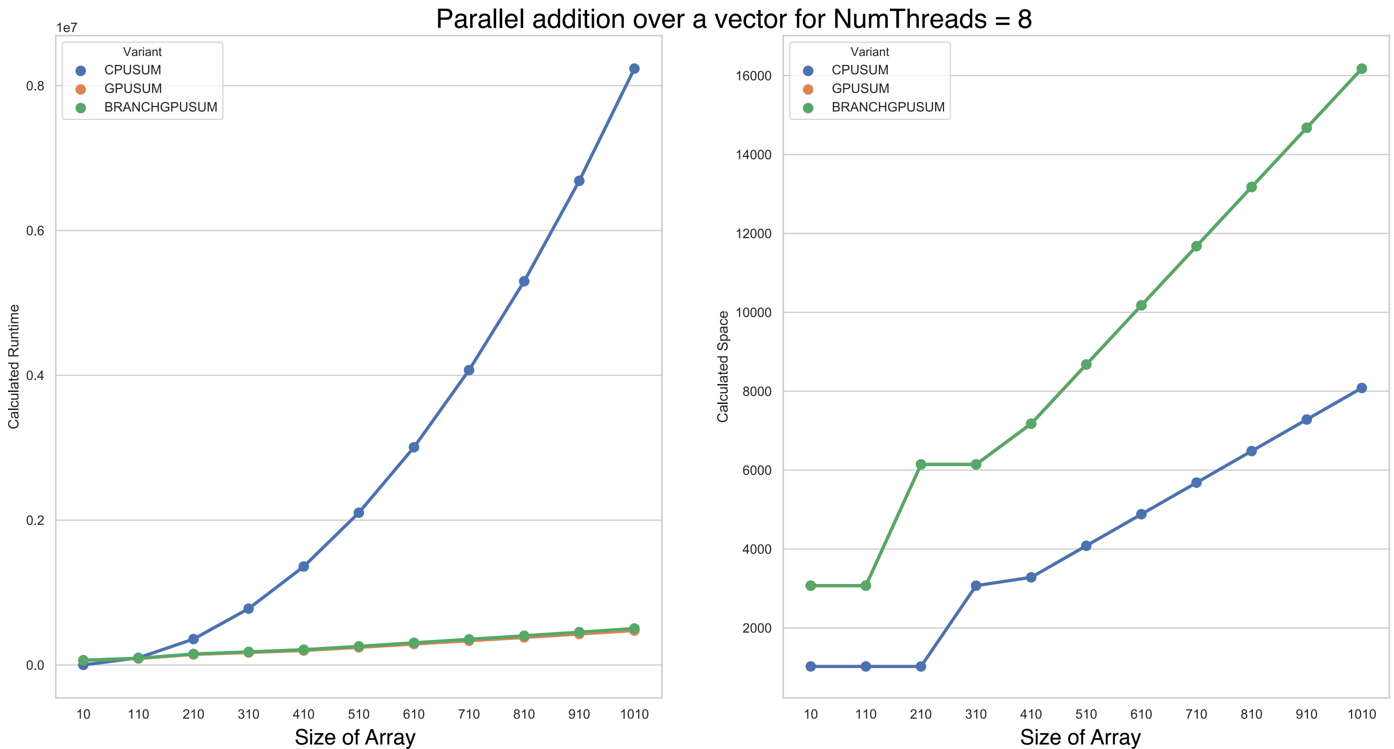


Figure 6.3: Calculated computational and space usage for the parallel addition of vectors. Notice that the CPU graph grow almost linearly with the size of the array (recall that the threads are fixed), while the GPU code is predicted to stay almost constant in comparison.

6.5.1 Comparing to Real-World Runtimes

In this section and in Figures 6.5 and 6.6, we compare the abstract numbers that we generated in the previous section, to the actual runtimes, and space usages of these same programs that we observe when running them. It is important to take note that the evaluation takes place in terms of *ratios* of costs as opposed to real-world costs. This is since while our cost semantics are meant to represent the costs of programs, these costs are abstract. Therefore in order to handle this issue of examining the accuracy of the predicted abstract costs with real-world costs, we use the fact that by comparing the ratios of runtimes the underlying (relative) magnitudes of costs largely cancel out, and we are left with the actual behavior of one program with respect to another. Thus, to determine the accuracy of the profiling framework we plot

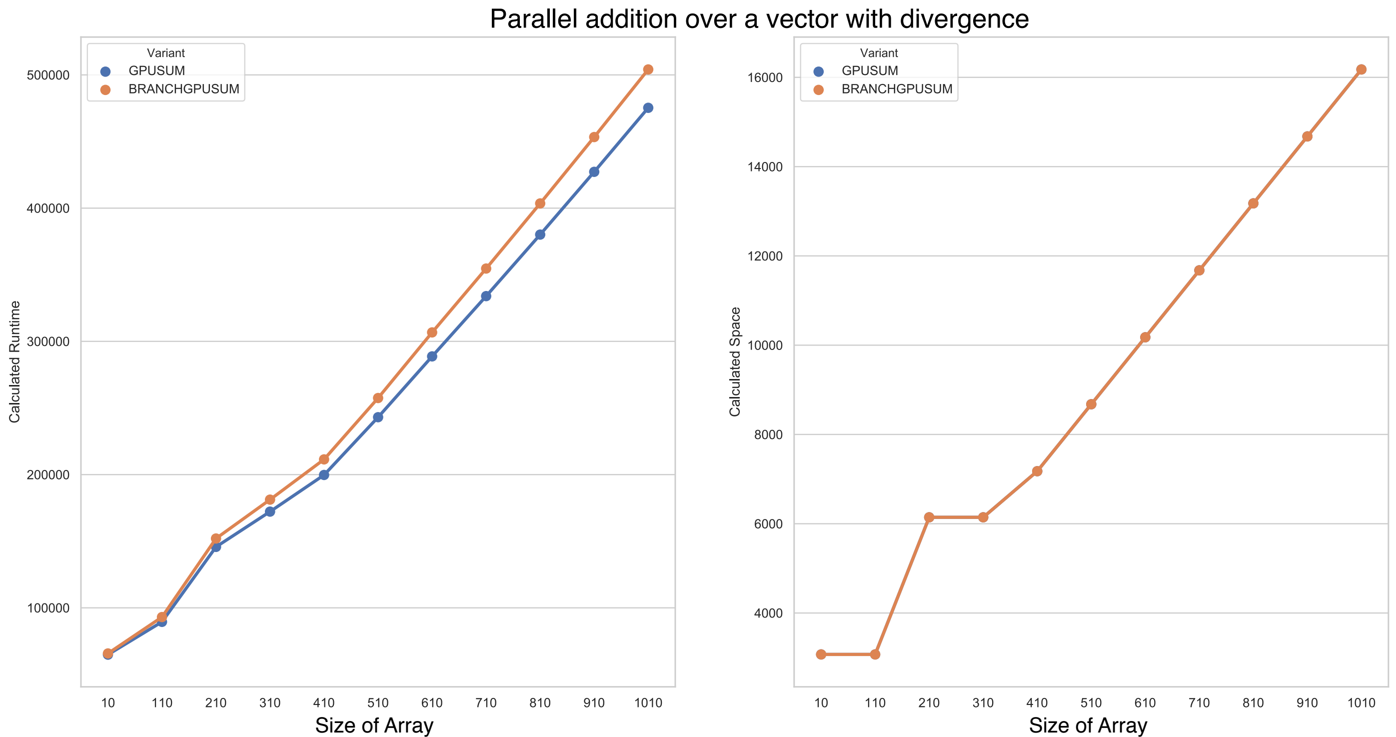


Figure 6.4: Calculated computational and space usage for the parallel addition of vectors on the GPU. One (BRANCHGPUSUM) uses a branching operation during the summation, to perform addition of a different constant based upon parity of the elements. The other (GPUSUM) simply performs straightforward addition.

the ratio of $abstractCost(p_1)/abstractCost(p_2)$ for two program variants p_1 and p_2, and plot the same ratios $realWorldCost(p_1)/realWorldCost(p_2)$ for each of the program's observed runtimes. We use the programs that we looked at in the previous section for this comparison as well.

Figure 6.5 presents the plots for the runtime and space usage of the parallel Fibonacci function variants that we saw previously in Figure 6.1. However, this time instead of plotting the runtime or predicted runtimes and space usage, we instead plot the ratio of the predicted runtimes and space usage of the two different Fibonacci functions, along with the plot of the ratios of the observed runtimes for these same functions.

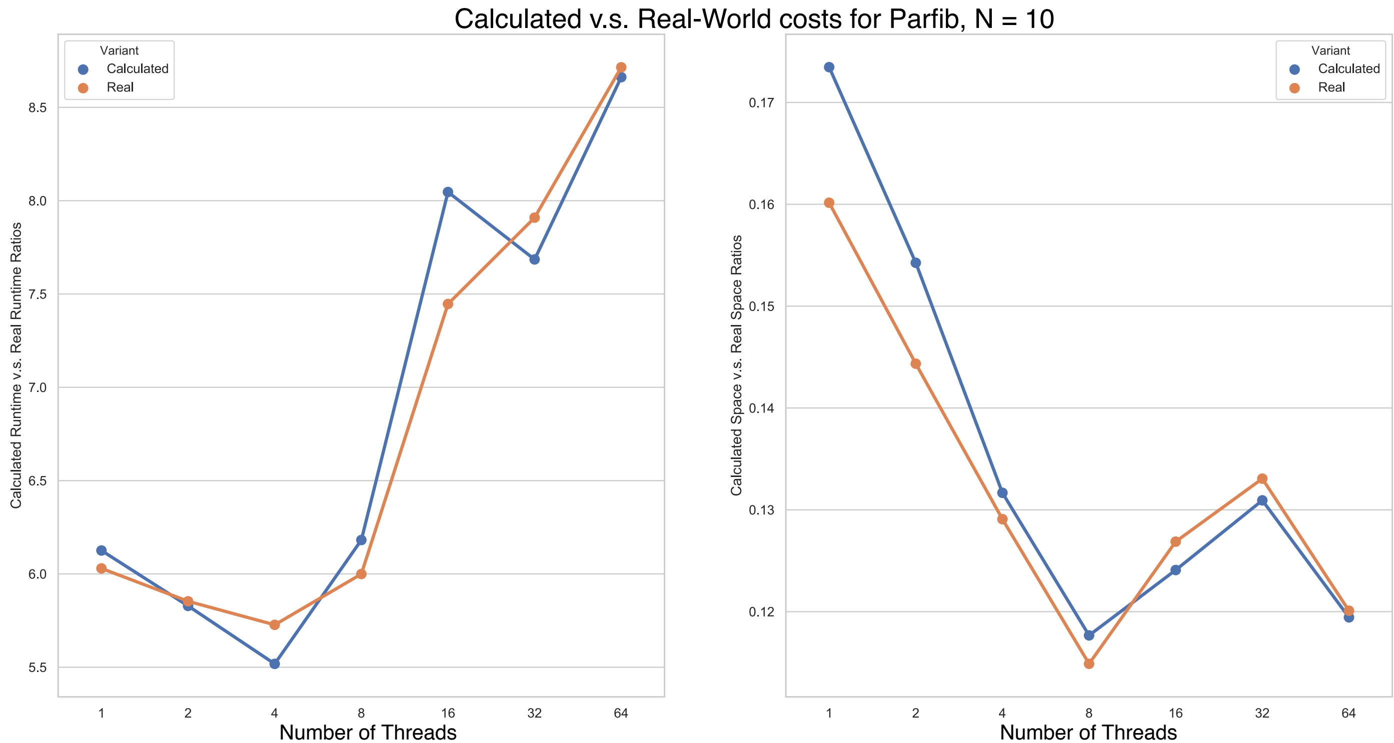


Figure 6.5: The ratios between using only one region and many (one for each resursive call) for the parallel Fibonacci function with $N = 10$. The blue line ("Real") are the observed ratios (both runtime and space) from the execution of the compiled function at the given thread counts. The other orange line ("Calculated") are the ratios between the calculated space usage and computational times for this function at the given thread counts.

Figure 6.6 likewise presents the plots of the ratios for the different versions of parallel vector addition. As we can see from the plot, while the overall ratios remain fairly accurate, the predicted runtimes around warp-divergence for the GPU do not fully match that of the GPU—as witnessed by the fact that our accuracy is off by 29% in particular since we are not able to factor in the slowdown due to warp-divergence enough and also do not factor in the unpredictability of computational time that arises from this as well. However, while we are not able to represent the difference in runtimes between warp-divergent code with complete accuracy, we still see that the cost semantics are still able to reflect these GPU-related artifacts with some perspicacity into the calculated cost.

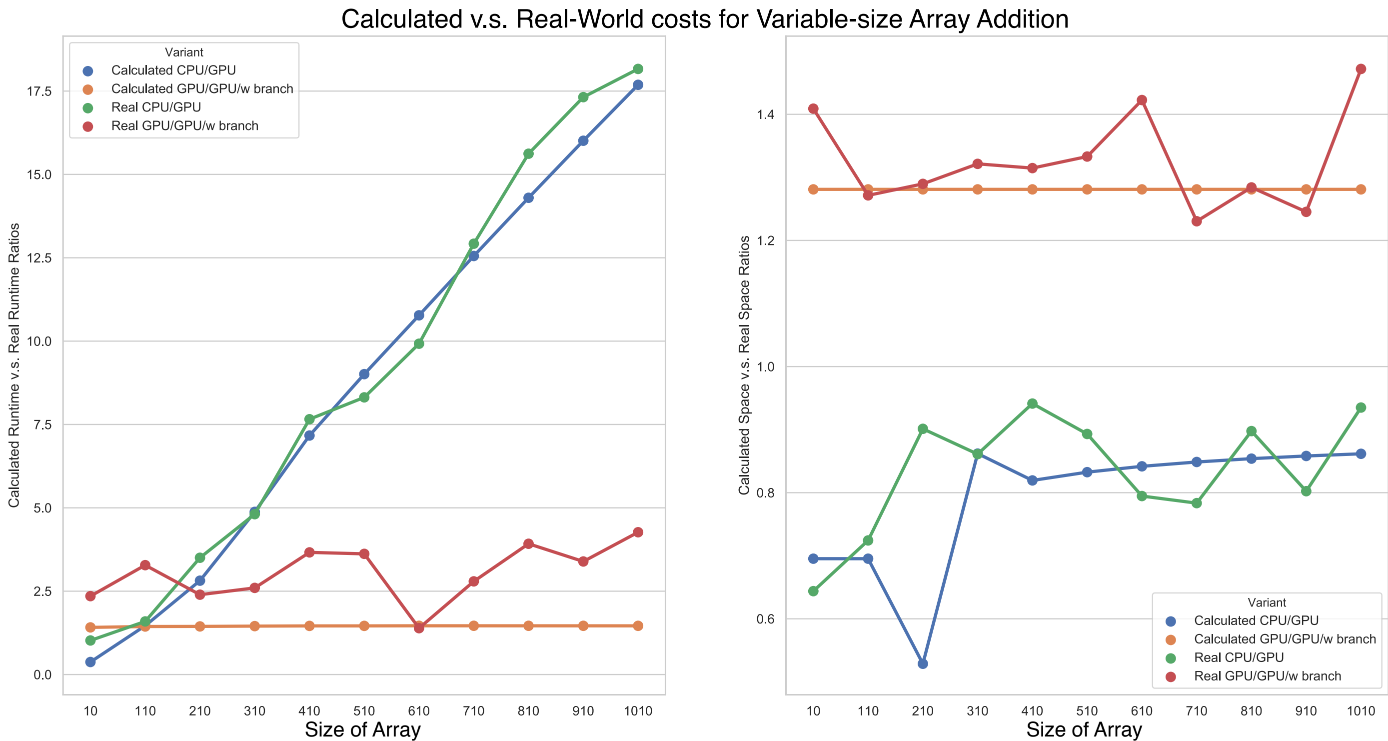


Figure 6.6: The ratios between vector addition on the CPU and GPU. The green and red lines are the observed ratios (both runtime and space) from the execution of the compiled functions for addition (CPU, GPU, and GPU with branching). The other orange and blue lines are the ratios between the calculated space usage and computational times for the compiled vector addition functions.

We can see from these results, that while the predicted costs are not accurate at the level of computational cycles, the ratios of the calculated costs as compared to the ratios of the observed costs are generally quite accurate; with an average difference in ratios of 7% and 3.5% for time and space respectively for the Fibonacci function variants, and 4.2% and 9.7% for time and space for CPU/GPU and 29.4% and 6.3% for GPU, and GPU addition with branching for the variable size array addition programs.

7
Conclusions and Future Work

Contents

7.1 **Limitations** 214
7.2 **Future Directions** 215

At the beginning of this dissertation we hypothesized that

> *Cost semantics are a viable and accurate model for analyzing and profiling high-level higher-order heterogeneous parallel functional languages. Further, such cost semantics allow the determination of relative (parallel) costs of programs running on either the CPU or GPU and the ability to inform computational placement of these programs.*

and in this dissertation we have built a cost semantics for a region-based heterogeneous parallel functional language (Chapters 3 to 5), and shown that these cost semantics are both implementable, and faithful to describing the abstract cost of computations in this setting (Chapter 6). In order to build these cost semantics, we have developed a number of new techniques in order to represent region-based

languages on the CPU and the weighted cost graphs that they entail, along with constructions for representing parallelism and computation on the GPU that takes into account nested parallelism, and shared work-group memory.

The work that we have presented can be seen as foundational in nature; a cost semantics serves as a base on which other work can be built and extended. Thus a number of future directions and applications are opened up by this work. Particularly, since many other areas require a cost semantics as a way to handle the concept of cost of the programs that they work on, having a cost semantics for heterogeneous parallelism allows researchers in these fields to take into account heterogeneous parallelism. We discuss some of these future directions, and how they tie in to this work in Section 7.2. However, while we have the expanded the applicability of cost semantics to new domains by the research that we have presented, there are still a number of limitations and things that we do *not* take into account. In the next section, we briefly summarize some of these limitations.

7.1 Limitations

Multiple CPUs and GPUs In the current setting the cost semantics for $\Lambda^{\mathsf{C}}_{\mathsf{G}}$ assumes that there is one host (CPU) and one device (GPU) in the system. One could extend the theory and implementation to take into account multiple CPU/GPU devices, however, the theory and implementation would need to be extended in order to take into account multiple computational worlds in both the type system and semantics. As well, there might be different transference rates between different host/device combinations that needs to be taken into account. Further, the theory for region merging and memory transfers relies on the fact that the kernel will always return to the host that sparked it, and therefore only the delta needs to be merged back. In a multi-device setting we may lose the key invariant, and the theory would need to be updated to handle this.

Memory Hierarchies on the CPU While we take into account the different hierarchies of memory on the GPU, we do not take this into account on the CPU portion of the language, and for good reason: while the GPU does not have cache eviction/loading policies based upon memory accesses, the various caches on the CPU do. Due to this policy, in the presence of parallelism determining which locations reside in cache becomes impossible to handle in a standard costing framework[1]: due to the fact that multiple threads of execution are non-deterministic in their execution, the loading and eviction of sections of memory from cache is also non-deterministic—and which sections of memory reside in cache.

Full GPU Memory Hierarchy Likewise, in order to simplify the theory, we do not take into account the full memory hierarchy on the GPU (constant/texture and thread local), and do not handle shifting of which regions reside in this local memory during the computation.[2] Nor do we handle loading of partial regions of memory into shared memory, and we only model the loading of entire regions.

7.2 Future Directions

There are a number of future directions to be explored from this work, and that have come up in the process of the research that has been presented. We provide a brief synopsis of each of these, and the possible ways that we think this work could be extended in these directions.

---

[1] See Section 7.2 for a possible way around this by using a novel non-standard costing policy.

[2] Note that this is different from the CPU cache case since we would be making the choice as to when the memory should be moved around as opposed to this being non-deterministically chosen for us by the threads.

Coupling Data Layout with Cost Semantics

Cost semantics are needed for amortized complexity analysis. However to the author's knowledge none of these systems for amortized complexity analysis take into account data layout in the underlying costs and hence cannot take this into account in their static—type-level—information. Extending the work that we have presented to use a layout (and RBMM)-based type system could allow data-layout aware amortized complexity analysis for heterogeneous parallel languages.

Moving to the distributed setting

The CPU/GPU can be seen as a special-case of a distributed system. Thus large portions of the work in this dissertation should be able to be extended to the distributed setting without too much work—in particular distributed memory parallelism. Developing a cost semantics for distributed memory parallelism in this manner could then allow asking and answering such questions as "is it more cost efficient to compute locally, or to place the computation in parallel on a different machine?" Furthermore, this could be tied in to start adding distributed cost types in which bounds of distributed parallel programs could be automatically verified at compilation (typechecking).

Cost Mobility Skeletons for CPU/GPU and Distributed programs

Autonomous Mobility Programs (AMPs) such as JoCaml [78] may migrate computation from one part of a distributed computation to another based upon a dynamic determination of where on the network/cluster is the best place to perform the computation. To extend this work to no longer require an explicit cost model, cost annotations, or mobility decision function Deng, Michaelson, and Trinder [79]

present Costed Autonomous Mobility Skeletons (CAMs) [79, 80] that can determine where to migrate computations based upon a automatic continuation cost analyzer. However in order to determine the cost of the continuations a cost semantics is required.

While we haven't created cost equations, or a continuation-based cost semantics in this dissertation, the cost semantics and the theory that we have developed should be readily extendable to this setting. Using such a cost semantics would allow CAMs systems to be able to be used in heterogeneous settings.

Paging Policies, Local Memory, and Distribution-based cost semantics

The I/O model can be used for more than just emulating a cache—it can be used for emulating hierarchical systems—in particular, the difference between main memory and on-disk memory. However, in the presence of parallelism there are issues that need to be handled; taking into account shared work-group memory on the GPU was only possible due to the fact that we didn't have to worry about cache eviction. However on the CPU and with the I/O model this is not nearly as straightforward—due to the way caching is performed on the CPU (eviction/pulling blocks into cache and/or paging based upon memory accesses), determining which memory locations are in (or not in) cache/memory becomes intractable in the presence of parallelism. However, taking inspiration from Das and Hoffmann [68] if we instead wanted to have infer *distributions* over costs as opposed to straightforward cost *constants* we could make the following observations:

- The profiler would be a probabilistic program itself, and the cost of the program would be the joint distribution over the cost distributions for the primitives in the language ($+, -$ etc. just as in Das and Hoffmann [68]).

- Since the profiler would be a probabilistic program itself, we could run the program "forwards" to determine the primitive distributions, and then use this to perform program inference. In fact this could be seen as type of abstract interpretation of the program—and the profiler can be implemented as an interpreter for the language (in a probabilistic programming language).

Adding Cost Constants

While the costs that we infer are *abstract* and result in the same complexity class as the original program, the costs that are inferred need not—and indeed are more likely than not—to be the same as the actual runtime of the program. It would be interesting to extend the abstract cost semantics that we have presented to also factor in cost constants, and to learn these cost constants and through this create a cost semantics that can more accurately predict the real-world costs of the programs. However, doing this would also mean that we would need to "de-abstract" the PCPs that we developed; we developed the ATPR to largely move the costs that we learned from the real-world into the abstract costs that we required so we would no longer require this, and could simply learn the PCP costs directly.

Appendices

Index of Notation

Chapter 2 Index of Notation

$E; \sigma; R; e \Downarrow v; \sigma'; c; s$ .... **Cost semantic judgement for NESL**, 15

$E; e \Downarrow v; c$ .... **Basic cost semantic judgement for λ-calculus**, 9

$S_r(Q_i, \sigma_i)$ .... **Reachable space at step i with state σ_i and queue Q_i**, 17

$\Gamma, e : \tau, \varphi$ .... **Effect typing judgement for STLC with regions** 43

$\mathsf{live}(R, v)$ .... **Live locations from roots R in v**, 26

$\mathsf{scan}(R, v)$ .... **Scan of nursery w.r.t R and v as in** live, 26

$\mathsf{space}_{v,h}(\trianglelefteq)$ .... **Space used to compute v under $\trianglelefteq$ and h**, 23

$\bowtie$ .... **Simultaneously scheduled operator**, 21

$\ell \equiv_{\mu'} \ell'$ .... **Location equivalence w.r.t. cache block μ'**, 26

$\ell_1 \preceq_h \ell_2$ .... **Path reachability between ℓ_1 and ℓ_2 in h**, 23

$\equiv_\mu$ .... **Neighbor relation over memory μ**, 26

$\mu \oplus \beta$ .... **Expansion of memory μ by block β**, 26

$\oplus$ .... **Sequential cost graph composition**, 14

$\otimes$ .... **Parallel cost graph composition**, 14

$\prec_v$ ..... **Allocation ordering on allocation cache** v, 26

$\rho \ominus \beta$ ..... **Contraction of cache** ρ **by block** β, 26

$\mathsf{roots}_{h,\ell}(N)$ ..... **Roots in** h **w.r.t.** ℓ **after scheduling nodes** N, 22

$\lhd$ ..... **Strict schedule order given by "is parent"**, 20

$\unlhd$ ..... **Schedule order**, 20

$\widehat{N_i}$ ..... **Closure of the step** N_i, 21

Chapter 3 Index of Notation

R ..... **Region**, 56

$R; v \downarrow o@R'$ ..... v **stored at offset** o **in** R **results in updated region** R', 78

Δ ..... **Region constraint context**, 56

$\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$ ..... e **has type** τ **and effects bounded by** φ, 67

$\mathcal{E} \overset{\cong}{\leftarrow} \mathcal{S}$ ..... $\mathcal{E}$ **is consistent w.r.t. region stack** $\mathcal{S}$, 74

$\mathcal{E}$ ..... **Runtime region context**, 73

$\mathcal{E}; \mathcal{S} \vdash_{\text{vreg}} \Omega$ ..... **Well-formedness of** Ω **w.r.t.** $\mathcal{E}$ **and** $\mathcal{S}$, 74

$\mathcal{E}; \mathcal{S}; \Omega \vdash_{\text{val}} v : \tau$ ..... **Value** v **has type** τ **w.r.t.** $\mathcal{E}$**,** $\mathcal{S}$**, and** Ω, 74

Γ ..... **Typing context**, 56

$\Omega \vdash_{\text{place}} \rho$ ..... **Region** ρ **is a valid region variable in** Ω, 67

$\Omega \vdash_{\text{rctxt}} \Delta$ ..... **Region constraint context** Δ **is well-formed w.r.t.** Ω, 67

Ω ..... **Region context**, 56

$\Omega; \Delta \vdash_{\text{vctxt}} \Gamma$ ..... **Typing context** Γ **well-formed w.r.t.** Δ**,** Ω, 67

$\Omega; \Delta \vdash_{\text{btype}} \mu$ ..... **Boxed type** μ **is well-formed w.r.t.** Ω**,** Δ, 67

$\Omega; \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'$ ..... φ' **is a subset of** φ **w.r.t.** Ω**,** Δ, 67

$\Omega; \Delta \vdash_{\text{eff}} \varphi$ ..... **Effect** φ **is well-formed w.r.t.** Ω **and** Δ, 67

$\Omega; \Delta \vdash_{\text{er}} \varphi \ni \rho$ ..... **The region** ρ **is a region in effect** φ **w.r.t.** Ω**,** Δ, 67

$\Omega; \Delta \vdash_{\text{re}} \rho \geq \varphi$ ..... **All regions in** φ **outlive** ρ **w.r.t.** Ω**,** Δ, 67

$\Omega; \Delta \vdash_{\text{rr}} \rho \geq \rho'$ ..... **Region** ρ' **outlives** ρ **w.r.t.** Δ**,** Ω, 67

$\Omega; \Delta \vdash_{\mathsf{type}} \tau$ .... **Type τ is well-formed w.r.t. Ω, Δ**, 67

$\mathcal{S}$ .... **Region stack**, 56

$\mathcal{S}; \ell \uparrow v$ .... **Reading location ℓ in region stack $\mathcal{S}$ results in value v**, 78

ℓ .... **Location**, 56

$\vdash_{\mathsf{ctxt}} \Omega; \Delta; \Gamma; \varphi$ .... **Well-formedness of effect φ w.r.t Ω, Δ, and Γ**, 67

$\vdash_{\mathsf{reg}} \Omega$ .... **Region context well-formedness**, 67

μ .... **Boxed type**, 56

$\mathsf{steps}(\trianglelefteq_{\mathsf{w}})$ .... **Set of simultaneously scheduled nodes w.r.t. $\trianglelefteq_{\mathsf{w}}$**, 94

τ .... **Boxed or unboxed type**, 56

φ .... **Effect set**, 56

$\trianglelefteq_{\mathsf{w}}$ .... **Weighted schedule order**, 93

$regions(\Delta)$ .... **Set of all regions in Δ**, 67

$\mathsf{allocSize}$ .... **Default allocation size**, 77

Λ_{C} .... **CPU portion of language**, 55

Chapter 4 Index of Notation

$A[j]$ .... **The j'th address group** , 108

$B[j]$ .... **Index j of (GPU) memory bank B**, 106

$S^{\mathcal{P}}(\overline{\alpha})$ .... **Schedulable nodes in $\overline{\alpha}$**, 126

$S_i^{\mathcal{P}}$ .... **Shorthand for scheduleable nodes over $\overline{\alpha}_i$**, 127

$T(0), \ldots, T(p-1)$ .... **Set of all p GPU threads**, 107

$W(0), \ldots, W(p/w-1)$ .... **The set of warps $W(i)$ for the GPU**, 107

$W(i)$ .... **The i'th warp**, 107

$[S_i]_{\dot{\alpha}}$ .... **Simultaneously scheduled nodes w.r.t. S_i**, 128

$A_w(i)$ .... **i'th (GPU) address group of width w**, 111

$\mathcal{A}_R(o)$ .... **Translation of region offset o to global address**, 135

$[-]_B$ .... **Bank equivalence function**, 136

$B_w(i)$ .... **i'th (GPU) memory bank of width w**, 111
$\mathbb{B}_{cast}(\Sigma, L)$ .... **Broadcast of Σ w.r.t L**, 138
$\mathbb{B}^n_{conf}(\Sigma, L)$ .... **n-bank conflict in Σ w.r.t. L**, 137
$\mathbb{L}$ .... **Kernel identifier to local memory mapping**, 140
$DMM^l_w(p)$ .... **DMM with width w, threads p, and latency l**, 111
$\mathcal{D}$ .... **Device box**, 115
$\dot{\mathcal{S}}$ .... **Final region stack after cost graph creation**, 135
$HMM^l_w(p,d)$. **HMM with w, p, l as in the UMM, number of DMMs d**, 111
K .... **Kernel pool**, 119
Σ .... **Warp trace**, 117
$\Sigma(S^{\mathcal{P}}_i)$ .... **Step trace: trace at step $S^{\mathcal{P}}_i$**, 136
$UMM^l_w(p)$ .... **UMM with width w, threads p, and latency l**, 111
$\mathsf{W}^i_k \rightrightarrows \mathsf{W}^{i+1}_k; S^{\mathcal{P}}_i$ .... **Warp step relation**, 131
W^i_k .... **k'th warp at step i**, 131
W .... **Warp representation in GPU cost graph**, 114
$\mathcal{W}$ .... **Warp pool**, 114
$\mathcal{W}(k)$ .... **Warp pool with k warps**, 115
$\blacksquare\ell$ .... **Meta location**, 114
$\mathsf{b}; \mathcal{W}(n)^i \Rrightarrow \left(\mathcal{W}(n)^{i+1}; S^{\mathcal{P}}_i; \mathsf{b}'\right)$ .... **Warp pool step relation**, 132
$\boxed{\mathcal{D}}$ .... **Quiesced device box**, 126
$\boxtimes$ .... **Bulk parallel operator**, 115
$\mathsf{cost}_{\Lambda^\mathsf{G}}\left(S^{\mathcal{P}}_i, L\right)$ .... **Cost for nodes in $S^{\mathcal{P}}_i$ w.r.t. local memory L**, 140
$R; v \downarrow o@R'$ .... **GPU allocation judgement**, 118
$\mathcal{S}; v \uparrow v'; \overline{\ell}$ .... **GPU lookup judgement**, 118
$\mathsf{cost}_{\Lambda^\mathsf{G}}\left(\overline{S^{\mathcal{P}}}, \mathbb{L}, \overline{\kappa}, \overline{\kappa'}\right)$ .... **Cost of step $S^{\mathcal{P}}$ w.r.t. $\mathbb{L}$ in device box**, 140
α .... **Cost graph node**, 117
$\mathsf{localaddr}^a(L)$ .... **Predicate specifying if a is in local memory L**, 137
$\mathsf{local}^a_v(L)$ .... **Predicate determining if v resides in local memory L**, 137

κ . . . . **Kernel identifier**, 114

$\mathsf{cost}_{\mathsf{G}}(\mathcal{D},\mathbb{L})$ . . . . **Total computational cost for $\mathcal{D}$ w.r.t. $\mathbb{L}$**, 140

$\mathsf{load}^{n}_{\mathcal{S}}(\overline{r})$ . . . . . . **Load names $\overline{r} \in \mathsf{dom}(\mathcal{S})$ into local memory of size n** , 138

$\mathsf{localaddrs}(L,\Sigma)$ . . . . **Set of local addresses in Σ w.r.t. L**, 137

$\mathsf{localcost}_{\mathcal{S}}(\Sigma,L)$ . . . . **Cost w.r.t. L, $\mathcal{S}$, and trace Σ**, 139

$\overline{\kappa}; \mathcal{D} \twoheadrightarrow \mathcal{D}'; \overline{S^{\mathcal{P}}_{i}}; \overline{\kappa'}$ . . . . **Device box stepping relation**, 133

$\overline{\ell}$ . . . . **Possible location/memory access**, 118

$\prec_{c}$ . . . . **Parent relationship in thread graph c**, 127

ψ . . . . **Kernel result pool**, 120

σ . . . . **Local memory state**, 114

$\mathsf{K}; \mathcal{S} \left|_{E}\, e \Downarrow^{\mathsf{t}} v; c \right| \mathcal{S}'; \mathsf{K}'$ . . . . **GPU thread graph judgement form**, 119

$\{c_1, \ldots, c_w\}$ . . . . **Set of thread-graphs in the GPU cost graph**, 113

p/w . . . . **Partition of total GPU threads into warps**, 107

Λ^{G} . . . . **GPU portion of language**, 101

$\bullet_{\ell}$ . . . . **Non-location**, 118

Chapter 5 Index of Notation

$R_1; R_1' \rhd R_2; \delta$ . . . . **Merge of regions R_1, R_1' with cost δ**, 153

$\Omega; \Delta; \Gamma^{\omega} \vdash_{\mathsf{exp}} e : \tau, \varphi$ . . . . **Same as in Chap. 3, but with worlds**, 147

$\mathcal{S}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}})$ . . . . **Space usage at node $\tilde{\alpha}$ under $\trianglelefteq_{\mathsf{w}}$**, 165

$\mathsf{cost}_{\mathsf{merge}}(R_1, R_1')$ . . . **PCP determining cost of merging regions R_1, R_2**, 153

$\mathsf{pull}_{\mathcal{S}}\left(\overline{r_i \to R_i}\right)$ . . . . . **Pull regions $\overline{R_i}$ to CPU and merge with parents**, 155

$\mathsf{push}\left(\overline{r_i \to R_i}\right)$ . . . . **Push regions $\overline{R_i}$ to the GPU**, 155

$\widetilde{\mathsf{frv}}(E, e)$ . . . . . . **Ranked free region variables of e in environment E**, 160

$\mathsf{space}^{\mathsf{C}}_{\mathsf{G}}(\trianglelefteq_{\mathsf{w}})$ . . . . **Heterogeneous space usage w.r.t. $\trianglelefteq_{\mathsf{w}}$**, 165

$\mathsf{space}^{\mathsf{C}}(\trianglelefteq_{\mathsf{w}})$ . . . . **CPU space usage w.r.t. schedule $\trianglelefteq_{\mathsf{w}}$**, 164

$\mathsf{space}\mathcal{D}(\tilde{\alpha}, \trianglelefteq_{\mathsf{w}})$ . . . **GPU space usage w.r.t. kernel launched at $\tilde{\alpha}$, $\trianglelefteq_{\mathsf{w}}$**, 165

step_w .... **Max weighted CPU edge in step**, 162
$\mathsf{step}^{\mathsf{H}}_w$ .... **Max weighted edge in step (heterogeneous)**, 162
$\tilde{\alpha}$ .... **Heterogeneous cost graph node**, 157
$\mathsf{time}^{\mathsf{H}}(e, \trianglelefteq_{\mathsf{w}}, \mathsf{determineLocal}^n(_,_,_))$ .... **Heterogeneous time cost for** e **w.r.t.** $\trianglelefteq_{\mathsf{w}}$ **and** $\mathsf{determineLocal}^n(_,_,_)$, 163
$\mathsf{determineLocal}^n(e, \mathcal{S}, E)$ .... **Region ranking function for loading into local GPU memory**, 160
$\mathsf{traceMerge}^{R'}_R(\ell)$ .... **New location of** $\ell \in R'$ **in the merge of** R**,** R', 154
$\lambda^{\mathsf{C}}_{\mathsf{tfr}}(\overline{R})$ .... **PCP for transferring regions** $\overline{R}$ **to the CPU**, 155
$\lambda^{\mathsf{G}}_{\mathsf{tfr}}(\overline{R})$ .... **PCP for transferring regions to the GPU**, 155
C .... **CPU computational world**, 144
G .... **GPU computational world**, 144
$\Lambda^{\mathsf{C}}_{\mathsf{G}}$ .... **Combined CPU/GPU language**, 143

Chapter 6 Index of Notation

$R_1 \boxplus R_2$ .... **Neighbor joining of** R_1 **and** R_2, 202
Δ^a .... **Allocation cost PCP**, 200
Δ^s .... **Reallocation cost PCP**, 201
$\mathsf{coalesce}\left(\overline{R}\right)$ .... **Join of all neighboring regions in** $\overline{R}$, 202
$[\![e, ret]\!]_{\mathsf{S}}$ .... **Statement hoisting transformation of** e, 177
$\mathsf{isNbr}\,(R, R')$ .... **Predicate determining if** R **and** R' **are contiguous**, 202
$\mathsf{cost}_{\mathsf{merge}}(R, R')$ .... **PCP for merging regions** R **and** R', 203
$\mathsf{calcExpand}\,(R, v)$ .... **Expansion of** R **w.r.t.** v **and** allocSize, 203
$\mathsf{transfer}^{\mathsf{C}}\left(\overline{R}\right)$ .... **Transfer of** $\overline{R}$ **from CPU to GPU**, 202
$\mathsf{transfer}^{\mathsf{G}}\left(\overline{R}\right)$ .... **Transfer of** $\overline{R}$ **from GPU to CPU**, 202
$\mathsf{kernLaunch}()$ .... **Empty kernel launch operation**, 204
κ_{sched} .... **Kernel scheduling overhead**, 204

B

Proofs of Theorems

Contents

B.1 Proof of Theorem 3.6.3 226
B.2 Proof of Theorem 3.4.1 227
B.3 Proof of Theorem B.3.1 228
B.4 Proof of Theorem 4.6.1 230
B.5 Proof of Theorem 5.2.1 231
B.6 Proof of Theorem 5.3.2 231

This appendix details the proofs for the theorems that we have in the rest of the dissertation.

B.1 Proof of Theorem 3.6.3

Theorem B.1.1 (Boundedness of $\Downarrow$)**.** *Let* $E; \mathcal{S}; e \Downarrow \ell; \mathcal{S}'; c$. *Then* $\mathcal{S} \leadsto \mathcal{S}'$.

Proof. The proof follows immediately from the following observations about region stacks, and the semantic rules–specifically the `letregion` rule:

- The memory within regions is immutable; and

- the only introduction form for regions is the `letregion` form.

With these two observations the proof follows almost immediately: by inspection of the `letregion` rule in Figure 3.18, we see that the region r introduced to the region stack $\mathcal{S}$ when evaluating e is removed from the region stack $\mathcal{S}'$ in the resulting region stack $\mathcal{S}'$ from the execution. Since no other semantic rules can introduce or remove regions from the region stack, we have that $\mathsf{dom}(\mathcal{S}) \subseteq \mathsf{dom}(\mathcal{S}')$. The second part, that $|\mathcal{S}'(r)| \geq |\mathcal{S}(r)|$ for all $r \in \mathsf{dom}(\mathcal{S})$ follows immediately from immutability of regions. □

B.2 Proof of Theorem 3.4.1

Theorem B.2.1. *$(\mathcal{S}, :\geq, \underline{R}, \overline{R})$ forms a bounded lattice with upper bound $\underline{R}$ and lower bound $\overline{R}$. Where $\underline{R}$ is the starting region, and $\overline{R}$ is the region at the top of the stack $\mathcal{S}$.*

Proof.

Reflexivity This follows immediately from the definition of $:\geq$.

Transitivity If $S \vdash R_1 :\geq R_2$ and $S \vdash R_2 :\geq R_3$, then by the definition of $:\geq$ we have that $\mathcal{S}$ must have the following structure:

$$\mathcal{S} = \mathcal{S}_1, r_1 \mapsto R_1, \mathcal{S}_2, r_2 \mapsto R_2, \mathcal{S}_3, r_3 \mapsto R_3, \mathcal{S}_4$$

from which we have that $S \vdash R_1 :\geq R_3$.

Antisymmetry If $S \vdash R_1 :\geq R_2$ then $\mathcal{S}$ must have the following structure:

$$\mathcal{S} = \mathcal{S}_1, r_1 \mapsto R_1, \mathcal{S}_2, r_2 \mapsto R_2, \mathcal{S}_3$$

Likewise, if and $S \vdash R_2 :\geq R_1$, then $\mathcal{S}$ must have the following structure:

$$\mathcal{S} = \mathcal{S}_1, r_2 \mapsto R_2, \mathcal{S}_2, r_1 \mapsto R_1, \mathcal{S}_3$$

But the only way that this region stack structure can happen is if R_1 is the same region as R_2.

Meet Let R_1 and R_2 be regions in $\mathcal{S}$. Then we have by the definition of $:\geq$ with respect to the region stack $\mathcal{S}$ that either $\mathcal{S} \vdash R_1 :\geq R_2$ or $\mathcal{S} \vdash R_2 :\geq R_1$. Without loss of generality, we will assume that $\mathcal{S} \vdash R_1 :\geq R_2$. Now take R_2 as the meet (thus a lower bound exists). To see that this is the greatest such lower bound, assume for contradiction that there existed another region R' such that $\mathcal{S} \vdash R_1 :\geq R'$, $\mathcal{S} \vdash R_2 :\geq R'$, and $\mathcal{S} \vdash R' :\geq R_2$. Then we have by the antisymmetry of $:\geq$ that $R' \equiv R_2$. Thus R_2 is the meet of the two.

Join Similar reasoning to the meet case, but taking the greater instead of lesser region.

Boundedness To see that this lattice is bounded, take $\bot = \underline{R}$ and $\top$ equal to the first element in $\mathcal{S}$–call this $\overline{R}$. Then we have by the definition of the rules for adding elements to $\mathcal{S}$, and by the definition of $:\geq$ that $\forall R \in \mathsf{range}(\mathcal{S})\ .\ \mathcal{S} \vdash \underline{R} :\geq R$, and likewise that $\forall R \in \mathsf{range}(\mathcal{S})\ .\ \mathcal{S} \vdash R :\geq \overline{R}$.

□

B.3 Proof of Theorem B.3.1

Theorem B.3.1 (Invariance Under Bounded Evolutions). *Let* $E; \mathcal{S}; e \Downarrow v_1; \mathcal{S}'; c$*, and such that* $trace(\mathcal{S}', v_1) = v$*. Let* $\mathcal{S} \rightsquigarrow \mathcal{S}_1$*. Then* $E; \mathcal{S}_1; e \Downarrow v_2; \mathcal{S}_1'; c'$*, and* $trace(\mathcal{S}_1', v_2) = v$*. Where* $trace$ *is defined as follows:*

$$\begin{array}{lcl}
trace(\mathcal{S}, i) & = & i \\
trace(\mathcal{S}, b) & = & b \\
trace(\mathcal{S},\ \mathtt{clos}\,(x:\tau, e, E)) & = & \mathtt{clos}\,(x:\tau, e, E) \\
trace(\mathcal{S},\ \mathtt{rclos}\,(\rho, \varphi, \varphi', u)) & = & \mathtt{rclos}\,(\rho, \varphi, \varphi', u) \\
trace(\mathcal{S}, (v_1, v_2)) & = & (trace(\mathcal{S}, v_1), trace(\mathcal{S}, v_2)) \\
trace(\mathcal{S}, [v_1, \ldots, v_n]) & = & [trace(\mathcal{S}, v_1), \ldots, trace(\mathcal{S}, v_n)] \\
trace(\mathcal{S}, \ell) & = & trace(\mathcal{S}, \mathcal{S}(\ell))
\end{array}$$

Proof. We proceed by induction on the structure of the input expression, and inspection of the operational semantics. Assume that $E; \mathcal{S}; e \Downarrow \ell; \mathcal{S}'; c$ and that $E; \mathcal{S}_1; e \Downarrow \ell'; \mathcal{S}_1'; c'$. We proceed by case analysis on the expressions e that can result in a location:

$\underline{e = \lambda x : \tau .^{\varphi} e'}$ By definition of the reduction rule for closures, we then have that $\mathcal{S}'(\ell) = \texttt{clos}\ (x : \tau, e', E)$, and that $\mathcal{S}_1'(\ell') = \texttt{clos}\ (x : \tau, e', E)$. Therefore we have that

$$\begin{aligned} trace(\mathcal{S}', \ell) &= trace(\mathcal{S}', \mathcal{S}'(\ell)) \\ &= trace(\mathcal{S}', \texttt{clos}\ (x : \tau, e', E)) \\ &= \texttt{clos}\ (x : \tau, e', E) \\ &= trace(\mathcal{S}_1', \texttt{clos}\ (x : \tau, e', E)) \\ &= trace(\mathcal{S}_1', \mathcal{S}_1'(\ell')) \\ &= trace(\mathcal{S}_1', \ell) \end{aligned}$$

$\underline{e = (e_1, e_2)\ \texttt{at}\ \rho}$ By the definition of the reduction rule for pairs, we have that

1. $E; \mathcal{S}; e_1 \Downarrow v_1; \mathcal{S}''; c_1, E; \mathcal{S}_1; e_1 \Downarrow v_1'; \mathcal{S}_1''; c_1'$ and inductively that $trace(\mathcal{S}'', v_1) = trace(\mathcal{S}_1'', v_1')$. We likewise have the same for e_2;

2. $E; \mathcal{S}''; e_2 \Downarrow v_2; \mathcal{S}'''; c_2, E; \mathcal{S}_1''; e_1 \Downarrow v_2'; \mathcal{S}_1'''; c_2'$ and inductively that $trace(\mathcal{S}''', v_2) = trace(\mathcal{S}_1''', v_2')$.

Proceeding by inspection of the pair rule we then have that

$$\mathcal{S}' = \mathcal{S}'''(E(\rho))[\ell \mapsto (v_1, v_2] \tag{B.1}$$

and that

$$\mathcal{S}_1' = \mathcal{S}_1'''(E(\rho))[\ell' \mapsto (v_1', v_2'] \tag{B.2}$$

Since the $\ell \notin v_1, \ell \notin v_2$, and $\ell' \notin v_1', \ell' \notin v_2'$ it is easy to see that we can replace $\mathcal{S}''$ and $\mathcal{S}'''$ in 1. by $\mathcal{S}'$, and $\mathcal{S}_1'', \mathcal{S}_1'''$ by $\mathcal{S}_1'$ as well. With this we then have that

$$\begin{aligned} trace(\mathcal{S}''', v_2) = trace(\mathcal{S}', v_2) = trace(\mathcal{S}_1''', v_2') = trace(\mathcal{S}_1', v_2') \\ trace(\mathcal{S}'', v_1) = trace(\mathcal{S}', v_1) = trace(\mathcal{S}_1'', v_1') = trace(\mathcal{S}_1', v_1') \end{aligned} \tag{B.3}$$

We therefore have that

$$
\begin{array}{lcr}
trace(\mathcal{S}', \ell) & = & \text{by def. of } trace \\
trace(\mathcal{S}', \mathcal{S}'(\ell)) & = & \text{by Equation (B.1) and def. of } trace \\
trace(\mathcal{S}', (v_1, v_2)) & = & \text{by Equation (B.3) and def. of } trace \\
trace(\mathcal{S}'_1, (v'_1, v'_2)) & = & \text{by Equation (B.2)} \\
trace(\mathcal{S}'_1, \mathcal{S}'_1(\ell')) & = & \text{by def. of } trace \\
trace(\mathcal{S}'_1, \ell') & = &
\end{array}
$$

The reasoning is similar for the other cases, so we omit them here. □

B.4 Proof of Theorem 4.6.1

Theorem B.4.1. *Let* $\mathsf{W} = \{c_1, \ldots, c_n\}$ *be a warp graph. Let* $\overline{\alpha}$ *be a set of considerable nodes, and* $S^{\mathcal{P}}(\overline{\alpha})$ *be its set of schedulable nodes for* W*. Then*

$$\forall c_j \in \mathsf{W}\ .\ \left|\{\alpha \mid \alpha \in \overline{\alpha}, \alpha \in c_j\}\right| \leq 1 \tag{B.4}$$

and therefore by the definition of $S^{\mathcal{P}}(\overline{\alpha})$ *that*

$$\forall c_j \in \mathsf{W}\ .\ \left|\{\alpha \mid \alpha \in S^{\mathcal{P}}(\overline{\alpha}), \alpha \in c_j\}\right| \leq 1 \tag{B.5}$$

Proof. Let $c_j \in \mathsf{W}$ be a thread graph. We want to show that at most $1\ \alpha \in \overline{\alpha}$ can be in c_j. Assume for contradiction that there exist two $\alpha_i \in \overline{\alpha}$ such that $\alpha_1, \alpha_2 \in c_j$. By the definition of the thread graph combinators in Figure 4.6, we have that either α_1 is a parent of α_2 or that α_2 is a parent of α_1. Without loss of generality, assume that α_2 is a parent of α_1. However, by the definition of considerable nodes, any two nodes in $\overline{\alpha}$ cannot be a parent of any of the others, but since α_2 is a parent of α_1 this is a contradiction.

The second statement can be seen easily by noting that $S^{\mathcal{P}}(\overline{\alpha}) \subseteq \overline{\alpha}$ and therefore the same argument holds over this set as it does over $\overline{\alpha}$. □

B.5 Proof of Theorem 5.2.1

Theorem B.5.1 (Syntactic Inclusion). *Let e be a closed, well-formed expression in $\Lambda^{\mathsf{C}}_{\mathsf{G}}$. Then the following statements hold:*

- $\Omega; \Delta; \Gamma^{\mathsf{G}} \vdash_{exp} e : \tau, \varphi$ *then* $e \in \Lambda^{\mathsf{G}}$*;*
- $\Omega; \Delta; \Gamma^{\mathsf{C}} \vdash_{exp} e : \tau, \varphi$ *then* $e \in \Lambda_{\mathsf{C}}$*;*
- $e \in \Lambda^{\mathsf{G}} \implies e \in \Lambda_{\mathsf{C}}$ *(up-to 're-coloring' of regions variables).*

Proof. The first two statements are a straightforward consequence of the typing rules—in particular that our typing rules for Λ_{C} and Λ^{G} only allow syntactically valid programs. Otherwise these programs would fail to typecheck.

To see that $\Lambda^{\mathsf{G}} \subseteq \Lambda_{\mathsf{C}}$ is a straightforward consequence of the typing rules for Λ_{C} and Λ^{G}—inspecting the typing rules for the two languages shows that any expression e that would typecheck under Λ^{G} would also typecheck under Λ_{C}. To see that the inclusion does not go the other way simply take $e = \texttt{letregion}\ e'\ in\ \rho$; there is no typing for this expression under the typing rules for Λ^{G}, but one does exist for for this expression in Λ_{C}. □

B.6 Proof of Theorem 5.3.2

Theorem B.6.1 (Merge Conflict Freedom). *Let R_p be the parent region for R_1(on C) and R'_1(on G), and $R_1; R'_1 \rhd R; \delta$. Then* $\mathsf{range}(R_1) \subseteq \mathsf{range}(R) \wedge \mathsf{range}(R'_1) \subseteq \mathsf{range}(R)$ *and* $\forall o \in \mathsf{dom}(R_1) \cap \mathsf{dom}(R'_1). v_1(o) \equiv v_2(o)$.

Proof. $\mathsf{range}(R_1) \subseteq \mathsf{range}(R)$ follows from the definition of the merge operation in Definition 5.3.1, the definition of allocation into regions in Definition 3.4.1, and the definition of image of a relation. More precisely, recall that values can only been allocated up to the current allocation pointer for the region R_1—ap_1—and that the merge

operation does not affect addresses below ap_1 (as the merge operation performs a memcopy starting at ap_1). Therefore the memory in v_1 remains untouched after the merge operation, and hence in the resulting memory v of R. We therefore have that $\mathsf{dom}(v_1) \subseteq \mathsf{dom}(v)$ and therefore $\mathsf{range}(R_1) = \mathsf{range}(v_1) \subseteq \mathsf{range}(v) = \mathsf{range}(R)$ since the image of a subset of a relation is a subset of the image.

To see that $\mathsf{range}(R_1') \subseteq \mathsf{range}(R)$ we first use the definition of $\underset{G\to C}{\mathsf{memcpy}}$[1] to get that $\mathsf{range}(R_1' \setminus R_1) = \mathsf{range}(R_1') \setminus \mathsf{range}(R_1)$. We then have by the argument in the previous paragraph that $\mathsf{range}(R_1) \subseteq \mathsf{range}(R)$ and by the definition of the merge operation that $(\mathsf{range}(R_1') \setminus \mathsf{range}(R_1)) \subseteq \mathsf{range}(R)$. Now, we have that

$$(\mathsf{range}(R_1') \setminus \mathsf{range}(R_1)) \cup (\mathsf{range}(R_1') \cap \mathsf{range}(R_1)) = \mathsf{range}(R_1') \tag{B.6}$$

Since $\mathsf{range}(R_1') \cap \mathsf{range}(R_1) \subseteq \mathsf{range}(R_1)$, we then have by Equation (B.6) that $\mathsf{range}(R_1')$ is the union of two sets in $\mathsf{range}(R)$ and therefore that $\mathsf{range}(R_1') \subseteq \mathsf{range}(R)$.

We likewise have by the definition of the merge operation and by the immutability of regions that $\forall o \,.\, o \in \mathsf{dom}(R_1) \cap \mathsf{dom}(R_1') \implies o \in \mathsf{dom}(R_p)$. Therefore $v_p(o) = v$. We then have by immutability that this value cannot have changed or been deallocated in either of its evolutions R_1 and R_1' (since these are both bounded evolutions of the parent region R_p). Therefore $\forall o \in \mathsf{dom}(R_1) \cap \mathsf{dom}(R_1') . v_1(o) \equiv v_2(o)$. □

[1]In particular, that $\underset{G\to C}{\mathsf{memcpy}}$ is 1-1 and onto.

References

[1] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate Amortized Resource Analysis". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 357–370. url: http://doi.acm.org/10.1145/1926385.1926427.

[2] Pedro B Vasconcelos. "Space cost analysis using sized types". PhD thesis. University of St Andrews, 2008.

[3] Patrick M. Sansom and Simon L. Peyton Jones. "Time and Space Profiling for Non-strict, Higher-order Functional Languages". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 355–366. url: http://doi.acm.org/10.1145/199448.199531.

[4] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. "A Cost Semantics for Self-adjusting Computation". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 186–199. url: http://doi.acm.org/10.1145/1480881.1480907.

[5] Guy E. Blelloch and Robert Harper. "Cache and I/O Efficent Functional Algorithms". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: ACM, 2013, pp. 39–50. url: http://doi.acm.org/10.1145/2429069.2429077.

[6] Daniel Spoonhower et al. "Space profiling for parallel functional programs". In: *Journal of Functional Programming* 20 (Special Issue 5-6 Nov. 2010), pp. 417–461.

[7] Eric Holk et al. "Region-based Memory Management for GPU Programming Languages: Enabling Rich Data Structures on a Spartan Host". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 141–155. url: http://doi.acm.org/10.1145/2660193.2660244.

[8] Mads Tofte and Jean-Pierre Talpin. "Region-based memory management". In: *Information and computation* 132.2 (1997), pp. 109–176.

[9] Eric Holk. “Region-based Memory Management for Expressive GPU Programming”. PhD thesis. Indiana University, Bloomington: School of Informatics and Computer Science, Indiana University, June 2016. url: `http://blog.theincredibleholk.org/papers/dissertation.pdf`.

[10] Mozilla Research. *The Rust Programming Languge*. `https://www.rust-lang.org`. Accessed: 2017-01-10. 2010.

[11] Edward Z Yang et al. “Efficient communication and collection with compact normal forms”. In: *ACM SIGPLAN Notices* 50.9 (2015), pp. 362–374.

[12] Lu Lu et al. “Lifetime-Based Memory Management for Distributed Data Processing Systems”. In: *CoRR* abs/1602.01959 (2016). url: `http://arxiv.org/abs/1602.01959`.

[13] Patrick M. Sansom and Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages*. Tech. rep. University of Glasgow, 1994.

[14] *GHC Users Guide – Profiling*. 2016 (Last Accessed Sept. 2016). url: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html`.

[15] David Sands. “Complexity Analysis for a Lazy Higher-Order Language”. In: *In Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, 1990, pp. 361–376.

[16] Chris Okasaki. *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press, 1998.

[17] Guy Blelloch and John Greiner. “Parallelism in Sequential Functional Languages”. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA ’95. La Jolla, California, USA: ACM, 1995, pp. 226–237. url: `http://doi.acm.org/10.1145/224164.224210`.

[18] Peter J. Landin. “The Mechanical Evaluation of Expressions”. In: *Computer Journal* 6.4 (Jan. 1964), pp. 308–320.

[19] Stephen A. Cook and Robert A. Reckhow. “Time-bounded Random Access Machines”. In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC ’72. Denver, Colorado, USA: ACM, 1972, pp. 73–80. url: `http://doi.acm.org/10.1145/800152.804898`.

[20] Steven Fortune and James Wyllie. “Parallelism in Random Access Machines”. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC ’78. San Diego, California, USA: ACM, 1978, pp. 114–118. url: `http://doi.acm.org/10.1145/800133.804339`.

[21] Guy E. Blelloch and John Greiner. “A Provable Time and Space Efficient Implementation of NESL”. In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 213–225. url: `http://doi.acm.org/10.1145/232627.232650`.

[22] Guy E. Blelloch. *NESL: A Nested Data-Parallel Language*. Tech. rep. Pittsburgh, PA, USA, 1992.

[23] Alok Aggarwal and S. Vitter Jeffrey. “The Input/Output Complexity of Sorting and Related Problems”. In: *Commun. ACM* 31.9 (Sept. 1988), pp. 1116–1127. url: http://doi.acm.org/10.1145/48529.48535.

[24] Matteo Frigo et al. “Cache-Oblivious Algorithms”. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 285–. url: http://dl.acm.org/citation.cfm?id=795665.796479.

[25] Greg Morrisett, Matthias Felleisen, and Robert Harper. “Abstract Models of Memory Management”. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA ’95. La Jolla, California, USA: ACM, 1995, pp. 66–77. url: http://doi.acm.org/10.1145/224164.224182.

[26] Dana S. Scott. “A Type-theoretical Alternative to ISWIM, CUCH, OWHY”. In: *Theor. Comput. Sci.* 121.1-2 (Dec. 1993), pp. 411–440. url: http://dx.doi.org/10.1016/0304-3975(93)90095-B.

[27] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Adaptive Functional Programming”. In: *ACM Trans. Program. Lang. Syst.* 28.6 (Nov. 2006), pp. 990–1034. url: http://doi.acm.org/10.1145/1186632.1186634.

[28] Guy Blelloch, Phil Gibbons, and Yossi Matias. “Provably Efficient Scheduling for Languages with Fine-Grained Parallelism”. In: *Proceedings Symposium on Parallel Algorithms and Architectures*. July 1995, pp. 1–12.

[29] John Greiner and Guy E. Blelloch. “A Provably Time-efficient Parallel Implementation of Full Speculation”. In: *ACM Trans. Program. Lang. Syst.* 21.2 (Mar. 1999), pp. 240–285. url: http://doi.acm.org/10.1145/316686.316690.

[30] John Greiner. “Semantics-based parallel cost models and their use in provably efficient implementations”. PhD thesis. Carnegie Mellon University, 1997.

[31] Robert Jonathan Ennals. “Adaptive evaluation of non-strict programs”. PhD thesis. University of Cambridge, 2004.

[32] Kathryn Van Stone. “A Denotational Approach to Measuring Complexity in Functional Programs”. AAI3262826. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 2003.

[33] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. “Denotational cost semantics for functional languages with inductive types”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015, pp. 140–151.

[34] Martin Avanzini, Ugo Dal Lago, and Georg Moser. “Analysing the Complexity of Functional Programs: Higher-order Meets First-order”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 152–164. url: `http://doi.acm.org/10.1145/2784731.2784753`.

[35] Ralph Benzinger. “Automated Higher-order Complexity Analysis”. In: *Theor. Comput. Sci.* 318.1-2 (June 2004), pp. 79–103. url: `http://dx.doi.org/10.1016/j.tcs.2003.10.022`.

[36] John Hughes, Lars Pareto, and Amr Sabry. “Proving the Correctness of Reactive Systems Using Sized Types”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 410–423. url: `http://doi.acm.org/10.1145/237721.240882`.

[37] Wei-Ngan Chin and Siau-Cheng Khoo. “Calculating Sized Types”. In: *Higher Order Symbol. Comput.* 14.2-3 (Sept. 2001), pp. 261–300. url: `http://dx.doi.org/10.1023/A:1012996816178`.

[38] Hans-Wolfgang Loidl and Kevin Hammond. “A Sized Time System for a Parallel Functional Language”. In: *Proceedings of the Glasgow Workshop on Functional Programming*. Ullapool, Scotland, July 1996.

[39] Stéphane Gimenez and Georg Moser. “The Complexity of Interaction”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: ACM, 2016, pp. 243–255. url: `http://doi.acm.org/10.1145/2837614.2837646`.

[40] Yves Lafont. “Interaction Nets”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. San Francisco, California, USA: ACM, 1990, pp. 95–108. url: `http://doi.acm.org/10.1145/96709.96718`.

[41] Roberto M. Amadio et al. *Certifying cost annotations in compilers*. Tech. rep. Oct. 2010. url: `https://hal.archives-ouvertes.fr/hal-00524715`.

[42] Roberto M. Amadio and Yann Régis-Gianas. “Certifying and reasoning on cost annotations of functional programs”. In: *CoRR* abs/1110.2350 (2011). url: `http://arxiv.org/abs/1110.2350`.

[43] Leaf Petersen et al. “A Type Theory for Memory Allocation and Data Layout”. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’03. New Orleans, Louisiana, USA: ACM, 2003, pp. 172–184. url: `http://doi.acm.org/10.1145/604131.604147`.

[44] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’78)*. Tucson, Arizona: ACM, 1978, pp. 84–96.

[45] Doran K. Wilde. *A Library for Doing Polyhedral Operations*. Tech. rep. 785. IRISA, Dec. 1993.

[46] Artjoms Šinkarovs and Sven-Bodo Scholz. "Data Layout Inference for Code Vectorisation". In: *International Conference on High Performance Computing & Simulation (HPCS'13)*. Accepted for publication at HPCS'13. 2013. url: `/publications/data-layouts.pdf`.

[47] Artjoms Šinkarovs and Sven-Bodo Scholz. "Type-driven data layouts for improved vectorisation". In: *Concurrency and Computation: Practice and Experience* (2015). cpe.3501. url: `/publications/ccpe-data-layouts.pdf`.

[48] Artjoms Šinkarovs and Sven-Bodo Scholz. "Semantics-Preserving Data Layout Transformations for Improved Vectorisation". In: *2nd Workshop on Functional High-Performance Computing (FHPC'13)*. Accepted for publication at FHPC'13. 2013. url: `/publications/layout-correctness.pdf`.

[49] Fritz Henglein, Henning Makholm, and Henning Niss. "Effect types and region-based memory management". In: *Advanced Topics in Types and Programming Languages* (2005), pp. 87–135.

[50] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[51] Guillem Pratx and Lei Xing. "GPU computing in medical physics: A review". In: *Medical physics* 38.5 (2011), pp. 2685–2697.

[52] Samuel S Stone et al. "Accelerating advanced MRI reconstructions on GPUs". In: *Proceedings of the 5th conference on Computing frontiers*. ACM. 2008, pp. 261–272.

[53] I Schmerken. "Wall street accelerates options analysis with gpu technology". In: *Wall Street Technology* 11 (2009).

[54] Xulong Tang et al. "VC-Bench: A Video Coding Benchmark Suite for Evaluation of Processor Capability". In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Springer, 2013, pp. 101–116.

[55] Seung In Park et al. "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture". In: *Applied Imagery Pattern Recognition Workshop, 2008. AIPR'08. 37th IEEE*. IEEE. 2008, pp. 1–7.

[56] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (Mar. 2008), pp. 40–53. url: `http://doi.acm.org/10.1145/1365490.1365500`.

[57] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. url: `http://dx.doi.org/10.1109/MCSE.2010.69`.

[58] NVIDIA. *Dynamic Parallelism in CUDA*. 2012. url: `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf`.

[59] AMD. *AMD APP SDK OpenCL User Guid*. 2015. url: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.

[60] NVIDIA. *How to Optimize Data Transfers in CUDA C/C++*. 2012. url: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/.

[61] NVIDIA. *Unified Memory in CUDA 6*. 2013. url: https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[62] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. "Copperhead: compiling an embedded data parallel language". In: *ACM SIGPLAN Notices* 46.8 (2011), pp. 47–56.

[63] Manuel M T Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *DAMP'11: Declarative Aspects of Multicore Programming*. 2011, pp. 3–14.

[64] Trevor L. McDonell et al. "Optimising Purely Functional GPU Programs". In: *ICFP'13: International Conference on Functional Programming*. 2013, pp. 49–60.

[65] Matthew Fluet and Greg Morrisett. "Monadic Regions". In: *J. Funct. Program.* 16.4-5 (July 2006), pp. 485–545. url: http://dx.doi.org/10.1017/S095679680600596X.

[66] David Culler et al. "LogP: Towards a Realistic Model of Parallel Computation". In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '93. San Diego, California, USA: ACM, 1993, pp. 1–12. url: http://doi.acm.org/10.1145/155332.155333.

[67] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. url: http://doi.acm.org/10.1145/79173.79181.

[68] Ankush Das and Jan Hoffmann. "ML for ML: Learning Cost Semantics by Experiment". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. 2017, pp. 190–207. url: http://dx.doi.org/10.1007/978-3-662-54577-5_11.

[69] Michael Bauer et al. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11. url: http://dl.acm.org/citation.cfm?id=2388996.2389086.

[70] Koji Nakano. "The hierarchical memory machine model for GPUs". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 591–600.

[71] K. Nakano. "Simple Memory Machine Models for GPUs". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. May 2012, pp. 794–803.

[72] NVIDIA Coporation. *CUDA C Best Practices Guide*. 2017. url: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[73] Amr Sabry and Matthias Felleisen. “Reasoning About Programs in Continuation-passing Style.” In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP ’92. San Francisco, California, USA: ACM, 1992, pp. 288–298.

[74] Lars Birkedal et al. *The ML Kit: Version 1*. DIKU, 1993.

[75] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[76] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. “Typed closure conversion”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, pp. 271–283.

[77] Thomas Johnsson. “Lambda lifting: Transforming programs to recursive equations”. In: *Conference on Functional programming languages and computer architecture*. Springer. 1985, pp. 190–203.

[78] S. Conchon and F. Le Fessant. “Jocaml: mobile agents for Objective-Caml”. In: *Proceedings. First and Third International Symposium on Agent Systems Applications, and Mobile Agents*. Oct. 1999, pp. 22–29.

[79] Xiao Yan Deng, Greg Michaelson, and Phil Trinder. “Automatically costed autonomous mobility”. In: *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IEEE Computer Society. 2007, pp. 95–101.

[80] Xiao Yan Deng, Greg Michaelson, and Phil Trinder. “Cost-driven autonomous mobility”. In: *Computer Languages, Systems & Structures* 36.1 (2010), pp. 34–59.