INTROSPECTIVE PUSHDOWN ANALYSIS AND NEBO

by

Christopher Earl

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2014

Copyright © Christopher Earl 2014

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Christopher Earl**

has been approved by the following supervisory committee members:

Name	Role	Date Approved
Matthew Might	, Chair	**11/4/2013**
Martin Berzins	, Member	**11/1/2013**
Matthew Flatt	, Member	**11/1/2013**
Mary Hall	, Member	**11/1/2013**
James C. Sutherland	, Member	**11/4/2013**

and by **Alan Davis**, Chair/Dean of

the Department of **School of Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

In the static analysis of functional programs, control-flow analysis (k-CFA) is a classic method of approximating program behavior as a finite state automata. CFA2 and abstract garbage collection are two recent, yet orthogonal improvements, on k-CFA. CFA2 approximates program behavior as a pushdown system, using summarization for the stack. CFA2 can accurately approximate arbitrarily-deep recursive function calls, whereas k-CFA cannot. Abstract garbage collection removes unreachable values from the store/heap. If unreachable values are not removed from a static analysis, they can become reachable again, which pollutes the final analysis and makes it less precise. Unfortunately, as these two techniques were originally formulated, they are incompatible. CFA2's summarization technique for managing the stack obscures the stack such that abstract garbage collection is unable to examine the stack for reachable values.

This dissertation presents introspective pushdown control-flow analysis, which manages the stack explicitly through stack changes (pushes and pops). Because this analysis is able to examine the stack by how it has changed, abstract garbage collection is able to examine the stack for reachable values. Thus, introspective pushdown control-flow analysis merges successfully the benefits of CFA2 and abstract garbage collection to create a more precise static analysis.

Additionally, the high-performance computing community has viewed functional programming techniques and tools as lacking the efficiency necessary for their applications. Nebo is a declarative domain-specific language embedded in C++ for discretizing partial differential equations for transport phenomena. For efficient execution, Nebo exploits a version of expression templates, based on the C++ template system, which is a type-less, completely-pure, Turing-complete functional language with burdensome syntax. Nebo's declarative syntax supports functional tools, such as point-wise lifting of complex expressions and functional composition of stencil operators. Nebo's primary abstraction is mathematical assignment, which separates what a calculation does from how that calculation is executed. Currently Nebo supports single-core execution, multicore (thread-based) parallel execution, and GPU execution. With single-core execution, Nebo performs on par with the loops and code that it replaces in Wasatch, a pre-existing high-performance simulation

project. With multicore (thread-based) execution, Nebo can linearly scale (with roughly 90% efficiency) up to 6 processors, compared to its single-core execution. Moreover, Nebo's GPU execution can be up to 37x faster than its single-core execution. Finally, Wasatch (the pre-existing high-performance simulation project which uses Nebo) can scale up to 262K cores.

For Emily, with love.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **x**

LIST OF TABLES . . . . . **xi**

ACKNOWLEDGEMENTS . . . . . **xii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Skirting undecidability 1
1.2 Dissertation 1
1.3 Introspective pushdown analysis 2
1.3.1 Accessing the whole stack *versus* just the top 3
1.3.2 Overview 5
1.4 Nebo: A domain-specific language for numerically solving PDEs in high-performance simulations 6
1.4.1 Other domain-specific languages solving PDEs 7
1.4.2 Uintah 7
1.4.3 Wasatch 8
1.4.4 Use of Nebo in Wasatch 9
1.4.5 Overview 10
1.5 Contributions 10

2. PUSHDOWN CONTROL-FLOW ANALYSIS **12**

2.1 Introduction 12
2.2 Setting: A-Normal Form λ-calculus 12
2.2.1 Semantics 13
2.2.1.1 Transition relation 14
2.3 An infinite-state abstract interpretation 15
2.3.1 Program injection 15
2.3.2 Atomic expression evaluation 16
2.3.3 Reachable configurations 17
2.3.4 Transition relation 17
2.3.5 Allocation: Polyvariance and context-sensitivity 18
2.3.5.1 Monovariance: Pushdown 0CFA 18
2.3.5.2 Context-sensitive: Pushdown 1CFA 18
2.3.5.3 Polymorphic splitting: Pushdown poly/CFA 19
2.3.5.4 Pushdown k-CFA 19
2.3.6 Partial orders 19
2.3.7 Soundness 21
2.4 From the abstracted CESK machine to a PDA 21

2.4.1 Problem: Doubly exponential complexity . . . 23
2.5 Summary . . . 23

3. COMPUTABLE AND TRACTABLE PUSHDOWN CONTROL-FLOW ANALYSIS . . . 24

3.1 Introduction . . . 24
3.2 Dyck state graph . . . 25
3.3 Compacting a rooted pushdown system into a Dyck state graph . . . 27
3.3.1 Complexity: Polynomial and exponential . . . 28
3.4 An efficient algorithm: Work-lists and ϵ-closure graphs . . . 29
3.4.1 ϵ-closure graphs . . . 29
3.4.2 Integrating a work-list . . . 30
3.4.3 A new fixed-point iteration-space . . . 30
3.4.4 The ϵ-closure graph work-list algorithm . . . 30
3.4.4.1 Sprouting . . . 34
3.4.4.2 Considering the consequences of a new push edge . . . 35
3.4.4.3 Considering the consequences of a new pop edge . . . 36
3.4.4.4 Considering the consequences of a new ϵ-edge . . . 37
3.4.5 Termination and correctness . . . 39
3.4.6 Complexity: Still exponential, but more efficient . . . 39
3.5 Polynomial-time complexity from widening . . . 40
3.5.1 Step 1: Refactor the concrete semantics . . . 40
3.5.2 Step 2: Refactor the abstract semantics . . . 41
3.5.3 Step 3: Single-thread the abstract store . . . 41
3.5.4 Step 4: Dyck state control-flow graphs . . . 42
3.5.4.1 A preliminary analysis of complexity . . . 42
3.5.5 Step 5: Reintroduce ϵ-closure graphs . . . 43
3.6 Summary . . . 45

4. ABSTRACT GARBAGE COLLECTION AND INTROSPECTIVE PUSHDOWN CONTROL-FLOW ANALYSIS . . . 46

4.1 Introduction . . . 46
4.2 Introspection for abstract garbage collection . . . 46
4.2.1 Garbage collection in monotonic introspective pushdown systems . . . 48
4.3 Problem: Reachability for introspective pushdown systems is uncomputable . 48
4.3.1 Garbage collection in monotonic introspective pushdown systems . . . 49
4.4 Computing reachability for monotonic introspective pushdown systems . . . 50
4.4.1 Compiling to Dyck state graphs . . . 50
4.4.2 Computing a round of $\mathcal{F}$ . . . 51
4.4.3 Correctness . . . 54
4.4.4 Simplifying garbage collection in introspective pushdown systems . . . 54
4.5 An algorithm for introspective pushdown analysis with garbage collection . . 55
4.6 Summary . . . 56

5. PERFORMANCE AND APPLICATIONS OF INTROSPECTIVE PUSH-DOWN ANALYSIS **57**

5.1 Introduction 57
5.2 Experimental evaluation 57
5.2.1 Plain k-CFA vs. pushdown k-CFA 58
5.2.1.1 Comparing precision 58
5.2.1.2 Comparing speed 60
5.2.2 Analyzing real-life programs with garbage-collecting pushdown CFA 62
5.3 Applications 62
5.3.1 Escape analysis 64
5.3.2 Interprocedural dependence analysis 64
5.4 Summary 64

6. SYNTAX AND SEMANTICS OF NEBO **66**

6.1 Introduction 66
6.2 Basic Nebo Expressions 67
6.3 Assignment 69
6.4 Reductions 70
6.5 Conditional expressions 71
6.6 Stencil operations 74
6.7 Summary 77

7. IMPLEMENTATION OF NEBO **78**

7.1 Introduction 78
7.2 Template metaprogramming 79
7.3 Field type system and stencils 81
7.4 Backends 84
7.4.1 Single-core implementation 85
7.4.2 Multicore implementation 87
7.4.3 Many-core (GPU) implementation 88
7.4.4 Reduction implementation 89
7.5 Summary 90

8. CASE STUDIES OF THE USE AND PERFORMANCE OF NEBO . . **91**

8.1 Introduction 91
8.2 Simple heat equation 92
8.3 Scalar right-hand side term 95
8.4 A detailed conditional expression 98
8.5 A complex use of Nebo 99
8.6 Comparing Wasatch to Arches and ICE 101
8.7 Weakly scaling Wasatch on Titan 102
8.8 Summary 103

9. RELATED WORK **104**

9.1 Introduction 104
9.2 Control-flow analysis 104
9.2.1 Context-free analysis of higher-order programs 105
9.2.2 Calculation approach to abstract interpretation 105

9.2.3 CFL-reachability and pushdown-reachability techniques 105
9.2.4 Model-checking higher-order recursion schemes 106
9.3 Parallel processing languages for large-scale parallel computation 106
9.3.1 Models of parallelism 107
9.3.1.1 Shared memory 107
9.3.1.2 Nonshared memory 108
9.3.1.3 PGAS 110
9.3.1.3.1 Early PGAS languages. 110
9.3.1.3.2 Modern PGAS languages. 111
9.3.1.4 Other models 113
9.3.1.4.3 Coordination languages. 115
9.3.1.4.4 Graph-based languages. 115
9.3.1.4.5 Logic and functional parallel languages. 115
9.3.1.4.6 GPU languages. 116
9.3.1.4.7 Domain-specific languages. 116
9.3.2 Language design approaches 117
9.3.2.1 Novel language approach 118
9.3.2.2 Language extension 118
9.3.2.3 Language as library 119
9.3.2.4 Domain-specific languages 119
9.3.3 Failure to gain traction 120
9.4 Summary 121

10. CONCLUSIONS AND FUTURE WORK 123

10.1 Introspective pushdown analysis 123
10.2 Nebo 124

APPENDIX: PUSHDOWN PRELIMINARIES 126

REFERENCES 132

LIST OF FIGURES

1.1 Double factorial example 3
1.2 Four analyses of the program in Figure 1.1 4
2.1 The concrete configuration-space 14
2.2 The abstract configuration-space and a Haskell transliteration. 16
2.3 Function to convert a program into a pushdown automata 22
3.1 Fixed point of the function $\mathcal{F}'(M)$ 31
3.2 A Haskell implementation of pushdown control-state reachability 33
3.3 An iteration function for PDCFA with a single-threaded store 44
4.1 Haskell conversion from Dyck state graphs to NFAs 52
5.1 Runtime performance of various analyses 61
6.1 Grammar for Nebo Expressions 68
8.1 Code to evaluate equation (8.1) 95
8.2 Nebo code to evaluate equation (8.1) 95
8.3 Original code for Scalar right-hand side term 96
8.4 Initial version of Nebo code for Scalar right-hand side term 97
8.5 Current version of Nebo code for Scalar right-hand side term 98
8.6 A complex use of Nebo conditionals 99
8.7 A complex use of Nebo 100
8.8 Weak scaling of the Taylor-Green vortex on Titan 102

LIST OF TABLES

5.1 Analysis benchmark results for toy programs 59
5.2 Analysis benchmark results for real-life programs 63
8.1 Increases in performance from better use of Nebo 93
8.2 Nebo speedup on real uses in Wasatch on problem size 64^3 93
8.3 Nebo speedup on real uses in Wasatch on problem size 128^3 93
8.4 Taylor-Green vortex test results comparing Wasatch with Arches and ICE . . . 101

ACKNOWLEDGEMENTS

As all things in life, this dissertation does not represent my work alone, but is the result of many influences and people. First, I wish to thank my parents for the values they instilled in me, especially the importance of education, as well as for all their help and guidance. Next, my graditude goes to my grandparents, who showed me that hard work can be its own reward regardless of what the work is. Life can be a lonely place, especially for a graduate student 1700 miles from home. My wife, Emily, certainly has made my life less lonely, in addition to all the support she gives me through her words and deeds.

Throughout my life my friends have supported my ambitions and have helped me question my assumptions about my place in the world, through our discussions on life, the universe, and everything. A special thanks to Adrian Young, Mary Snyder, Kshiti Vaghela, Josh Woolum, Katie Palmer, Phoebe Kalinowski, and Petey Aldous.

My education has come from many influential teachers. To name just a few: Deborah Groat was the first to introduce me to Latin and the first poetry that awed me. Through her classes and teaching, I learned how rewarding learning can be. Patricia Biehl showed me a larger world and encouraged me to aim higher than I ever would have dared. Rich Holton revealed the beauty of mathematics to me. Donald Lateiner gave me a deeper appreciation of the classics and years of guidance at Ohio Wesleyan. Sean McCulloch and Alan Zaring taught me most of the computer science I know. Additionally, Sean became a friend as well as a teacher, and ended up treating me like an academic equal before I deserved it. Through Alan, I learned the satisfaction of a well-written formal proof.

Next I wish to thank my committee for their time and patience. Mary Hall and Martin Berzins have pushed me to improve my work and myself. Even before I met him, Matthew Flatt showed me the simplicity and beauty of functional programming through the educational tools he has developed. His classes further deepened my appreciation of functional programming. Through my discussions with James Sutherland, I learned many of the practical aspects of software development. Finally, I wish to thank my advisor, Matthew Might, who allowed me the freedom to make my own mistakes and the support to correct them.

CHAPTER 1

INTRODUCTION

1.1 Skirting undecidability

The halting problem is a fundamental limit to program optimization. It is impossible to determine whether or not an arbitrary Turing-complete program halts. If it were possible to solve the halting problem, program optimization would be more powerful. While this is little more than wishful thinking, we can *skirt* the halting problem in two different ways: We can approximate program behavior, and we can consider programs that are not Turing-complete.

It is possible to approximate whether or not an arbitrary Turing-complete program halts. Rather than forcing an analysis to answer either "yes" or "no" but allowing it to answer "maybe" avoids the halting problem. When approximating, an analysis can describe some program behavior, which provides the basis for many program optimizations. This dissertation formulates the pushdown control-flow analysis of (CFA2) by Vardoulakis and Shivers [135] in direct-style and combines it with abstract garbage collection [98]. Furthermore, this dissertation empirically validates these analyses against both toy and real-life benchmarks.

Alternatively, it is possible to determine program behavior of programs already proven to halt. Nebo is a domain-specific language for numerically solving PDEs in high-performance simulations. By the definition of its semantics, a Nebo program always halts. Thus, standard optimization techniques can easily apply to all Nebo programs. However, this restriction greatly limits the problems that Nebo programs can solve. Fortunately, Nebo has been embedded in C++, which allows Nebo to solve small parts of much larger programs. This dissertation defines Nebo's syntax, semantics, and implementation. Finally, this dissertation discusses real-life usage of Nebo and its performance.

1.2 Dissertation

Introspective pushdown control-flow analysis with abstract garbage collection is feasible and useful for higher-order languages, *and* Nebo is an expressive, portable, efficient,

and scalable domain-specific language embedded in C++ for numerically solving partial differential equations.

1.3 Introspective pushdown analysis

The development of a context-free[1] approach to control-flow analysis (CFA2) by Vardoulakis and Shivers provoked a shift in the static analysis of higher-order programs [135]. Prior to CFA2, a precise analysis of recursive behavior had been a challenge—even though flow analyses have an important role to play in optimization for functional languages, such as flow-driven inlining [97], interprocedural constant propagation [121] and type-check elimination [140].

While it had been possible to statically analyze recursion *soundly*, CFA2 made it possible to analyze recursion *precisely* by matching calls and returns without approximation. In its pursuit of recursion, clever engineering steered CFA2 just shy of undecidability. Its payoff is significant reductions in analysis time *as a result of* corresponding increases in precision.

For a visual measure of the impact, consider the program in Figure 1.1. In this program, `id` is the identity function, `f` is a recursive factorial function, and `g` is the sum of squares of numbers from 1 to its given argument, `n`. The result of this program is the sum of `f` applied to 3 and `g` applied to 4. Additionally, `id` is called on `f` and `g`, which does not change the result of the program but does change its control-flow. Figure 1.2 renders the abstract transition graph (a model of all possible traces through the program) for the toy program in Figure 1.1. For this example, pushdown analysis eliminates spurious return-flow from the use of recursion. However, recursion is just one problem of many for flow analysis. For instance, pushdown analysis still gets tripped up by the spurious cross-flow problem; at calls to `(id f)` and `(id g)` in the previous example, it thinks `(id g)` could be `f` *or* `g`.

Powerful techniques such as abstract garbage collection [98] were developed to solve the cross-flow problem.[2] In fact, abstract garbage collection, by itself, also delivers significant improvements to analytic speed and precision. (See Figure 1.2 again for a visualization of that impact.)

It is natural to ask: can abstract garbage collection and pushdown analysis work together? Can their strengths be multiplied? At first, the answer appears to be a disheartening "*No*."

---

[1] As in context-free language, not context-sensitivity.

[2] The cross-flow problem arises because monotonicity prevents revoking a judgment like "procedure `f` flows to `x`," or "procedure `g` flows to `x`," once it has been made.

```
(define (id x) x)

(define (f n)
  (cond [(<= n 1)  1]
        [else      (* n (f (- n 1)))]))

(define (g n)
  (cond [(<= n 1)  1]
        [else      (+ (* n n) (g (- n 1)))]))

(print (+ ((id f) 3) ((id g) 4)))
```

Figure 1.1. Double factorial example. This program is a small example to illuminate the strengths and weaknesses of both pushdown analysis and abstract garbage collection.

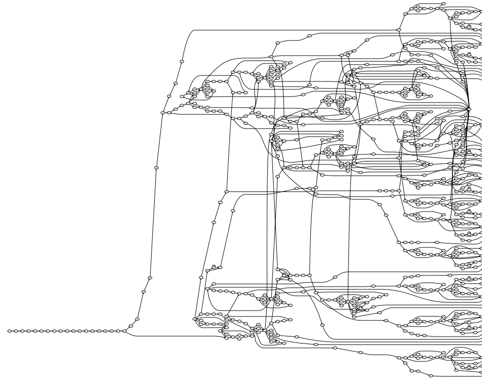
1.3.1 Accessing the whole stack *versus* just the top

Abstract garbage collection seems to require more than pushdown analysis can decidably provide: access to the full stack. Abstract garbage collection, like its name implies, discards unreachable values from an abstract store during the analysis. Like concrete garbage collection, abstract garbage collection also begins its sweep with a root set, and like concrete garbage collection, it must traverse the abstract stack to compute that root set. However, pushdown systems are restricted to viewing the top of the stack (or a bounded depth)—a condition violated by this traversal.

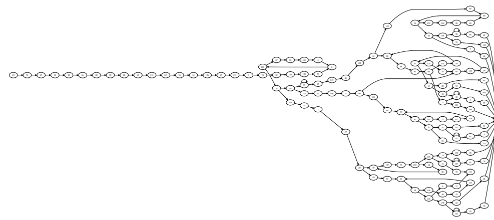
Fortunately, abstract garbage collection does not need to arbitrarily modify the stack. In fact, it does not even need to know the order of the frames; it only needs the *set* of frames on the stack. This dissertation defines a richer class of machine—*introspective* pushdown systems—that provide read-only access to the stack. Control-state reachability for the straightforward formulation of these systems ends up being uncomputable (but barely). By introducing a relatively weak monotonicity constraint on transitions, introspective pushdown systems have just enough restrictions to compute reachable control states, yet few enough to enable abstract garbage collection.

It is therefore possible to fuse the full benefits of abstract garbage collection with pushdown analysis. The dramatic reduction in abstract transition graph size from the top to the bottom in Figure 1.2 (and echoed by later benchmarks) conveys the impact of this fusion.

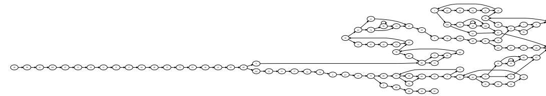
There are four secondary motivations for the static analysis in this dissertation: (1) bringing context-sensitivity to pushdown analysis; (2) exposing the context-freedom of



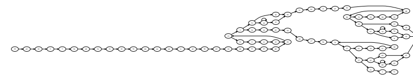
(a) without pushdown analysis or abstract GC: 653 states



(b) with pushdown only: 139 states



(c) with GC only: 105 states



(d) with pushdown analysis and abstract GC: 77 states

Figure 1.2. Four analyses of the program in Figure 1.1. The four different analysis are (a) without pushdown analysis or abstract garbage collection; (b) with only abstract garbage collection; (c) with only pushdown analysis; (d) with both pushdown analysis and abstract garbage collection. With only pushdown or abstract GC, the abstract transition graph shrinks by an order of magnitude, but in different ways. The pushdown-only analysis is confused by variables that are bound to several different higher-order functions, but for short durations. The abstract-GC-only is confused by the nontail-recursive loop structure. With both techniques enabled, the graph shrinks by nearly half yet again and fully recovers the control structure of the original program.

the analysis; (3) enabling pushdown analysis without continuation-passing style; and (4) unambiguously defining an algorithm for computing pushdown analysis, introspectively or otherwise.

In CFA2, monovariant (0CFA-like) context-sensitivity is etched directly into the abstract semantics, which are in turn, phrased in terms of an explicit (imperative) summarization algorithm for a partitioned continuation-passing style. The development here exposes the classical parameters (exposed as allocation functions in a semantics) that allow one to tune the context-sensitivity and polyvariance.

In addition, the context-freedom of CFA2 is buried implicitly inside an imperative summarization algorithm. No pushdown system or context-free grammar is explicitly identified. Thus, a necessary precursor to our work was to make the pushdown system in CFA2 explicit, and to make the control-state reachability algorithm purely functional.

A third motivation was to show that a transformation to continuation-passing style is unnecessary for pushdown analysis. In fact, pushdown analysis is arguably more natural over direct-style programs. By abstracting all machine components except for the program stack, it converts naturally and readily into a pushdown system.

Finally, to bring much-needed clarity to the algorithmic formulation of pushdown analysis, parts of a reference implementation in Haskell are included throughout the formulation. The code is as close in form to the mathematics as possible, so that where concessions are made to the implementation, they are obvious.

1.3.2 Overview

Appendix A defines the terminology and notation used in this dissertation for introspective pushdown analysis. Chapter 2 begins with with a concrete CESK-machine-style semantics for A-Normal Form λ-calculus. Next, an infinite-state abstract interpretation is constructed by bounding the C(ontrol), E(nvironment) and S(tore) portions of the machine. Uncharacteristically, the stack component—the K(ontinuation)—is left unbounded.

A shift in perspective reveals that this abstract interpretation is a pushdown system. The abstract interpretation is encoded as a pushdown automaton explicitly, and control state reachability becomes a decidable language intersection problem.

Chapter 3 extracts a rooted pushdown system from the pushdown automaton. For completeness, this chapter fully develops pushdown analysis for higher-order programs, including an efficient algorithm for computing reachable control states. This chapter finishes with characterizing complexity and demonstrating the approximations necessary to get to a polynomial-time algorithm.

Chapter 4 introduces abstract garbage collection and demonstrates that abstract garbage collection violates the pushdown model with its traversals of the stack. Towards a proof of the decidability of control-state reachability, this chapter formulates introspective pushdown systems, and recasts abstract garbage collection within this framework. This chapter shows that control-state reachability is decidable for introspective pushdown systems as well when introspective pushdown systems are subjected to a straightforward monotonicity constraint.

Finishing the static analysis portion of this dissertation, Chapter 5 discusses an implementation and empirical evaluation that shows strong synergies between pushdown analysis and abstract garbage collection, including significant reductions in the size of the abstract state transition graph.

Finally, Section 9.2 discusses work related to introspective pushdown analysis.

1.4 Nebo: A domain-specific language for numerically solving PDEs in high-performance simulations

High-performance computing applications are by definition very sensitive and adverse to inefficient code. To avoid inefficiencies, most high-performance computing code is written at a very low level. However, with the rise of new architectures, such as multicore CPUs, many-core CPUs, and GPUs, all of this code must be rewritten for each new architecture the project is to use. Rewriting all this code is an labor-intensive and error-prone process.

To create code portable between multiple architectures, this dissertation describes Nebo, an efficient domain-specific language (DSL) embedded in C++. In the parlance of Nebo's target domain (computational fluid dynamics), Nebo provides an abstraction layer for numerically solving partial differential equations for transport phenomena. In terms more familiar to computer scientists, Nebo is a declarative DSL for numeric computation over arrays of any dimensionality.

Nebo has some unusual design constraints. From the start, Nebo was built for use within the existing high-performance simulation projects, Uintah [23, 108, 22] and Wasatch [103], which together currently scale to 262K cores. Sections 1.4.2 and 1.4.3 discuss Uintah and Wasatch, respectively, in detail. Understandably the team that built Uintah and Wasatch did not want to rewrite the entire codebase in Nebo. Thus, Nebo has to allow incremental adoption: Wherever Nebo lacks needed functionality, application domain experts need to be able to write working code in C++. Generally, when new Nebo functionality becomes available, domain experts end up rewriting code using Nebo that is more flexible and easier to maintain than the original. The domain experts prefer using Nebo over handwriting C++ code because Nebo separates what computation should be performed from how that

computation should be done. With the separation of "what" from "how," Nebo is able to support single-thread, multithread, and GPU implementations of the same computation by changing compile-time and runtime options. Furthermore, Nebo's single-core implementation needs to run on par with C++ code hand-written by domain experts. To accomplish this efficiency, Nebo produces code that the compiler can easily optimize.

Nebo has a restrictive declarative syntax so that the computation can be represented as an abstract syntax tree (AST) in the C++ template system. Moreover, Nebo's restricted semantics limits what can be computed within Nebo. For example, all Nebo code will terminate: Nebo calculations only read data from any memory location a fixed number of times, usually once. Nebo calculations only read a finite amount of immutable data. Finally, Nebo calculations write results to a finite amount of mutable data, which cannot be read from within the same Nebo call. From the AST representing the Nebo calculation, Nebo can generate efficient code for a variety of backend implementations.

1.4.1 Other domain-specific languages solving PDEs

Nebo is not the first domain-specific language in this domain. POOMA [114], Blitz++ [137], Liszt [43], and OptiMesh [128] are domain-specific languages that also numerically solve partial differential equations. POOMA and Blitz++ use algebraic abstractions, like Nebo, for their syntax. Liszt and OptiMesh use geometric abstractions for their syntax. Like Nebo, POOMA, Liszt, and OptiMesh support multiple backends. Liszt and OptiMesh both support thread-based and GPU-based parallelism. POOMA supports message-passing and thread-based parallelism on CPUs only.

Nebo can be considered an indirect successor to POOMA, since the two share many design principles and goals. POOMA is also embedded in C++ and uses similar abstractions. POOMA has a higher level of abstraction, and therefore has more potential for optimization. POOMA also supports more functionality than Nebo ever plans to support. Unfortunately for POOMA, most of its development happened over a decade ago and as such shows its age. With the increase in the number of processors in supercomputers in the last decade, POOMA has not been shown to scale to the extent Nebo has. Moreover, Nebo supports GPUs, and POOMA does not. Sections 9.3.1.4.7 and 9.3.2.4 specifically discuss domain-specific languages similar to Nebo in more detail.

1.4.2 Uintah

Uintah [23, 108] is a software framework for running full-physics simulations on large-scale clusters. Full-physics means that Uintah supports simulations of both fluid dynamics

and rigid body dynamics. Additionally, Uintah supports simulations of strong interactions between fluids and solids, such as temperature and velocity interactions as well as chemical and physical transformations. Since Uintah is a framework, each of its software components implements the interactions and the numeric calculations underlying the simulations. Uintah's components primarily focus on fluid dynamics simulations, and particularly on numerically solving partial differential equations.

Uintah's main goal is to handle parallelism at a high level so that its components can focus on the numeric calculations underlying the simulations. To this end, Uintah exploits both of the major types of parallelism, task and data parallelism. For data parallelism, Uintah decomposes the physical simulation domain into patches. Uintah sets up the MPI process for each patch and handles all communication between patches. Additionally, Uintah provides support for automated load balancing of patches across processors. For task parallelism, Uintah supports a task graph. Each component specifies various tasks as well as the dependencies between tasks. Uintah handles scheduling these tasks and decides when, where, and how to run tasks in parallel. Furthermore, Uintah handles any communication that occurs between tasks. Thus, components of Uintah focus on what happens within a Uintah task on a single patch, rather than the interactions between tasks or patches.

Uintah provides support for other administrative duties for running simulations on large-scale clusters. Uintah supports adaptive mesh refinement, through which Uintah can increase or decrease resolution of individual patches based upon runtime flags. Uintah can handle memory management and data storage (file input and output). For components that support GPU execution, Uintah can manage data transfer and kernel scheduling.

All of the support Uintah provides to its components is centered around tasks that every simulation project must handle. Moreover, most of Uintah's features deal with global resource management. Thus, component developers can focus their efforts on implementing models and numeric calculation, independent of resource concerns.

1.4.3 Wasatch

Wasatch [103] is a component of Uintah. Similar to Uintah, Wasatch decomposes the simulation model and numeric calculations into a task graph. Unlike Uintah, Wasatch's task graph is inferred at runtime. Each Wasatch task declares which values it computes and upon which values it depends. At runtime, Wasatch recursively constructs the task graph by adding tasks that compute needed dependencies.

Each Wasatch task calculates one or more terms from a partial differential equation. Wasatch strives to provide a collection of expression tasks that implement partial differential equations for a variety of models and numeric calculations. Thus end users can quickly and easily design and run simulations of a variety of models. By allowing tasks to be selected at runtime, Wasatch allows end users to specify the models and equations for their simulations at runtime.

Wasatch implements its own scheduler for its task graph. Because Wasatch is a component of Uintah, Wasatch can and does use Uintah tasks. While Uintah tasks are implemented by MPI processes and threads, Wasatch tasks are implemented using threads. Wasatch focuses on a more local level of parallelism than Uintah does.

1.4.4 Use of Nebo in Wasatch

Nebo assignments and reductions are used within individual Wasatch tasks to speed up development and improve performance. Nebo's syntax is designed to avoid implementation details for two reasons. First, Wasatch developers can focus on their core competencies: The physical and chemical models and methods of simulation. Using Nebo, Wasatch developers implement numeric calculations at a higher level of abstraction, making the code easer to write, read, and maintain. Second, Nebo's abstraction creates portable code: The same Nebo code can be run on multiple architectures. Wasatch tasks written entirely in Nebo can be executed by a single thread, multiple threads or on a GPU. For these tasks, the Wasatch scheduler can decide on each iteration through the task graph how each task will be executed. Other components of Uintah that support GPU execution must maintain separate codebases for CPU and GPU execution. Maintaining different codebases for the same functionality on different architectures greatly slows down new development and regular maintenance.

While Nebo does provide flexibility and performance, Nebo's scope is very limited. Each Nebo assignment is compiled ignorant of the code surrounding it. Thus, for any given assignment, Nebo cannot tell which backend is best to use. Nebo leaves the duties of scheduling, load balancing, resource management and memory management to Wasatch and Uintah. However, Nebo does give Wasatch and Uintah more options for these duties.

Half of the source files in Wasatch that relate to field calculations use Nebo in some form. It should be noted that I do not have privileges to change Wasatch code directly, and as such I have not written any of the code in Wasatch. Many of the files that do not currently use Nebo cannot because many of these files contain calculations that Nebo cannot yet handle. This measure does not reveal the extent that Nebo is used. In files where Nebo is used, Nebo is often the primary means of performing field calculations. Wasatch

developers view Nebo as their workhorse for field calculations. For example, when stencil expressions were added to Nebo, several Wasatch developers rewrote working code to use the new feature solely for the sake of simple code and improved performance.

1.4.5 Overview

Chapter 6 contains Nebo's syntax and semantics. Chapter 7 discusses the technical details of Nebo's implementation. Chapter 8 contains case studies of realistic uses of Nebo, most of which are taken directly from Wasatch. This chapter also contains performance results from these uses of Nebo for all of Nebo's backends. Finally, Section 9.3 discusses the history of parallel processing languages and domain-specific languages similar to Nebo.

1.5 Contributions

This dissertation makes the following contributions:

1. The static analysis presented here makes the context-free aspect of CFA2 explicit: The pushdown system is clearly defined and explicitly identified. A classical CESK machine is systematically abstracted until a pushdown system emerges. Also the orthogonal frame-local-bindings aspect of CFA2 are removed, so as to rely solely on the pushdown nature of the analysis.

2. The main static analysis contribution is demonstrating the decidability of fusing abstract garbage collection with pushdown flow analysis of higher-order programs. Proof comes in the form of a fixed-point solution for computing the reachable control-states of an introspective pushdown system and an embedding of abstract garbage collection as an introspective pushdown system.

3. The claims of improved precision are empirically validated on a suite of benchmarks. These empirical results show synergies between pushdown analysis and abstract garbage collection that makes the whole greater that the sum of its parts.

4. Nebo, a portable, efficient, and scalable language for numerically solving partial differential equations is defined and implemented for single-core, multicore (thread-based), and many-core (GPU-based) architectures.

5. To prove Nebo's portability and efficiency, case studies of real users' Nebo code are presented. In particular, these case studies show that the same Nebo code can perform well on multiple architectures.

6. To prove Nebo's efficiency (compared to other software approaches) and scalability, the Taylor-Green vortex, as implemented in Wasatch, using Nebo, is presented as a benchmark. Single-core performance of Wasatch's Taylor-Green vortex compares favorably against performance of implementations in Arches and ICE, which are other components of Wasatch targeting somewhat different domains. For scalability, single-core performance of Wasatch's Taylor-Green vortex is weakly scaled to 262K processors, on the supercomputer Titan, using Uintah's MPI framework for communication between processors.

CHAPTER 2

PUSHDOWN CONTROL-FLOW ANALYSIS

2.1 Introduction

Before defining an analysis, we must have a program from a language to analyze. The analysis presented in this dissertation uses A-Normal Form λ-calculus as its input language. Section 2.2 defines the syntax, semantics, and transition relation of A-Normal Form λ-calculus. The analysis of this dissertation works directly on A-Normal Form, unlike CFA2 [135]. CFA2 targets Partitioned Continuation-Passing Style λ-calculus, which explicitly denotes continuations as distinct from lambda terms. Partitioned Continuation-Passing Style λ-calculus is generally translated from a direct-style language, such as A-Normal Form λ-calculus. By targeting A-Normal Form λ-calculus, the analysis of this dissertation does not require a translation step.

Section 2.3 defines the basic abstractions and abstract CESK machine that form the basis of the analysis in this dissertation. In particular, Section 2.3 defines the abstract configuration-space, program injection (initiation), atomic expression evaluation, reachable configurations, transition relation, and possible allocation functions as well as providing a soundness theorem of the abstract CESK machine. Unfortunately, the size of this abstract CESK machine is unbounded, which means this analysis as presented in this section is not guaranteed to terminate.

Section 2.4 recasts the abstract CESK machine of the previous section as a pushdown automaton (PDA). Pushdown automata can represent an infinite number of configurations (control states with stacks) as a finite number of control states and transitions. By generating the bounded PDA representation of the unbounded abstract CESK machine of the last section, the analysis is now guaranteed to terminate. This section also includes a theorem about the decidability of control states and a brief discussion about the size (complexity) of the control-state-space of the PDA representation.

2.2 Setting: A-Normal Form λ-calculus

Since the goal of pushdown control-flow analysis is the analysis of *higher-order languages*, I operate on the λ-calculus. To simplify presentation of the concrete and abstract semantics,

I have chosen A-Normal Form λ-calculus. (This is a strictly cosmetic choice: Pushdown control-flow analysis can be replayed *mutatis mutandis* in the standard direct-style setting as well.) ANF enforces an order of evaluation and it requires that all arguments to a function be atomic:

$$
\begin{array}{rlr}
e \in \mathsf{Exp} ::= & (\mathtt{let}\ ((v\ \mathit{call}))\ e) & \text{[non-tail call]} \\
\mid & \mathit{call} & \text{[tail call]} \\
\mid & \text{æ} & \text{[return]} \\
f, \text{æ} \in \mathsf{Atom} ::= & v \mid \mathit{lam} & \text{[atomic expressions]} \\
\mathit{lam} \in \mathsf{Lam} ::= & (\lambda\ (v)\ e) & \text{[lambda terms]} \\
\mathit{call} \in \mathsf{Call} ::= & (f\ \text{æ}) & \text{[applications]} \\
v \in \mathsf{Var} & \text{is a set of identifiers} & \text{[variables].}
\end{array}
$$

This grammar can be transliterated into Haskell:

```
data Exp    = Ret AExp
            | App Call
            | Let1 Var Call Exp
data AExp   = Ref Var
            | Lam Lambda
data Lambda = Var :=> Exp
data Call   = AExp :@ AExp
type Var    = String
```

Additionally, I use the standard instances of type classes like `Ord` and `Eq`.

I use the CESK machine of Felleisen and Friedman [47] to specify a small-step semantics for ANF. The CESK machine has an explicit stack, and under a structural abstraction, the stack component of this machine directly becomes the stack component of a pushdown system. See Figure 2.1 for the definition of concrete configurations (*Conf*) for this machine.

2.2.1 Semantics

The definition of the semantics has five components:

1. $\mathcal{I} : \mathsf{Exp} \to \mathit{Conf}$ injects an expression into a configuration:

$$c_0 = \mathcal{I}(e) = (e, [], [], \langle\rangle).$$

$$
\begin{aligned}
c \in \mathit{Conf} &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} && \text{[configurations]}\\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} && \text{[environments]}\\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathit{Clo} && \text{[stores]}\\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} && \text{[closures]}\\
\kappa \in \mathit{Kont} &= \mathit{Frame}^* && \text{[continuations]}\\
\phi \in \mathit{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} && \text{[stack frames]}\\
a \in \mathit{Addr} & \text{ is an infinite set of addresses} && \text{[addresses]}.
\end{aligned}
$$

Figure 2.1. The concrete configuration-space.

2. $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates atomic expressions:

$$
\begin{aligned}
\mathcal{A}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho) && \text{[closure creation]}\\
\mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) && \text{[variable look-up]}.
\end{aligned}
$$

3. $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$ transitions between configurations. (Defined below.)

4. $\mathcal{E} : \mathsf{Exp} \to \mathcal{P}(\mathit{Conf})$ computes the set of reachable machine configurations for a given program:

$$\mathcal{E}(e) = \{c : \mathcal{I}(e) \Rightarrow^* c\}.$$

5. $\mathit{alloc} : \mathsf{Var} \times \mathit{Conf} \to \mathit{Addr}$ chooses fresh store addresses for newly bound variables. The address-allocation function is an opaque parameter in these semantics, so that the forthcoming abstract semantics may also parameterize allocation. This parameterization provides the knob to tune the polyvariance and context-sensitivity of the resulting analysis. For the sake of defining the concrete semantics, letting addresses be natural numbers suffices, and then the allocator can choose the lowest unused address:

$$\mathit{Addr} = \mathbb{N}$$
$$\mathit{alloc}(v, (e, \rho, \sigma, \kappa)) = 1 + \max(\mathit{dom}(\sigma)).$$

2.2.1.1 Transition relation

The definition of the transition $c \Rightarrow c'$ has three rules. The first rule handles tail calls by evaluating the function into a closure, evaluating the argument into a value and then moving to the body of the closure's λ-term:

$$\overbrace{([\![\texttt{(}f\ æ\texttt{)}]\!], \rho, \sigma, \kappa)}^{c} \Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'}, \text{ where}$$

$$([\![\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}]\!], \rho') = \mathcal{A}(f, \rho, \sigma)$$

$$a = alloc(v, c)$$

$$\rho'' = \rho'[v \mapsto a]$$

$$\sigma' = \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].$$

Nontail calls push a frame onto the stack and evaluate the call:

$$\overbrace{([\![\texttt{(let ((}v\ call\texttt{))}\ e\texttt{)}]\!], \rho, \sigma, \kappa)}^{c} \Rightarrow \overbrace{(call, \rho, \sigma, (v, e, \rho) : \kappa)}^{c'}.$$

Function return pops a stack frame:

$$\overbrace{(æ, \rho, \sigma, (v, e, \rho') : \kappa)}^{c} \Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'}, \text{ where}$$

$$a = alloc(v, c)$$

$$\rho'' = \rho'[v \mapsto a]$$

$$\sigma' = \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].$$

2.3 An infinite-state abstract interpretation

The first step toward a static analysis is an abstract interpretation into an *infinite* state-space. A pushdown analysis abstracts away less than a traditional control-flow analysis. Specifically, the stack height is left unbounded.

Figure 2.2 details the abstract configuration-space. To synthesize it, I force addresses to be a finite set, but crucially, I leave the stack untouched. When I compact the set of addresses into a finite set, the machine may run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple closures to reside at the same address. As a result, there is no choice but to force the range of the store to become a power set in the abstract configuration-space. The abstract transition relation has components analogous to those from the concrete semantics.

2.3.1 Program injection

The abstract injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{Conf}$ pairs an expression with an empty environment, an empty store and an empty stack to create the initial abstract configuration:

$$\hat{c}_0 = \hat{\mathcal{I}}(e) = (e, [], [], \langle\rangle).$$

$$
\begin{aligned}
\hat{c} \in \widehat{Conf} &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} && \text{[configurations]}\\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} && \text{[environments]}\\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right) && \text{[stores]}\\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} && \text{[closures]}\\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^{*} && \text{[continuations]}\\
\hat{\phi} \in \widehat{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} && \text{[stack frames]}\\
\hat{a} \in \widehat{Addr} & \text{ is a \textit{finite} set of addresses} && \text{[addresses].}
\end{aligned}
$$

```
-- Abstract state-space:
type AConf  = (Exp, AEnv, AStore, AKont)
type AEnv   = Var :-> AAddr
type AStore = AAddr :-> AD
type AD     =  (AVal)
data AVal   = AClo (Lambda, AEnv)
type AKont  = [AFrame]
type AFrame = (Var, Exp, AEnv)

data AAddr = ABind Var AContext
type AContext = [Call]
```

Figure 2.2. The abstract configuration-space and a Haskell transliteration. In the Haskell code, the abstract addresses are defined, such that they are able to support k-CFA-style polyvariance.

2.3.2 Atomic expression evaluation

The abstract atomic expression evaluator, $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{Env} \times \widehat{Store} \to \mathcal{P}(\widehat{Clo})$, returns the value of an atomic expression in the context of an environment and a store; it returns a *set* of abstract closures:

$$
\begin{aligned}
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\} && \text{[closure creation]}\\
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) && \text{[variable look-up].}
\end{aligned}
$$

The corresponding implementation in Haskell of the mathematical version:

```
aeval :: (AExp,AEnv,AStore) -> AD
aeval (Ref v, ρ, σ) = σ!!(ρ!v)
aeval (Lam l, ρ, σ) = set $ AClo (l, ρ)
```

2.3.3 Reachable configurations

The abstract program evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \to \mathcal{P}(\widehat{Conf})$ returns all of the configurations reachable from the initial configuration:

$$\hat{\mathcal{E}}(e) = \left\{\hat{c} : \hat{\mathcal{I}}(e) \leadsto^* \hat{c}\right\}.$$

Because there are an infinite number of abstract configurations, a naïve implementation of this function may not terminate. Pushdown analysis provides a way of precisely computing this set and both finitely and compactly representing the result.

2.3.4 Transition relation

The abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ has three rules, one of which has become nondeterministic. In Haskell, the abstract transition relation is encoded as a function that returns lists of states:

```
astep :: AConf -> [AConf]
```

A tail call may fork because there could be multiple abstract closures that it is invoking:

$$\overbrace{(\llbracket\texttt{(}f\ \text{æ}\texttt{)}\rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'}, \text{ where}$$
$$(\llbracket\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}\rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{a} = \widehat{alloc}(v, \hat{c})$$
$$\hat{\rho}'' = \hat{\rho}'[v \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})].$$

In Haskell:

```
astep (App (f :@ ae), ρ, σ, κ) = [(e, ρ'', σ', κ) |
   AClo(v :=> e, ρ') <- Set.toList $ aeval(f, ρ, σ),
   let a = aalloc(v, App (f :@ ae)),
   let ρ'' = ρ' // [v ==> a],
   let σ' = σ  [a ==> aeval(ae, ρ, σ)] ]
```

Partial orders are defined in Section 2.3.6, but updating a store becomes:

$$(\hat{\sigma} \sqcup \hat{\sigma}')(\hat{a}) = \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a}).$$

A nontail call pushes a frame onto the stack and evaluates the call:

$$\overbrace{(\llbracket\texttt{(let ((}v\ call\texttt{))}\ e\texttt{)}\rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(call, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}) : \hat{\kappa})}^{\hat{c}'}.$$

In Haskell:

```
astep (Let1 v call e, ρ, σ, κ) =
  [(App call, ρ, σ, (v, e, ρ) : κ)]
```

A function return pops a stack frame:

$$\overbrace{(æ, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}') : \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'} \text{, where}$$

$$\hat{a} = \widehat{alloc}(v, \hat{c})$$

$$\hat{\rho}'' = \hat{\rho}'[v \mapsto \hat{a}]$$

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})].$$

In Haskell:

```
astep (Ret ae, ρ, σ, (v, e, ρ') : κ) = [(e, ρ'', σ', κ)]
 where a = aalloc(v, Ret ae)
       ρ'' = ρ' // [v ==> a]
       σ' = σ   [a ==> aeval(ae, ρ, σ)]
```

2.3.5 Allocation: Polyvariance and context-sensitivity

In the abstract semantics, the abstract allocation function $\widehat{alloc} : \mathsf{Var} \times \widehat{Conf} \to \widehat{Addr}$ determines the polyvariance of the analysis. In a control-flow analysis, *polyvariance* literally refers to the number of abstract addresses (variants) there are for each variable. An advantage of this framework over CFA2 is that varying this abstract allocation function instantiates pushdown versions of classical flow analyses. All of the following allocation approaches can be used with the abstract semantics. The abstract allocation function is a parameter to the analysis.

2.3.5.1 Monovariance: Pushdown 0CFA

Pushdown 0CFA uses variables themselves for abstract addresses:

$$\widehat{Addr} = \mathsf{Var}$$

$$alloc(v, \hat{c}) = v.$$

2.3.5.2 Context-sensitive: Pushdown 1CFA

Pushdown 1CFA pairs the variable with the current expression to get an abstract address:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}$$

$$alloc(v, (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = (v, e).$$

2.3.5.3 Polymorphic splitting: Pushdown poly/CFA

Assuming the program compiled is from a programming language with let-bound polymorphism and the let-bound functions were marked, polymorphic splitting can be used:

$$\widehat{Addr} = \mathsf{Var} + \mathsf{Var} \times \mathsf{Exp}$$
$$alloc(v, ([\![(f\ æ)]\!], \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = \begin{cases} (v, [\![(f\ æ)]\!]) & f \text{ is let-bound} \\ v & \text{otherwise.} \end{cases}$$

2.3.5.4 Pushdown k-CFA

For pushdown k-CFA, we need to look beyond the current state and at the last k states. By concatenating the expressions in the last k states together, and pairing this sequence with a variable we get pushdown k-CFA:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}^k$$
$$\widehat{alloc}(v, \langle (e_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\kappa}_1), \ldots \rangle) = (v, \langle e_1, \ldots, e_k \rangle).$$

2.3.6 Partial orders

Each set $\hat{X}$ inside the abstract configuration-space uses the natural partial order, $(\sqsubseteq_{\hat{X}}) \subseteq \hat{X} \times \hat{X}$. Abstract addresses and syntactic sets have flat partial orders. For the other sets, the partial order lifts:

- point-wise over environments:

$$\hat{\rho} \sqsubseteq \hat{\rho}' \text{ iff } \hat{\rho}(v) = \hat{\rho}'(v) \text{ for all } v \in dom(\hat{\rho});$$

- component-wise over closures:

$$(lam, \hat{\rho}) \sqsubseteq (lam, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}';$$

- point-wise over stores:

$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma});$$

- component-wise over frames:

$$(v, e, \hat{\rho}) \sqsubseteq (v, e, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}';$$

- element-wise over continuations:

$$\langle \hat{\phi}_1, \ldots, \hat{\phi}_n \rangle \sqsubseteq \langle \hat{\phi}'_1, \ldots, \hat{\phi}'_n \rangle \text{ iff } \hat{\phi}_i \sqsubseteq \hat{\phi}'_i\text{; and}$$

- component-wise across configurations:

$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \sqsubseteq (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'.$$

In Haskell, I define a typeclass for lattices:

```
class Lattice a where
 bot :: a
 top :: a
 (⊑) :: a -> a -> Bool
 (⊔) :: a -> a -> a
 (⊓) :: a -> a -> a
```

Instances can be lifted to sets and maps:

```
instance (Ord s, Eq s) => Lattice (ℙ s) where
 bot = Set.empty
 top = error "no representation of universal set"
 x ⊔ y = x ‘Set.union‘ y
 x ⊓ y = x ‘Set.intersection‘ y
 x ⊑ y = x ‘Set.isSubsetOf‘ y

instance (Ord k, Lattice v) => Lattice (k :-> v) where
 bot = Map.empty
 top = error "no representation of top map"
 f ⊑ g = Map.isSubmapOfBy (⊑) f g
 f ⊔ g = Map.unionWith (⊔) f g
 f ⊓ g = Map.intersectionWith (⊓) f g

(⊔) :: (Ord k, Lattice v) => (k :-> v) -> [(k,v)] -> (k :-> v)
f ⊔ [(k,v)] = Map.insertWith (⊔) k v f

(!!) :: (Ord k, Lattice v) => (k :-> v) -> k -> v
f !! k = Map.findWithDefault bot k f
```

2.3.7 Soundness

To prove soundness, an abstraction map α connects the concrete and abstract configuration-spaces:

$$\begin{aligned}
\alpha(e, \rho, \sigma, \kappa) &= (e, \alpha(\rho), \alpha(\sigma), \alpha(\kappa)) \\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) \\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \{\alpha(\sigma(a))\} \\
\alpha\langle \phi_1, \ldots, \phi_n \rangle &= \langle \alpha(\phi_1), \ldots, \alpha(\phi_n) \rangle \\
\alpha(v, e, \rho) &= (v, e, \alpha(\rho)) \\
\alpha(a) &\text{ is determined by the allocation functions.}
\end{aligned}$$

It is then easy to prove that the abstract transition relation simulates the concrete transition relation:

Theorem 1 *If $\alpha(c) \sqsubseteq \hat{c}$ and $c \Rightarrow c'$, then there exists $\hat{c}' \in \widehat{Conf}$ such that $\alpha(c') \sqsubseteq \hat{c}'$ and $\hat{c} \rightsquigarrow \hat{c}'$.*

Proof. The proof follows by case analysis on the expression in the configuration. It is a straightforward adaptation of similar proofs, such as that of Might [93] for k-CFA. ■

2.4 From the abstracted CESK machine to a PDA

The previous section constructed an infinite-state abstract interpretation of the CESK machine. The infinite-state nature of the abstraction makes it difficult to see how to answer static analysis questions. Consider, for instance, a control flow-question:

> At the call site `(`f $æ$`)`, may a closure over *lam* be called?

If the abstracted CESK machine were a finite-state machine, an algorithm could answer this question by enumerating all reachable configurations and looking for an abstract configuration $([\![(f\ æ)]\!], \hat{\rho}, \hat{\sigma}, \hat{\kappa})$ in which $(lam, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$. However, because the abstracted CESK machine may contain an infinite number of reachable configurations, an algorithm cannot enumerate them.

Fortunately, the abstracted CESK can be recast as a special kind of infinite-state system: A pushdown automaton (PDA). Pushdown automata occupy a sweet spot in the theory of computation: They have an infinite configuration-space, yet many useful properties (e.g., word membership, nonemptiness, control-state reachability) and remain decidable. Once

the abstracted CESK machine becomes a PDA, we can answer the control-flow question by checking whether a specific regular language, when intersected with the language of the PDA, turns into the empty language.

The recasting as a PDA is a shift in perspective. A configuration has an expression, an environment and a store. A stack character is a frame. I make the alphabet the set of control states, so that the language accepted by the PDA will be sequences of control-states visited by the abstracted CESK machine. Thus, every transition will consume the control-state to which it transitioned as an input character. Figure 2.3 defines the program-to-PDA conversion function $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$. (Note the implicit use of the isomorphism $Q \times \widehat{Kont} \cong \widehat{Conf}$.)

At this point, we can answer questions about whether a specified control state is reachable by formulating a question about the intersection of a regular language with a context-free language described by the PDA. That is, if we want to know whether the control state $(e', \hat{\rho}, \hat{\sigma})$ is reachable in a program e, we can reduce the problem to determining:

$$\Sigma^* \cdot \left\{(e', \hat{\rho}, \hat{\sigma})\right\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset,$$

where $L_1 \cdot L_2$ is the concatenation of formal languages L_1 and L_2.

Theorem 2 *Control-state reachability is decidable.*

$$\begin{aligned}
\widehat{\mathcal{PDA}}(e) &= (Q, \Sigma, \Gamma, \delta, q_0, F, \langle\rangle), \text{ where} \\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\
\Sigma &= Q \\
\Gamma &= \widehat{Frame} \\
(q, \epsilon, q', q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q, \hat{\phi}_-, q', q') \in \delta &\text{ iff } (q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q, \hat{\phi}'_+, q', q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\phi}' : \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e) \\
F &= Q.
\end{aligned}$$

Figure 2.3. Function to convert a program into a pushdown automata. $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$.

Proof. The intersection of a regular language and a context-free language is context-free. The emptiness of a context-free language is decidable. ■

Now, consider how to use control-state reachability to answer the control-flow question from earlier. There are a finite number of possible control states in which the λ-term *lam* may flow to the function f in call site `(`f æ`)`; let us call the this set of states $\hat{S}$:

$$\hat{S} = \left\{ ([\![(f\ \text{æ})]\!], \hat{\rho}, \hat{\sigma}) : (lam, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \text{ for some } \hat{\rho}' \right\}.$$

What we want to know is whether any state in the set $\hat{S}$ is reachable in the PDA. In effect what we are asking is whether there exists a control state $q \in \hat{S}$ such that:

$$\Sigma^* \cdot \{q\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset.$$

If this is true, then *lam* may flow to f; if false, then it does not.

2.4.1 Problem: Doubly exponential complexity

The nonemptiness-of-intersection approach establishes decidability of pushdown control-flow analysis. Two exponential complexity barriers make this technique impractical.

First, there are an exponential number of both environments ($|\widehat{Addr}|^{|\mathsf{Var}|}$) and stores ($2^{|\widehat{Clo}| \times |\widehat{Addr}|}$) to consider for the set $\hat{S}$. On top of that, computing the intersection of a regular language with a context-free language will require enumeration of the (exponential) control-state-space of the PDA. As a result, this approach is doubly exponential. The goal of the next chapter is to lower the complexity of pushdown control-flow analysis.

2.5 Summary

This chapter defines the basics necessary for a static analysis that approximates program behavior as a pushdown automaton. Additionally, this chapter overcomes the first challenge of a pushdown static analysis: Bounding the search-space. By shifting one's perspective from searching the configuration-space (control-states with stacks) to searching just the control-state-space, one goes from exploring a potentially infinite set to exploring a provably finite set. This shift does not harm precision because the PDA framework implicitly manages the stack. Unfortunately, while the analysis will terminate and provides desirable and decidable properties, the exponential complexity of the analysis in the form presented in this chapter is intractable. Fortunately, the next chapter solves this issue by limiting the search-space of the analysis only to reachable control states. Furthermore, the next chapter reduces the complexity through widening, which can negatively impact precision but reduce the complexity of the analysis to polynomial-time.

CHAPTER 3

COMPUTABLE AND TRACTABLE PUSHDOWN CONTROL-FLOW ANALYSIS

3.1 Introduction

At the end of the last chapter, we saw that control-flow analysis reduces to the reachability of certain control states within a pushdown system. The last chapter also determined reachability by converting the abstracted CESK machine into a PDA, and using emptiness-testing on a language derived from that PDA. Unfortunately, this approach is deeply exponential.

Since control-flow analysis reduced to the reachability of control-states in the PDA, one could skip the language problems and go directly to reachability algorithms of Bouajjani et al. [30]; Kodumal and Aiken [78]; Reps [112] and Reps et al. [113] that determine the reachable *configurations* within a pushdown system. These algorithms are even polynomial-time. Unfortunately, some of them are polynomial-time in the number of control states, and in the abstracted CESK machine, there are an exponential number of control states. We do not want to *enumerate* the entire control state-space, or else the search becomes exponential in even the best case.

To avoid this worst-case behavior, this chapter presents a straightforward pushdown-reachability algorithm that considers only the *reachable* control states. This reachability algorithm is cast as a fixed-point iteration, which incrementally construct the reachable subset of a pushdown system. I call these algorithms "iterative Dyck state graph construction."

Section 3.2 defines what a Dyck state graph is and the abstract transition relation associated with Dyck state graphs. Section 3.3 defines the pushdown analysis algorithm as a finding the least fixed point iteratively and discusses the complexity of this approach. Section 3.4 describes a more efficient algorithm based on work-lists and ϵ-graphs. Additionally, Section 3.4 includes a theorem proving termination and a discussion of complexity. Since the work-list algorithm of Section 3.4 still has exponential-time complexity, Section

3.5 describes how to add widening to the analysis to achieve polynomial-time complexity. Section 3.5 ends with a discussion of the complexity of the widened work-list algorithm.

3.2 Dyck state graph

A *Dyck state graph* is a compacted, rooted pushdown system $G = (S, \Gamma, E, s_0)$, in which:

1. S is a finite set of nodes;
2. Γ is a set of frames;
3. $E \subseteq S \times \Gamma_\pm \times S$ is a set of stack-action edges; and
4. s_0 is an initial state;

such that for any node $s \in S$, it must be the case that:

$$(s_0, \langle\rangle) \underset{G}{\overset{*}{\longmapsto}} (s, \vec{\gamma}) \text{ for some stack } \vec{\gamma}.$$

In other words, a Dyck state graph is equivalent to a rooted pushdown system in which there is a legal path to every control state from the initial control state.[1]

We use $\mathbb{DSG}$ to denote the class of Dyck state graphs. Clearly:

$$\mathbb{DSG} \subset \mathbb{RPDS}.$$

A Dyck state graph is a rooted pushdown system with the "fat" trimmed off; in this case, unreachable control states and unreachable transitions are the "fat."

We can formalize the connection between rooted pushdown systems and Dyck state graphs with a map:

$$\mathcal{DSG} : \mathbb{RPDS} \to \mathbb{DSG}.$$

Given a rooted pushdown system $M = (Q, \Gamma, \delta, q_0)$, its equivalent Dyck state graph is $\mathcal{DSG}(M) = (S, \Gamma, E, q_0)$, where the set S contains reachable nodes:

$$S = \left\{ q : (q_0, \langle\rangle) \underset{M}{\overset{*}{\longmapsto}} (q, \vec{\gamma}) \text{ for some stack } \vec{\gamma} \right\},$$

and the set E contains reachable edges:

$$E = \left\{ q \overset{g}{\rightarrowtail} q' : q \underset{M}{\overset{g}{\longmapsto\!\!\!\twoheadrightarrow}} q' \right\},$$

and $s_0 = q_0$.

[1] The term *Dyck state graph* was chosen because the sequences of stack actions along valid paths through the graph correspond to substrings in Dyck languages. A *Dyck language* is a language of balanced, "colored" parentheses. In this case, each character in the stack alphabet is a color.

In practice, the real difference between a rooted pushdown system and a Dyck state graph is that our rooted pushdown system will be defined intensionally (having come from the components of an abstracted CESK machine), whereas the Dyck state graph will be defined extensionally, with the contents of each component explicitly enumerated during its construction.

In this case, looking at the Haskell definition of a DSG helps:

```
type DSG control frame = (Edges control frame, control)
type Edges control frame = control :->  (StackAct frame, control)
```

The near-term goals are (1) to convert the abstracted CESK machine of the last chapter into a rooted pushdown system and (2) to find an *efficient* method for computing an equivalent Dyck state graph from a rooted pushdown system.

To convert the abstracted CESK machine into a rooted pushdown system, we use the function $\widehat{\mathcal{RPDS}} : \mathsf{Exp} \to \mathbb{RPDS}$:

$$\begin{aligned}
\widehat{\mathcal{RPDS}}(e) &= (Q, \Gamma, \delta, q_0) \\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\
\Gamma &= \widehat{Frame} \\
q \overset{\epsilon}{\rightarrowtail} q' \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
q \overset{\hat{\phi}_-}{\rightarrowtail} q' \in \delta &\text{ iff } (q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
q \overset{\hat{\phi}_+}{\rightarrowtail} q' \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\phi} : \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e).
\end{aligned}$$

In Haskell, the abstract transition relation is embedded as a pushdown transition relation as follows:

```
adelta :: TopDelta AControl AFrame
adelta (e, ρ, σ) γ = [ ((e', ρ', σ'), g) |
 (e', ρ', σ', κ) <- astep (e, ρ, σ, [γ]),
 let g = case κ of
          []        -> Pop γ
          [γ1 , _ ] -> Push γ1
          [ _ ]     -> Unch ]
```

```
adelta' :: NopDelta AControl AFrame
adelta' (e, ρ, σ) = [ ((e', ρ', σ'), g) |
 (e', ρ', σ', κ) <- astep (e, ρ, σ, []),
 let g = case κ of
          [γ1]  -> Push γ1
          [ ]   -> Unch ]
```

3.3 Compacting a rooted pushdown system into a Dyck state graph

We now turn our attention to compacting a rooted pushdown system (defined intensionally) into a Dyck state graph (defined extensionally). That is, we want to find an implementation of the function $\mathcal{DSG}$. To do so, we first phrase the Dyck state graph construction as the least fixed point of a monotonic function. This will provide a method (albeit an inefficient one) for computing the function $\mathcal{DSG}$. The next section looks at an optimized work-list driven algorithm that avoids the inefficiencies of this version.

The function $\mathcal{F} : \mathbb{RPDS} \to (\mathbb{DSG} \to \mathbb{DSG})$ generates the monotonic iteration function we need:

$$\begin{aligned}
\mathcal{F}(M) &= f, \text{ where} \\
M &= (Q, \Gamma, \delta, q_0) \\
f(S, \Gamma, E, s_0) &= (S', \Gamma, E', s_0), \text{ where} \\
S' &= S \cup \left\{ s' : s \in S \text{ and } s \underset{M}{\longmapsto\!\!\!\!\rightarrow} s' \right\} \cup \{s_0\} \\
E' &= E \cup \left\{ s \overset{g}{\rightarrowtail} s' : s \in S \text{ and } s \overset{g}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} s' \right\}.
\end{aligned}$$

Given a rooted pushdown system M, each application of the function $\mathcal{F}(M)$ accretes new edges at the frontier of the Dyck state graph. Once the algorithm reaches a fixed point, the Dyck state graph is complete:

Theorem 3 $\mathcal{DSG}(M) = \text{lfp}(\mathcal{F}(M))$.

Proof. Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathcal{F}(M)$. Observe that $\text{lfp}(f) = f^n(\emptyset, \Gamma, \emptyset, q_0)$ for some n. When $N \subseteq M$, then it easy to show that $f(N) \subseteq M$. Hence, $\mathcal{DSG}(M) \supseteq \text{lfp}(\mathcal{F}(M))$.

To show $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathcal{DSG}(M)$ that is not in $\text{lfp}(\mathcal{F}(M))$. Let (s, g, s') be one such edge, such that the state s *is* in $\text{lfp}(\mathcal{F}(M))$. Let m be the lowest natural number such that s appears

in $f^m(M)$. By the definition of f, this edge must appear in $f^{m+1}(M)$, which means it must also appear in $\text{lfp}(\mathcal{F}(M))$, which is a contradiction. Hence, $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$. ■

3.3.1 Complexity: Polynomial and exponential

To determine the complexity of this algorithm, there are two questions to ask: How many times would the algorithm invoke the iteration function in the worst case, and how much does each invocation cost in the worst case? The size of the final Dyck state graph bounds the run-time of the algorithm. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. Since there are at most $2|\Gamma|m^2 + m^2$ edges in the final graph, the maximum number of iterations is $2|\Gamma|m^2 + m^2$.

The cost of computing each iteration is harder to bound. The cost of determining whether to add a push edge is proportional to the size of the stack alphabet, while the cost of determining whether to add an ϵ-edge is constant, so the cost of determining all new push and pop edges to add is proportional to $|\Gamma|m + m$. Determining whether or not to add a pop edge is expensive. To add the pop edge $s \rightarrowtail^{\gamma_-} s'$, we must prove that there exists a configuration-path to the control state s, in which the character γ is on the top of the stack. This reduces to a CFL-reachability query [90] at each node, the cost of which is $O(|\Gamma_\pm|^3 m^3)$ [78].

To summarize, in terms of the number of reachable control states, the complexity of the most recent algorithm is:

$$O((2|\Gamma|m^2 + m^2) \times (|\Gamma|m + m + |\Gamma_\pm|^3 m^3)) = O(|\Gamma|^4 m^5).$$

While this approach is polynomial in the number of reachable control states, it is far from efficient. The next section provides an optimized version of this fixed-point algorithm that maintains a work-list and an ϵ-closure graph to avoid spurious recomputation.

Moreover, the complexity analysis above is carefully phrased in terms of "reachable" control states because, in practice, Dyck state graphs will be extremely sparse, and because, the maximum number of control states is exponential in the size of the input program. After the subsequent refinement, I develop a hierarchy of pushdown control-flow analyses that employs widening to achieve a polynomial-time algorithm at its foundation in Section 3.5.

3.4 An efficient algorithm: Work-lists and ϵ-closure graphs

The last section developed a fixed-point formulation of the Dyck state graph construction algorithm, but found that, in each iteration, it wasted effort by passing over all discovered states and edges, even though most will not contribute new states or edges. Taking a cue from graph search, we can adapt the fixed-point algorithm with a work-list. That is, our next algorithm will keep a work-list of new states and edges to consider, instead of reconsidering all of them. In each iteration, it will pull new states and edges from the work list, insert them into the Dyck state graph and then populate the work-list with new states and edges that have to be added as a consequence of the recent additions.

3.4.1 ϵ-closure graphs

Figuring out what edges to add as a consequence of another edge requires care, for adding an edge can have ramifications on distant control states. Consider, for example, adding the ϵ-edge $q \rightarrowtail^{\epsilon} q'$ into the following graph:

$$q_0 \xrightarrow{\gamma_+} q \qquad q' \xrightarrow{\gamma_-} q_1$$

As soon this edge drops in, an ϵ-edge "implicitly" appears between q_0 and q_1 because the net stack change between them is empty; the resulting graph looks like:

$$q_0 \xrightarrow{\gamma_+} q \xrightarrow{\epsilon} q' \xrightarrow{\gamma_-} q_1 \qquad (\text{dashed } q_0 \overset{\epsilon}{\dashrightarrow} q_1)$$

where the implicit ϵ-edge is illustrated as a dotted line.

Keeping track of these implicit edges requires constructing a second graph in conjunction with the Dyck state graph: An ϵ-closure graph. In the ϵ-closure graph, every edge indicates the existence of a no-net-stack-change path between control states. The ϵ-closure graph simplifies the task of figuring out which states and edges are impacted by the addition of a new edge.

Formally, an **ϵ-closure graph**, is a pair $G_\epsilon = (N, H)$, where N is a set of states, and $H \subseteq N \times N$ is a set of edges. Of course, all ϵ-closure graphs are reflexive: Every node has a self loop. The symbol $\mathbb{ECG}$ denotes the class of all ϵ-closure graphs.

I have two notations for finding ancestors and descendants of a state in an ϵ-closure graph $G_\epsilon = (N, H)$:

$$\overleftarrow{G}_\epsilon[s] = \left\{s' : (s', s) \in H\right\} \qquad \text{[ancestors]}$$

$$\overrightarrow{G}_\epsilon[s] = \left\{s' : (s, s') \in H\right\} \qquad \text{[descendants]}.$$

3.4.2 Integrating a work-list

Since only new states and edges are considered in each iteration, we need a work-list, or in this case, two work-graphs. A Dyck state work-graph is a pair $(\Delta S, \Delta E)$ in which the set ΔS contains a set of states to add, and the set ΔE contains edges to be added to a Dyck state graph.[2] $\Delta\mathbb{DSG}$ refers to the class of all Dyck state work-graphs.

An ϵ-closure work-graph is a set ΔH of new ϵ-edges. $\Delta\mathbb{ECG}$ refers to the class of all ϵ-closure work-graphs.

3.4.3 A new fixed-point iteration-space

Instead of consuming a Dyck state graph and producing a Dyck state graph, the new fixed-point iteration function will consume and produce a Dyck state graph, an ϵ-closure graph, a Dyck state work-graph and an ϵ-closure work graph. Hence, the iteration space of the new algorithm is:

$$IDSG = \mathbb{DSG} \times \mathbb{ECG} \times \Delta\mathbb{DSG} \times \Delta\mathbb{ECG}.$$

(The I in $IDSG$ stands for *intermediate*.)

3.4.4 The ϵ-closure graph work-list algorithm

The function $\mathcal{F}' : \mathbb{RPDS} \to (IDSG \to IDSG)$ generates the required iteration function (Figure 3.1). Please note that union is implicitly distributed across tuples:

$$(X, Y) \cup (X', Y') = (X \cup X, Y \cup Y').$$

The functions *sprout*, *addPush*, *addPop*, *addEmpty* calculate the additional Dyck state graph and ϵ-closure graph edges (potentially) introduced by a new state or edge. The function *sprout* adds a new state to the Dyck state graph and calculates any new push or ϵ edges beginning at the new state. The function *addPush* adds a new push edge to the Dyck state graph and calculates any pop edges that are enabled by the new push edge. The function *addPop* adds a new pop edge to the Dyck state graph and calculates all ϵ edges that result from all matching push edges. Finally, the function *addEmpty* adds a new ϵ-closure graph edge and calculates any new edges that need to be added to complete the transitive closure of the ϵ-closure graph. These functions are defined formally later in this section. In general, these functions do not use direct mutual recursion to find all new edges

[2] Technically, a work-graph is not an actual graph, since $\Delta E \not\subseteq \Delta S \times \Gamma_{\pm} \times \Delta S$; a work-graph is just a set of nodes and a set of edges.

$$
\begin{aligned}
\mathcal{F}'(M) &= f, \text{ where} \\
M &= (Q, \Gamma, \delta, q_0) \\
f(G, G_\epsilon, \Delta G, \Delta H) &= (G', G'_\epsilon, \Delta G', \Delta H' - H), \text{ where} \\
(S, \Gamma, E, s_0) &= G \\
(S, H) &= G_\epsilon \\
(\Delta S, \Delta E) &= \Delta G \\
(\Delta E_0, \Delta H_0) &= \bigcup_{s \in \Delta S} \mathit{sprout}_M(s) \\
(\Delta E_1, \Delta H_1) &= \bigcup_{(s, \gamma_+, s') \in \Delta E} \mathit{addPush}_M(G, G_\epsilon)(s, \gamma_+, s') \\
(\Delta E_2, \Delta H_2) &= \bigcup_{(s, \gamma_-, s') \in \Delta E} \mathit{addPop}_M(G, G_\epsilon)(s, \gamma_-, s') \\
(\Delta E_3, \Delta H_3) &= \bigcup_{(s, \epsilon, s') \in \Delta E} \mathit{addEmpty}_M(G, G_\epsilon)(s, s') \\
(\Delta E_4, \Delta H_4) &= \bigcup_{(s, s') \in \Delta H} \mathit{addEmpty}_M(G, G_\epsilon)(s, s') \\
S' &= S \cup \Delta S \\
E' &= E \cup \Delta E \\
H' &= H \cup \Delta H \\
\Delta E' &= \Delta E_0 \cup \Delta E_1 \cup \Delta E_2 \cup \Delta E_3 \cup \Delta E_4 \\
\Delta S' &= \left\{ s' : (s, g, s') \in \Delta E' \right\} \\
\Delta H' &= \Delta H_0 \cup \Delta H_1 \cup \Delta H_2 \cup \Delta H_3 \cup \Delta H_4 \\
G' &= (S \cup \Delta S, \Gamma, E', q_0) \\
G'_\epsilon &= (S', H') \\
\Delta G' &= (\Delta S' - S', \Delta E' - E').
\end{aligned}
$$

Figure 3.1. Fixed point of the function $\mathcal{F}'(M)$. This fixed point contains the Dyck state graph of the rooted pushdown system M.

that result from adding a new state or edge. Instead these functions add new states and edges to the worklist, as seen in Figure 3.1, using a breadth-first search style approach to finding all new states and edges.

In Haskell, the function `dsg` will invoke the fixed point solver:

```
dsg :: (Ord control, Ord frame) =>
       (Delta control frame) ->
       control ->
       frame ->
       DSG control frame
dsg (δ, δ') q0 0 =
 (summarize (δ, δ') etg1 ecg1 [] dE dH, q0) where
 etg1 = (Map.empty // [q0 ==> Set.empty],
         Map.empty // [q0 ==> Set.empty])
 ecg1 = (Map.empty // [q0 ==> set q0],
         Map.empty // [q0 ==> set q0])
 (dE, dH) = sprout (δ, δ') q0
```

To expose the structure of the computation, I have added a few types:

```
-- A set of edges, encoded as a map:
type Edges control frame =
 control :-> ℙ (StackAct frame,control)

-- Epsilon edges:
type EpsEdge control = (control, control)

-- Explicit transition graph:
type ETG control frame =
 (Edges control frame, Edges control frame)

-- Epsilon closure graph:
type ECG control =
 (control :-> ℙ(control), control :-> ℙ(control))
```

Figure 3.2 provides the Haskell code for `summarize`, which conducts the fixed point calculation, the executable equivalent of Figure 3.1:

```
summarize (δ, δ') (fw, bw) (fe, be) [] [] [] = fw

summarize (δ, δ') (fw, bw) (fe, be) (q:dS) [] []
 | fe `contains` q = summarize (δ, δ') (fw, bw) (fe, be) dS [] []
summarize (δ, δ') (fw, bw) (fe, be) (q:dS) [] [] =
 summarize (δ, δ') (fw', bw') (fe', be') dS dE' dH' where
  (dE', dH') = sprout (δ, δ') q
  fw' = fw ⊔ [q ==> Set.empty]
  bw' = bw ⊔ [q ==> Set.empty]
  fe' = fe ⊔ [q ==> set q]
  be' = be ⊔ [q ==> set q]

summarize (δ, δ') (fw, bw) (fe, be) dS ((q, g, q'):dE) []
 | (q, g, q') `isin'` fw  = summarize (δ, δ') (fw, bw) (fe, be) dS dE []
summarize (δ, δ') (fw, bw) (fe, be) dS ((q, Push , q'):dE) [] =
 summarize (δ, δ') (fw', bw') (fe', be') dS' dE'' dH' where
  (dE', dH') = addPush (fw, bw) (fe, be) (δ, δ') (q, Push , q')
  dE'' = dE' ++ dE''
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Push , q')]
  bw' = bw ⊔ [q' ==> set (Push , q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ, δ') (fw, bw) (fe, be) dS ((q, Pop , q'):dE) [] =
 summarize (δ, δ') (fw', bw') (fe', be') dS' dE'' dH' where
  (dE', dH') = addPop (fw, bw) (fe, be) (δ, δ') (q, Pop , q')
  dE'' = dE ++ dE'
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Pop , q')]
  bw' = bw ⊔ [q' ==> set (Pop , q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ, δ') (fw, bw) (fe, be) dS ((q, Unch, q'):dE) [] =
 summarize (δ, δ') (fw', bw') (fe', be') dS' dE [(q, q')] where
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Unch, q')]
  bw' = bw ⊔ [q' ==> set (Unch, q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ, δ') (fw, bw) (fe, be) dS dE ((q, q'):dH)
 | (q, q') `isin` fe  = summarize (δ, δ') (fw, bw) (fe, be) dS dE dH
summarize (δ, δ') (fw, bw) (fe, be) dS dE ((q, q'):dH) =
 summarize (δ, δ') (fw, bw) (fe', be') dS dE' dH' where
   (dE', dH') = addEmpty (fw, bw) (fe, be) (δ, δ') (q, q')
   fe' = fe ⊔ [q  ==> set q ]
   be' = fe ⊔ [q' ==> set q']
```

Figure 3.2. A Haskell implementation of pushdown control-state reachability.

```
summarize :: (Ord control, Ord frame) =>
             (Delta control frame) ->
             (ETG control frame) ->
             (ECG control) ->
             [control] ->
             [Edge control frame] ->
             [EpsEdge control] ->
             (Edges control frame)
```

An explicit transition graph is an explicit encoding of the reachable subset of the transition relation. The function `summarize` takes six parameters:

1. the pushdown transition function;
2. the current explicit transition graph;
3. the current ϵ-closure graph;
4. a work-list of states to add;
5. a work-list of explicit transition edges to add; and
6. a work-list of ϵ-closure transition edges to add.

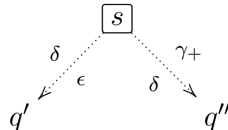
The function `summarize` processes ϵ-closure edges first, then explicit transition edges and then individual states. It *must* process ϵ-closure edges first to ensure that the ϵ-closure graph is closed when considering the implications of other edges.

3.4.4.1 Sprouting

Whenever a new state gets added to the Dyck state graph, the algorithm must check whether that state has any new edges to contribute. Both push edges and ϵ-edges do not depend on the current stack, so any such edges for a state in the pushdown system's transition function belong in the Dyck state graph. The sprout function:

$$sprout_{(Q,\Gamma,\delta)} : Q \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks whether a new state could produce any new push edges or no-change edges. Diagrammatically the sprout function's behavior is:



which means when adding control state s:

add edge $s \rightarrowtail^{\epsilon} q'$ if it exists in δ, and

add edge $s \rightarrowtail^{\gamma+} q''$ if it exists in δ.

Formally:

$$sprout_{(Q,\Gamma,\delta)}(s) = (\Delta E, \Delta H), \text{ where}$$

$$\Delta E = \left\{s \overset{\epsilon}{\rightarrowtail} q : s \overset{\epsilon}{\rightarrowtail} q \in \delta\right\} \cup \left\{s \overset{\gamma+}{\rightarrowtail} q : s \overset{\gamma+}{\rightarrowtail} q \in \delta\right\}$$
$$\Delta H = \left\{s \rightarrowtail q : s \overset{\epsilon}{\rightarrowtail} q \in \delta\right\}.$$

In Haskell:

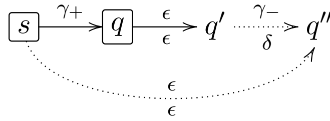
```
sprout :: (Ord control) =>
          Delta control frame ->
          control ->
          ([Edge control frame], [EpsEdge control])
sprout (δ, δ') q = (dE, dH) where
  edges = δ' q
  dE = [ (q, g, q') | (q', g) <- edges, isPush g ]
  dH = [ (q, q')    | (q', g) <- edges, isUnch g ]
```

3.4.4.2 Considering the consequences of a new push edge

Once the algorithm adds a new push edge to a Dyck state graph, there is a chance that it will enable new pop edges for the same stack frame somewhere downstream. If and when it does enable pops, it will also add new edges to the ϵ-closure graph. The *addPush* function:

$$addPush_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable states that could produce a pop. Diagrammatically this action is:



which means if adding push-edge $s \rightarrowtail^{\gamma+} q$:

if pop-edge $q' \rightarrowtail^{\gamma-} q''$ is in δ, then

add edge $q' \rightarrowtail^{\gamma_-} q''$, and

add ϵ-edge $s \rightarrowtail q''$.

Formally:

$$\begin{aligned}
&addPush_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s \overset{\gamma_+}{\rightarrowtail} q) = (\Delta E, \Delta H), \text{ where} \\
&\Delta E = \left\{ q' \overset{\gamma_-}{\rightarrowtail} q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\} \\
&\Delta H = \left\{ s \rightarrowtail q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\}.
\end{aligned}$$

In Haskell:

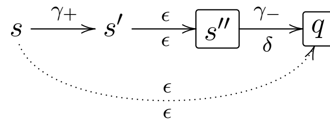
```
addPush :: (Ord control) =>
           ETG control frame ->
           ECG control ->
           Delta control frame ->
           Edge control frame ->
           ([Edge control frame], [EpsEdge control])
addPush (fw, bw) (fe, be) (δ, δ') (s, Push γ, q) = (dE, dH) where
 qset' = Set.toList $ fe!q
 dE = [ (q', g, q'') | q' <- qset', (q'', g) <- δ q' γ, isPop g ]
 dH = [ (s, q'')    | (q', Pop _, q'') <- dE ]
```

3.4.4.3 Considering the consequences of a new pop edge

Once the algorithm adds a new pop edge to a Dyck state graph, it will create at least one new ϵ-closure graph edge and possibly more by matching up with upstream pushes. The *addPop* function:

$$addPop_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable push-edges that could match this pop-edge. Diagrammatically this action is:



which means if adding pop-edge $s'' \rightarrowtail^{\gamma_-} q$:

if push-edge $s \rightarrowtail^{\gamma_+} s'$ is already in the Dyck state graph, then

add ϵ-edge $s \rightarrowtail q$.

Formally:

$$addPop_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \overset{\gamma-}{\rightarrowtail} q) = (\Delta E, \Delta H), \text{ where}$$
$$\Delta E = \emptyset \text{ and } \Delta H = \left\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s \overset{\gamma+}{\rightarrowtail} s' \in G \right\}.$$

In Haskell:

```
addPop :: (Ord control) =>
          ETG control frame ->
          ECG control ->
          Delta control frame ->
          Edge control frame ->
          ([Edge control frame], [EpsEdge control])
addPop (fw, bw) (fe, be) (δ, δ') (s'', Pop γ, q) = (dE, dH) where
 sset' = Set.toList $ be!s''
 dH = [ (s, q) | s' <- sset',
                 (g, s) <- Set.toList $ bw!s', isPush g ]
 dE = []
```

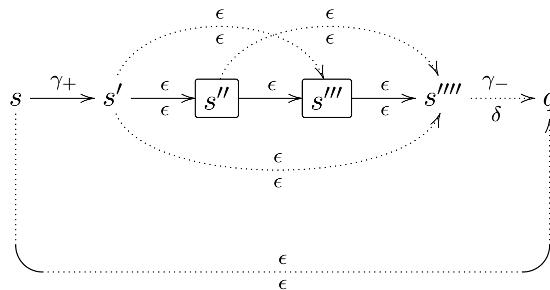
Clearly, the new edges parameter `dE` could be eliminated for the function `addPop`, but it has been retained it for stylistic symmetry.

3.4.4.4 Considering the consequences of a new ϵ-edge

Once the algorithm adds a new ϵ-closure graph edge, it may transitively have to add more ϵ-closure graph edges, and it may connect an old push to (perhaps newly enabled) pop edges. The *addEmpty* function:

$$addEmpty_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to (Q \times Q) \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for newly enabled pops and ϵ-closure graph edges: Once again, diagrammatically this action is:



which means if adding ϵ-edge $s'' \rightarrowtail s'''$:

if pop-edge $s'''' \rightarrowtail^{\gamma-} q$ is in δ, then

add ϵ-edge $s \rightarrowtail q$; and

add edge $s'''' \rightarrowtail^{\gamma-} q$;

add ϵ-edges $s' \rightarrowtail s'''$, $s'' \rightarrowtail s''''$, and $s' \rightarrowtail s''''$.

Formally:

$$
\begin{aligned}
\mathit{addEmpty}_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \rightarrowtail s''') &= (\Delta E, \Delta H), \text{ where} \\
\Delta E = \big\{ s'''' \overset{\gamma-}{\rightarrowtail} q : s' \in \overleftarrow{G}_\epsilon[s''] &\text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and} \\
& s \overset{\gamma+}{\rightarrowtail} s' \in G \big\} \\
\Delta H = \big\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] &\text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and} \\
& s \overset{\gamma+}{\rightarrowtail} s' \in G \big\} \\
& \cup \left\{ s' \rightarrowtail s''' : s' \in \overleftarrow{G}_\epsilon[s''] \right\} \\
& \cup \left\{ s'' \rightarrowtail s'''' : s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\} \\
& \cup \left\{ s' \rightarrowtail s'''' : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\}.
\end{aligned}
$$

In Haskell, the function `addEmpty` has many cases to consider:

```
addEmpty :: (Ord control) =>
            ETG control frame ->
            ECG control ->
            Delta control frame ->
            EpsEdge control ->
            ([Edge control frame], [EpsEdge control])

addEmpty (fw, bw) (fe, be) (δ, δ') (s'', s''') = (dE, dH) where
 sset'    = Set.toList $ be!s''
 sset'''' = Set.toList $ fe!s'''
 dH'   = [ (s', s'''')  | s' <- sset',  s'''' <- sset'''' ]
 dH''  = [ (s', s''')   | s' <- sset' ]
 dH''' = [ (s'', s'''') | s'''' <- sset'''' ]

 sEdges = [ (g, s) | s' <- sset',  (g, s) <- Set.toList $ bw!s' ]
```

```
dE = [ (s'''', g', q) | s'''' <- sset'''',
                        (g, s) <- sEdges,
                        isPush g, let Push γ = g,
                        (q, g') <- δ s'''' γ,
                        isPop g' ]

dH'''' = [ (s, q) | (_, s) <- sEdges, (_, _, q) <- dE ]

dH = dH' ++ dH'' ++ dH''' ++ dH''''
```

3.4.5 Termination and correctness

Because the iteration function is no longer monotonic, the existence of the fixed point must be proved. It is trivial to show that the Dyck state graph component of the iteration-space ascends monotonically with each application; that is:

Lemma 1 *Given* $M \in \mathbb{RPDS}, G \in \mathbb{DSG}$ *such that* $G \subseteq M$*, if* $\mathcal{F}'(M)(G, G_\epsilon, \Delta G) = (G', G'_\epsilon, \Delta G')$*, then* $G \subseteq G'$*.*

Since the size of the Dyck state graph is bounded by the original pushdown system M, the Dyck state graph will eventually reach a fixed point. Once the Dyck state graph reaches a fixed point, both work-graphs/sets will be empty, and the ϵ-closure graph will also stabilize. This algorithm can also be proven correct:

Theorem 4 $\text{lfp}(\mathcal{F}'(M)) = (\mathcal{DSG}(M), G_\epsilon, (\emptyset, \emptyset), \emptyset)$*.*

Proof. The proof is similar in structure to the previous one. ■

3.4.6 Complexity: Still exponential, but more efficient

As with the previous algorithm, to determine the complexity of this algorithm, there are two questions to ask: How many times would the algorithm invoke the iteration function in the worst case, and how much does each invocation cost in the worst case? The run-time of the algorithm is bounded by the size of the final Dyck state graph plus the size of the ϵ-closure graph. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. There are at most $2|\Gamma|m^2 + m^2$ edges in the Dyck state graph and at most m^2 edges in the ϵ-closure graph, which bounds the number of iterations.

Next, we must reason about the worst-case cost of adding an edge: How many edges might an individual iteration consider? In the worst case, the algorithm will consider every edge in every iteration, leading to an asymptotic time-complexity of:

$$O((2|\Gamma|m^2 + 2m^2)^2) = O(|\Gamma|^2 m^4).$$

While still high, this is a an improvement upon the previous algorithm. For sparse Dyck state graphs, this is a reasonable algorithm.

3.5 Polynomial-time complexity from widening

The previous section developed a more efficient fixed-point algorithm for computing a Dyck state graph. Even with the core improvements, the algorithm remained exponential in the worst case, owing to the fact that there could be an exponential number of reachable control states. When an abstract interpretation is intolerably complex, the standard approach for reducing complexity and accelerating convergence is widening [42]. (Of course, widening techniques trade away some precision to gain this speed.) It turns out that the small-step variants of finite-state CFAs are exponential without some sort of widening as well.

To achieve polynomial time complexity for pushdown control-flow analysis requires the same two steps as the classical case: (1) Widening the abstract interpretation to use a global, "single-threaded" store and (2) selecting a monovariant allocation function to collapse the abstract configuration-space. Widening eliminates a source of exponentiality in the size of the store; monovariance eliminates a source of exponentiality from environments. This section redevelops the pushdown control-flow analysis framework with a single-threaded store and calculates its complexity.

3.5.1 Step 1: Refactor the concrete semantics

First, consider defining the reachable states of the concrete semantics using fixed points. That is, let the system-space of the evaluation function be sets of configurations:

$$C \in \mathit{System} = \mathcal{P}\left(\mathit{Conf}\right) = \mathcal{P}\left(\mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont}\right).$$

The concrete evaluation function can be redefined to:

$$\mathcal{E}(e) = \operatorname{lfp}(f_e), \text{ where } f_e : \mathit{System} \to \mathit{System} \text{ and}$$
$$f_e(C) = \{\mathcal{I}(e)\} \cup \left\{c' : c \in C \text{ and } c \Rightarrow c'\right\}.$$

3.5.2 Step 2: Refactor the abstract semantics

The same approach can be taken with the abstract evaluation function, first redefining the abstract system-space:

$$\begin{aligned}\hat{C} \in \widehat{System} &= \mathcal{P}\left(\widehat{Conf}\right) \\ &= \mathcal{P}\left(\mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont}\right),\end{aligned}$$

and then the abstract evaluation function:

$$\begin{aligned}\hat{\mathcal{E}}(e) &= \mathrm{lfp}(\hat{f}_e), \text{ where } \hat{f}_e : \widehat{System} \to \widehat{System} \text{ and} \\ \hat{f}_e(\hat{C}) &= \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{\hat{c}' : \hat{c} \in \hat{C} \text{ and } \hat{c} \leadsto \hat{c}'\right\}.\end{aligned}$$

What we would like to do is shrink the abstract system-space with a refactoring that corresponds to a widening.

3.5.3 Step 3: Single-thread the abstract store

A set of abstract stores $\{\hat{\sigma}_1, \ldots, \hat{\sigma}_n\}$ can be approximated with the least-upper-bound of those stores: $\hat{\sigma}_1 \sqcup \cdots \sqcup \hat{\sigma}_n$. This approximation can be exploited by creating a new abstract system space in which the store is factored out of every configuration. Thus, the system-space contains a set of *partial configurations* and a single global store:

$$\begin{aligned}\widehat{System}' &= \mathcal{P}\left(\widehat{PConf}\right) \times \widehat{Store} \\ \hat{\pi} \in \widehat{PConf} &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Kont}.\end{aligned}$$

The store can be factored out of the abstract transition relation as well, so that $(\overset{\hat{\sigma}}{\rightarrowtriangle}) \subseteq \widehat{PConf} \times (\widehat{PConf} \times \widehat{Store})$:

$$(e, \hat{\rho}, \hat{\kappa}) \overset{\hat{\sigma}}{\rightarrowtriangle} ((e', \hat{\rho}', \hat{\kappa}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}'),$$

which gives us a new iteration function, $\hat{f}'_e : \widehat{System}' \to \widehat{System}'$,

$$\begin{aligned}\hat{f}'_e(\hat{P}, \hat{\sigma}) &= (\hat{P}', \hat{\sigma}'), \text{ where} \\ \hat{P}' &= \left\{\hat{\pi}' : \hat{\pi} \overset{\hat{\sigma}}{\rightarrowtriangle} (\hat{\pi}', \hat{\sigma}'')\right\} \cup \{\hat{\pi}_0\} \\ \hat{\sigma}' &= \bigsqcup \left\{\hat{\sigma}'' : \hat{\pi} \overset{\hat{\sigma}}{\rightarrowtriangle} (\hat{\pi}', \hat{\sigma}'')\right\} \\ (\hat{\pi}_0, \langle\rangle) &= \hat{\mathcal{I}}(e).\end{aligned}$$

3.5.4 Step 4: Dyck state control-flow graphs

Following the earlier Dyck state graph reformulation of the pushdown system, the set of partial configurations can be reformulated as a *Dyck state control-flow graph*. A *Dyck state control-flow graph* is a frame-action-labeled graph over partial control states, and a *partial control state* is an expression paired with an environment:

$$\begin{aligned}
\widehat{System}'' &= \widehat{DSCFG} \times \widehat{Store} \\
\widehat{DSCFG} &= \mathcal{P}(\widehat{PState}) \times \mathcal{P}(\widehat{PState} \times \widehat{Frame}_{\pm} \times \widehat{PState}) \\
\hat{\psi} \in \widehat{PState} &= \mathsf{Exp} \times \widehat{Env}.
\end{aligned}$$

In a Dyck state control-flow graph, the partial control states are partial configurations which have dropped the continuation component; the continuations are encoded as paths through the graph.

If we wanted to do so, we could define a new monotonic iteration function analogous to the simple fixed-point formulation of Section 3.3:

$$\hat{f}_e : \widehat{System}'' \to \widehat{System}'',$$

again using CFL-reachability to add pop edges at each step.

3.5.4.1 A preliminary analysis of complexity

Even without defining the system-space iteration function, we can ask, *How many iterations will it take to reach a fixed point in the worst case?* This question is really asking, *How many edges can we add?* Also, *How many entries are there in the store?* Summing these together, we arrive at the worst-case number of iterations:

$$\overbrace{|\widehat{PState}| \times |\widehat{Frame}_{\pm}| \times |\widehat{PState}|}^{\text{DSCFG edges}} + \overbrace{|\widehat{Addr}| \times |\widehat{Clo}|}^{\text{store entries}}.$$

With a monovariant allocation scheme that eliminates abstract environments, the number of iterations ultimately reduces to:

$$|\mathsf{Exp}| \times (2|\widehat{\mathsf{Var}}| + 1) \times |\mathsf{Exp}| + |\mathsf{Var}| \times |\mathsf{Lam}|,$$

which means that, in the worst case, the algorithm makes a cubic number of iterations with respect to the size of the input program.[3]

[3] In computing the number of frames, note that in every continuation, the variable and the expression uniquely determine each other based on the let-expression from which they both came. As a result, the number of abstract frames available in a monovariant analysis is bounded by both the number of variables and the number of expressions, i.e., $|\widehat{Frame}| = |\mathsf{Var}|$.

The worst-case cost of each iteration would be dominated by a CFL-reachability calculation, which, in the worst case, must consider every state and every edge:

$$O(|\mathsf{Var}|^3 \times |\mathsf{Exp}|^3).$$

Thus, each iteration takes $O(n^6)$ and there are a maximum of $O(n^3)$ iterations, where n is the size of the program. So, total complexity would be $O(n^9)$ for a monovariant pushdown control-flow analysis with this scheme, where n is again the size of the program. Although this algorithm is polynomial-time, we can do better.

3.5.5 Step 5: Reintroduce ϵ-closure graphs

Replicating the evolution from Section 3.4 for this store-widened analysis, we arrive at a more efficient polynomial-time analysis. An ϵ-closure graph in this setting is a set of pairs of store-less, continuation-less partial states:

$$\widehat{ECG} = \mathcal{P}\left(\widehat{PState} \times \widehat{PState}\right).$$

Then, we can set the system space to include ϵ-closure graphs:

$$\widehat{System}''' = \widehat{DSG} \times \widehat{ECG} \times \widehat{Store}.$$

Before we redefine the iteration function, we need another factored transition relation. The stack- and action-factored transition relation $(\rightharpoonup_g^{\hat{\sigma}}) \subseteq \widehat{PState} \times \widehat{PState} \times Store$ determines if a transition is possible under the specified store and stack-action:

$$\begin{aligned}
(e, \hat{\rho}) \xrightarrow[\hat{\phi}_+]{\hat{\sigma}} ((e', \hat{\rho}'), \hat{\sigma}') &\text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa}) \\
(e, \hat{\rho}) \xrightarrow[\hat{\phi}_-]{\hat{\sigma}} ((e', \hat{\rho}'), \hat{\sigma}') &\text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}') \\
(e, \hat{\rho}) \xrightarrow[\epsilon]{\hat{\sigma}} ((e', \hat{\rho}'), \hat{\sigma}') &\text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}').
\end{aligned}$$

Now, we can redefine the iteration function (Figure 3.3).

Theorem 5 *Pushdown 0CFA can be computed in $O(n^6)$-time, where n is the size of the program.*

Proof. As before, the maximum number of iterations is cubic in the size of the program for a monovariant analysis. Fortunately, the cost of each iteration is also now bounded by the number of edges in the graph, which is also cubic. ■

$$\hat{f}((\hat{P}, \hat{E}), \hat{H}, \hat{\sigma}) = ((\hat{P}', \hat{E}'), \hat{H}', \hat{\sigma}''), \text{ where}$$

$$\begin{aligned}
\hat{T}_{+} &= \left\{ (\hat{\psi} \overset{\hat{\phi}_{+}}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\hat{\phi}_{+}}{\overset{\hat{\sigma}}{\rightharpoonup}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_{\epsilon} &= \left\{ (\hat{\psi} \overset{\epsilon}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\epsilon}{\overset{\hat{\sigma}}{\rightharpoonup}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_{-} &= \{ (\hat{\psi}'' \overset{\hat{\phi}_{-}}{\rightarrowtail} \hat{\psi}''', \hat{\sigma}') : \hat{\psi}'' \underset{\hat{\phi}_{-}}{\overset{\hat{\sigma}}{\rightharpoonup}} (\hat{\psi}''', \hat{\sigma}') \text{ and} \\
&\qquad\qquad \hat{\psi} \overset{\hat{\phi}_{+}}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and} \\
&\qquad\qquad \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \} \\
\hat{T}' &= \hat{T}_{+} \cup \hat{T}_{\epsilon} \cup \hat{T}_{-} \\
\hat{E}' &= \left\{ \hat{e} : (\hat{e}, _) \in \hat{T}' \right\} \\
\hat{\sigma}'' &= \bigsqcup \left\{ \hat{\sigma}' : (_, \hat{\sigma}') \in \hat{T}' \right\} \\
\hat{H}_{\epsilon} &= \left\{ \hat{\psi} \rightarrowtail \hat{\psi}'' : \hat{\psi} \rightarrowtail \hat{\psi}' \in \hat{H} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \right\} \\
\hat{H}_{+-} &= \{ \hat{\psi} \rightarrowtail \hat{\psi}''' : \hat{\psi} \overset{\hat{\phi}_{+}}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \\
&\qquad \text{and } \hat{\psi}'' \overset{\hat{\phi}_{-}}{\rightarrowtail} \hat{\psi}''' \in \hat{E} \} \\
\hat{H}' &= \hat{H}_{\epsilon} \cup \hat{H}_{+-} \\
\hat{P}' &= \hat{P} \cup \left\{ \hat{\psi}' : \hat{\psi} \overset{g}{\rightarrowtail} \hat{\psi}' \right\}.
\end{aligned}$$

Figure 3.3. An iteration function for PDCFA with a single-threaded store.

3.6 Summary

This chapter defines three different algorithms for computing pushdown control-flow analysis. The first algorithm finds the least fixed point of a monotonic function (in Section 3.3). The second algorithm uses work-lists and ϵ-graphs (in Section 3.4). This second algorithm is more efficient than the first algorithm but still has exponential-time complexity. The third algorithm uses widening to sacrifice some precision for the sake of having polynomial-time complexity (in Section 3.5).

These algorithms for pushdown control-flow analysis have three advantages over CFA2 [135]: First, the analysis here does not obscure the stack and stack frames inside a summarization technique, as CFA2 does, but handles pushes and pops of stack frames explicitly. While CFA2 approximates its input as a pushdown system, this pushdown system is not explicit and therefore can be harder to understand (depending on one's familiarity with the summarization technique). Second, the abstract allocation function is a parameter to this analysis (unlike CFA2), which allows for polyvariance (multiple abstract addresses for each variable) in the analysis. Third, because this analysis handles the stack explicitly, the analysis is able to examine the stacks that may be paired with any given control state. As the next chapter describes, abstract garbage collection [98] is able to examine the stack to find reachable addresses and so is able to be combined with pushdown analysis. Because of CFA2's summarization technique, CFA2 cannot be combined with abstract garbage collection.

CHAPTER 4

ABSTRACT GARBAGE COLLECTION AND INTROSPECTIVE PUSHDOWN CONTROL-FLOW ANALYSIS

4.1 Introduction

Abstract garbage collection [98] yields large improvements in precision by using the abstract interpretation of garbage collection to make more efficient use of the finite address space available during analysis. Because of the way abstract garbage collection operates, it grants exact precision to the flow analysis of variables whose bindings die between invocations of the same abstract context. Because pushdown analysis grants exact precision in tracking return-flow, it is clearly advantageous to combine these techniques. Unfortunately, abstract garbage collection breaks the pushdown model by requiring a full traversal of the stack to discover the root set.

Section 4.2 defines abstract garbage collection, as originally presented by Might et al. [98]. Section 4.3 constrains the set of pushdown systems we consider to monotonic introspective pushdown systems. Section 4.3 also describes how abstract garbage collection works with a monotonic introspective pushdown system. Section 4.4 defines control state reachability and describes how to incorporate abstract garbage collection into least fixed point algorithm from Section 3.3. Finally for this chapter, Section 4.5 discusses combining abstract garbage collection with the work-list approach of Section 3.4.

4.2 Introspection for abstract garbage collection

Abstract garbage collection modifies the transition relation to conduct a "stop-and-copy" garbage collection before each transition. To do this, I define a garbage collection function $\hat{G} : \widehat{Conf} \to \widehat{Conf}$ on configurations:

$$\hat{G}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = (e, \hat{\rho}, \hat{\sigma}|\mathit{Reachable}(\hat{c}), \hat{\kappa}),$$

where the pipe operation $f|S$ yields the function f, but with inputs not in the set S mapped to bottom—the empty set. The reachability function $\mathit{Reachable} : \widehat{Conf} \to \mathcal{P}(\widehat{Addr})$ first

computes the root set, and then the transitive closure of an address-to-address adjacency relation:

$$\mathit{Reachable}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = \left\{ \hat{a} : \hat{a}_0 \in \mathit{Root}(\hat{c}) \text{ and } \hat{a}_0 \underset{\hat{\sigma}}{\overset{*}{\rightharpoonup}} \hat{a} \right\},$$

where the function $\mathit{Root} : \widehat{\mathit{Conf}} \to \mathcal{P}(\widehat{\mathit{Addr}})$ finds the root addresses:

$$\mathit{Root}(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) = \mathit{range}(\hat{\rho}) \cup \mathit{StackRoot}(\hat{\kappa}),$$

and the $\mathit{StackRoot} : \widehat{\mathit{Kont}} \to \mathcal{P}(\widehat{\mathit{Addr}})$ function finds roots down the stack:

$$\mathit{StackRoot}\langle(v_1, e_1, \hat{\rho}_1), \ldots, (v_n, e_n, \hat{\rho}_n)\rangle = \bigcup_i \mathit{range}(\hat{\rho}_i),$$

and the relation $(\rightharpoonup) \subseteq \widehat{\mathit{Addr}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Addr}}$ connects adjacent addresses:

$$\hat{a} \underset{\hat{\sigma}}{\rightharpoonup} \hat{a}' \text{ iff there exists } (\mathit{lam}, \hat{\rho}) \in \hat{\sigma}(\hat{a}) \text{ such that } \hat{a}' \in \mathit{range}(\hat{\rho}).$$

The new abstract transition relation is thus the composition of abstract garbage collection with the old transition relation:

$$(\leadsto_{\mathrm{GC}}) = (\leadsto) \circ \hat{G}.$$

In the formulation of pushdown systems, the transition relation is restricted to looking at the top frame, and even in less restricted formulations, at most a bounded number of frames can be inspected. Thus, the relation $(\leadsto_{\mathrm{GC}})$ cannot be computed as a straightforward pushdown analysis using summarization.

To accommodate the richer structure of the relation $(\leadsto_{\mathrm{GC}})$, I now define *introspective* pushdown systems. Once defined, one can embed the garbage-collecting abstract interpretation within this framework, and then focus on developing a control-state reachability algorithm for these systems.

An *introspective pushdown system* is a quadruple $M = (Q, \Gamma, \delta, q_0)$:

1. Q is a finite set of control states;
2. Γ is a stack alphabet;
3. $\delta \subseteq Q \times \Gamma^* \times \Gamma_\pm \times Q$ is a transition relation; and
4. q_0 is a distinguished root control state.

The second component in the transition relation is a realizable stack at the given control-state. This realizable stack distinguishes an introspective pushdown system from a general pushdown system. $\mathbb{IPDS}$ denotes the class of all introspective pushdown systems.

Determining how (or if) a control state q transitions to a control state q', requires knowing a path taken to the state q. Thus, reachability needs to be defined inductively. When $M = (Q, \Gamma, \delta, q_0)$, transition from the initial control state considers only empty stacks:

$$q_0 \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} q \text{ iff } (q_0, \langle\rangle, g, q) \in \delta.$$

For nonroot states, the paths to that state matter, since they determine the stacks realizable with that state:

$$q \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ iff there exists } \vec{g} \text{ such that } q_0 \underset{M}{\overset{\vec{g}}{\longmapsto\!\!\!\!\rightarrow}} q \text{ and } (q, [\vec{g}], g, q') \in \delta,$$
$$\text{where } q \underset{M}{\overset{\langle g_1,\ldots,g_n\rangle}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ iff } q \underset{M}{\overset{g_1}{\longmapsto\!\!\!\!\rightarrow}} q_1 \underset{M}{\overset{g_2}{\longmapsto\!\!\!\!\rightarrow}} \cdots \underset{M}{\overset{g_n}{\longmapsto\!\!\!\!\rightarrow}} q'.$$

4.2.1 Garbage collection in monotonic introspective pushdown systems

To convert the garbage-collecting, abstracted CESK machine into an introspective pushdown system, I define the function $\widehat{\mathcal{IPDS}} : \mathsf{Exp} \to \mathbb{IPDS}$:

$$\begin{aligned}
\widehat{\mathcal{IPDS}}(e) &= (Q, \Gamma, \delta, q_0) \\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\
\Gamma &= \widehat{Frame} \\
(q, \hat{\kappa}, \epsilon, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \\
(q, \hat{\phi} : \hat{\kappa}, \hat{\phi}_-, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \\
(q, \hat{\kappa}, \hat{\phi}_+, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\phi} : \hat{\kappa}) \\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e).
\end{aligned}$$

4.3 Problem: Reachability for introspective pushdown systems is uncomputable

As currently formulated, computing control-state reachability for introspective pushdown systems is uncomputable. The problem is that the transition relation expects to enumerate every possible stack for every control point at every transition. Because there are an unbounded number of stacks at each control point, it is impossible to know (without peering into the otherwise-opaque contents of the transition relation) all the successors.

To make introspective pushdown systems computable, introspective pushdown systems must operate on *sets* of stacks and must include a monotonicity constraint.

A *monotonic introspective pushdown system* is a quadruple $M = (Q, \Gamma, \delta, q_0)$:

1. Q is a finite set of control states;

2. Γ is a stack alphabet;

3. $\delta \subseteq Q \times \mathcal{P}(\Gamma^*) \times \Gamma_\pm \times Q$ is a monotonic transition relation; and

4. q_0 is a distinguished root control state.

The monotonicity constraint on the transition relations guarantees that as the set of stacks grows larger, no previously included transitions will suddenly be excluded:

$$T \subseteq T' \text{ and } \delta(q, T, g, q') \text{ implies } \delta(q, T', g, q').$$

4.3.1 Garbage collection in monotonic introspective pushdown systems

Of course, abstract garabge collection must be adapted to this refined framework. To convert the garbage-collecting, abstracted CESK machine into a monotonic introspective pushdown system, I define the function $\widehat{\mathcal{IPDS}}' : \mathsf{Exp} \to \mathbb{IPDS}$:

$$\begin{aligned}
\widehat{\mathcal{IPDS}}'(e) &= (Q, \Gamma, \delta, q_0) \\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\
\Gamma &= \widehat{Frame} \\
(q, \hat{K}, \epsilon, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for any } \hat{\kappa} \in \hat{K} \\
(q, \hat{K}, \hat{\phi}_-, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for any } \hat{\phi} : \hat{\kappa} \in \hat{K} \\
(q, \hat{K}, \hat{\phi}_+, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\phi} : \hat{\kappa}) \text{ for any } \hat{\kappa} \in \hat{K} \\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e).
\end{aligned}$$

Assuming we can overcome the difficulty of computing with an opaquely represented set of stacks, we can already see that this control-state reachability with garbage collection in this formulation should be computable: Garbage collection operates on sets of frames, and for any given control point there a finite number of sets of sets of frames.

The last challenge to consider before delving into the mechanics of computing reachable control states is *how* to represent the sets of stacks that may be paired with each control state. Fortunately, a regular language can describe the set of stacks at a control point, *and*, fortuitously, this regular language is already encoded in the structure of the Dyck state graph that is accumulated while computing reachable control states.

4.4 Computing reachability for monotonic introspective pushdown systems

Having defined monotonic introspective pushdown systems and embedded our abstract, garbage-collecting semantics within them, I am ready to define control-state reachability.

As with ordinary pushdown systems, the reachability algorithm for introspective pushdown systems is cast as finding a fixed-point, which incrementally accretes the reachable control states into a Dyck state graph.

The goal here is to compile an implicitly-defined introspective pushdown system into an explicited-constructed Dyck state graph. During this transformation, the per-state path considerations of an introspective pushdown are "baked into" the Dyck state graph. This compilation process is formulated as a map, $\mathcal{DSG} : \mathbb{IPDS} \to \mathbb{DSG}$.

Given an introspective pushdown system $M = (Q, \Gamma, \delta, q_0)$, its equivalent Dyck state graph is $\mathcal{DSG}(M) = (S, \Gamma, E, q_0)$, where $s_0 = q_0$, the set S contains reachable nodes:

$$S = \left\{ q : q_0 \underset{M}{\overset{\vec{g}}{\longmapsto\!\!\!\!\twoheadrightarrow}} q \text{ for some stack-action sequence } \vec{g} \right\},$$

and the set E contains reachable edges:

$$E = \left\{ q \overset{g}{\rightarrowtail} q' : q \underset{M}{\overset{g}{\longmapsto\!\!\!\!\twoheadrightarrow}} q' \right\}.$$

Now the goal is to find a method for computing a Dyck state graph from an introspective pushdown system.

4.4.1 Compiling to Dyck state graphs

We now turn our attention to compiling a monotonic introspective pushdown system (defined implicitly) into a Dyck state graph (defined explicitly). That is, we want an implementation of the function $\mathcal{DSG}$. As with ordinary pushdown systems, first the Dyck state graph construction is defined as the least fixed point of a monotonic function. This formulation provides a straightforward iterative method for computing the function $\mathcal{DSG}$.

The function $\mathcal{F} : \mathbb{IPDS} \to (\mathbb{DSG} \to \mathbb{DSG})$ generates the monotonic iteration function we need:

$$
\begin{aligned}
\mathcal{F}(M) &= f, \text{ where} \\
M &= (Q, \Gamma, \delta, q_0) \\
f(S, \Gamma, E, s_0) &= (S', \Gamma, E', s_0), \text{ where} \\
S' &= S \cup \left\{ s' : s \in S \text{ and } s \underset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} s' \right\} \cup \{s_0\} \\
E' &= E \cup \left\{ s \overset{g}{\rightarrowtail} s' : s \in S \text{ and } s \underset{M}{\overset{g}{\longmapsto\!\!\!\!\twoheadrightarrow}} s' \right\}.
\end{aligned}
$$

Given an introspective pushdown system M, each application of the function $\mathcal{F}(M)$ accretes new edges at the frontier of the Dyck state graph.

4.4.2 Computing a round of $\mathcal{F}$

The formalism obscures an important detail in the computation of an iteration: The transition relation ($\longmapsto\!\!\!\!\!\rightarrow$) for the introspective pushdown system must compute all possible stacks in determining whether or not there exists a transition. Fortunately, this is not as onerous as it seems: The set of all possible stacks for any given control-point is a regular language, and the finite automaton that encodes this language can be lifted (or read off) the structure of the Dyck state graph. The function $\mathit{Stacks} : \mathbb{DSG} \to S \to \mathbb{NFA}$ performs exactly this extraction:

$$\mathit{Stacks}(\overbrace{S, \Gamma, E, s_0}^{M})(s) = (S, \Gamma, \delta, s_0, \{s\}), \text{ where}$$
$$(s', \gamma, s'') \in \delta \text{ if } (s', \gamma_+, s'') \in E$$
$$(s', \epsilon, s'') \in \delta \text{ if } s' \overset{\vec{g}}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} s'' \text{ and } [\vec{g}] = \epsilon.$$

Figure 4.1 renders this DSG to NFA converter in Haskell.

The Haskell code must also change the definition of transition functions to accept an NFA describing all stacks:

```
type IDelta control frame =
  (ITopDelta control frame, INopDelta control frame)

type ITopDelta control frame =
  control ->
  NFA control frame ->
  frame ->
  [(control, StackAct frame)]

type INopDelta control frame =
  control ->
  NFA control frame ->
  [(control, StackAct frame)]
```

This means that any function that invokes the pushdown transition relation must have access to either an NFA describing all stacks for the initial point, or it must have enough information to compute it. In the original formulation, there are four functions that require this—`sprout`, `addPush`, `addPop` and `addEmpty`:

```
stacks :: (Ord control, Ord frame) =>
          ETG control frame ->
          ECG control ->
          control ->
          control ->
          NFA control frame
stacks (fw, bw) (fe, be) s0 s' = (f'', b'', s0, s') where

 fedges = [ (q, Set.fromAscList trans') |
            (q, trans) <- Map.toAscList fw,
            let trans' = [ (Just c, q') |
                           (g, q') <- Set.toAscList $ trans,
                           isPush g,
                           let c = frame g ] ]

 fedges' = [ (q, Set.fromAscList trans') |
             (q, trans) <- Map.toAscList fe,
             let trans' = [ (Nothing, q') |
                            q' <- Set.toList $ trans ] ]

 f  = Map.fromAscList fedges
 f' = Map.fromAscList fedges'
 f'' = Map.union f f'

 bedges = [ (q, Set.fromAscList trans') |
            (q, trans) <- Map.toAscList bw,
            let trans' = [ (Just c, q') |
                           (g, q') <- Set.toAscList $ trans,
                           isPush g,
                           let c = frame g ] ]

 bedges' = [ (q, Set.fromAscList trans') |
             (q, trans) <- Map.toAscList be,
             let trans' = [ (Nothing, q') |
                            q' <- Set.toList $ trans ] ]

 b  = Map.fromAscList bedges
 b' = Map.fromAscList bedges'
 b'' = Map.union b b'
```

Figure 4.1. Haskell conversion from Dyck state graphs to NFAs.

```
isprout :: (Ord control) =>
           IDelta control frame ->
           control ->
           NFA control frame ->
           ([Edge control frame], [EpsEdge control])

addIPush :: (Ord control, Ord frame) =>
            control ->
            ETG control frame ->
            ECG control ->
            IDelta control frame ->
            Edge control frame ->
            ([Edge control frame], [EpsEdge control])

addIPop :: (Ord control, Ord frame) =>
           control ->
           ETG control frame ->
           ECG control ->
           IDelta control frame ->
           Edge control frame ->

           ([Edge control frame], [EpsEdge control])

addIEmpty :: (Ord control, Ord frame) =>
             control ->
             ETG control frame ->
             ECG control ->
             IDelta control frame ->
             EpsEdge control ->
             ([Edge control frame], [EpsEdge control])
```

For sprouting, the exact NFA is passed, since we are concerned with only one control state. For the remainder, the initial control state is passed, so that it may compute the NFA as necessary.

4.4.3 Correctness

Once the algorithm reaches a fixed point, the Dyck state graph is complete:

Theorem 6 $\mathcal{DSG}(M) = \text{lfp}(\mathcal{F}(M))$.

Proof. Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathcal{F}(M)$. Observe that $\text{lfp}(f) = f^n(\emptyset, \Gamma, \emptyset, q_0)$ for some n. When $N \subseteq M$, then it easy to show that $f(N) \subseteq M$. Hence, $\mathcal{DSG}(M) \supseteq \text{lfp}(\mathcal{F}(M))$.

To show $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathcal{DSG}(M)$ that is not in $\text{lfp}(\mathcal{F}(M))$. By the defintion of $\mathcal{DSG}(M)$, each edge must be part of a sequence of edges from the initial state. Let (s, g, s') be the first edge in its sequence from the initial state that is not in $\text{lfp}(\mathcal{F}(M)$. Because the proceeding edge is in $\text{lfp}(\mathcal{F}(M))$, the state s *is* in $\text{lfp}(\mathcal{F}(M))$. Let m be the lowest natural number such that s appears in $f^m(M)$. By the definition of f, this edge must appear in $f^{m+1}(M)$, which means it must also appear in $\text{lfp}(\mathcal{F}(M))$, which is a contradiction. Hence, $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$. ■

4.4.4 Simplifying garbage collection in introspective pushdown systems

Because monotonic introspective pushdown systems consider all possible stacks at every control point, they may provide more path-sensitivity than necessary for the intended application. This is certainly the case with abstract garbage collection.

At the very least, abstract garbage collection is indifferent to the ordering of stack frames: Sets of frames suffice. Even then, an analyzer need not consider all sets of sets of frames. For instance, if the regular expression $(A|B)C^*$ describes all the stacks at a given control point, then the sets of possible frames are $\{A\}$, $\{B\}$, $\{A, C\}$ and $\{B, C\}$. Because $\{A\} \subseteq \{A, C\}$ and $\{B\} \subseteq \{B, C\}$, the monotonicity of the abstract transition relation guarantees that any states considered by collecting $\{A\}$ or $\{B\}$ would be covered by states considered when collecting $\{A, C\}$ or $\{B, C\}$. This simplification comes at no cost to precision.

If one is willing to sacrifice precision, then one can union all of the sets of reachable frames together when garbage collecting at a given control point. In the prior example, it would mean considering garbage collection with only the set of frames $\{A, B, C\}$.

4.5 An algorithm for introspective pushdown analysis with garbage collection

The reachability-based analysis for a pushdown system described in Section 3.3 requires two mutually-dependent pieces of information in order to add another edge:

1. The topmost frame on a stack for a given control state q. This is essential for *return* transitions, as this frame should be popped from the stack and the store and the environment of a caller should be updated respectively.

2. Whether a given control state q is reachable or not from the initial state q_0 along realizable sequences of stack actions. For example, a path from q_0 to q along edges labeled "push, pop, pop, push" is not realizable: The stack is empty after the first pop, so the second pop cannot happen—let alone the subsequent push.

These two data are enough for a classic pushdown reachability summarization to proceed one step further, and Section 3.4 presents an efficient algorithm to compute those. However, the presence of an abstract garbage collector, and the graduation to an *introspective* pushdown system, imposes the requirement for a third item of data:

3. For a given control state q, what are *all* possible frames that could happen to be *on* the stack at the moment the IPDS is in the state q?

The crucial addition to the algorithm is maintaining for each node q' in the DSG a set of ϵ-*predecessors*, i.e., nodes q, such that $q \longmapsto\!\!\!\!\rightarrow^{\vec{g}}_{M} q'$ and $[\vec{g}] = \epsilon$. In fact, only two out of three kinds of transitions can cause a change to the set of ϵ-predecessors for a particular node q: An addition of an ϵ-edge or a pop edge to the DSG.

A little reflection on ϵ-predecessors and top frames reveals a mutual dependency between these items during the construction of a DSG. Informally:

- A *top frame* for a state q can be pushed as a direct predecessor, or as a direct predecessor to an ϵ-predecessor.

- When a new ϵ-edge $q \xrightarrow{\epsilon} q'$ is added, all ϵ-predecessors of q become also ϵ-predecessors of q'. That is, ϵ-summary edges are transitive.

- When a γ_{-}-pop-edge $q \xrightarrow{\gamma_{-}} q'$ is added, new ϵ-predecessors of a state q_1 can be obtained by checking if q' is an ϵ-predecessor of q_1 and examining all existing ϵ-predecessors of q, such that γ_{+} is their possible top frame.

The third component—*all* possible frames on the stack for a state q—is straightforward to compute with ϵ-predecessors: Starting from q, trace out only the edges which are labeled ϵ (summary or otherwise) or γ_+. The frame for any action γ_+ in this trace is a possible stack action. Since these sets grow monotonically, it is easy to cache the results of the trace, and in fact, propagate incremental changes to these caches when new ϵ-summary or γ_+ nodes are introduced.

4.6 Summary

This chapter defines introspective pushdown control-flow analysis and then combines it with abstract garbage collection. Introspection provides the ability to examine the stacks available at any and every control-state. By being able to examine the stack, one can find the root set of reachable addresses for each configuration (control-state and stack). Since abstract garbage collection requires access to the root set of reachable addresses for each configuration, combining pushdown control-flow analysis and abstract garbage collection requires introspection. Pushdown control-flow analyses, such as the ones presented in Chapters 2 and 3 as well as CFA2, cannot soundly be combined with abstract garbage collection because of the lack of introspection.

The last chapter provided three algorithms for computing the pushdown control-flow analysis of Chapter 2, and this chapter discusses two of them. The third algorithm presented in Section 3.5, which uses widening, has polynomial-time complexity. The simplification of the time complexity comes from simplifying the store to a single global store. Abstract garbage collection improves performance by reducing the store associated with a state to only what is reachable in that state. In a global store, most (if not all) values are reachable from some state and thus cannot be removed. Therefore, it is generally not worth the effort to add abstract garbage collection to an analysis that has been widened to a global store.

This chapter with the two before it represent the theoretic contribution to static analysis of this dissertation. The next chapter presents the empirical results and applications of these theoretic contributions.

CHAPTER 5

PERFORMANCE AND APPLICATIONS OF INTROSPECTIVE PUSHDOWN ANALYSIS

5.1 Introduction

Unlike the preceding three chapters, this chapter focuses on empirical results and applications, rather than theory. Section 5.2 presents the empirical results from a working implementation of algorithms presented in Sections 3.4 and 4.5. Section 5.2 compares the speed and results of k-CFA and k-PDCFA ($k \in \{0, 1\}$) with and without abstract garbage collection. Both toy and real-life benchmarks are used in these tests. The toy benchmarks were designed to showcase the worst behavior of some of the analyses. As such, the toy benchmarks give an indication of how bad these analyses can be for small programs. However, these toy benchmarks do not convey how useful these analyses are on programs that are used for tasks other than testing static analyses. The real-life benchmarks do convey how useful these analyses are on real-life programs. These real-life benchmarks include an implementation of the public-key cryptography and a Scheme to Java compiler.

Section 5.3 discusses a few of the applications of introspective pushdown control-flow analysis (with or without abstract garbage collection). Escape analysis determines when it is safe to allocate a heap-allocated value on the stack, which depends upon whether the heap-allocated values outlive the stack frame which created them. Interprocedural dependence analysis determines resource usage based upon what and where procedures are called, their resource usage, and the resource usage of procedures they call.

5.2 Experimental evaluation

A fair comparison between different families of analyses should compare both precision and speed. k-CFA was implemented for a subset of R5RS Scheme and was instrumented with a possibility to optionally enable pushdown analysis, abstract garbage collection or both. The implementation source code and benchmarks are available:

`http://github.com/ilyasergey/reachability`

As expected, the fused analysis does at least as well as the best of either analysis alone in terms of singleton flow sets (a good metric for program optimizability) and better than both in some cases. Also worthy of note is the dramatic reduction in the size of the abstract transition graph for the fused analysis—even on top of the already large reductions achieved by abstract garbage collection and pushdown flow analysis individually. The size of the abstract transition graph is a good heuristic measure of the temporal reasoning ability of the analysis, e.g., its ability to support model-checking of safety and liveness properties [94].

5.2.1 Plain k-CFA vs. pushdown k-CFA

In order to exercise both well-known and newly-presented instances of CESK-based CFAs, the implementation was run on a series of *small* benchmarks exhibiting archetypal control-flow patterns (see Table 5.1). Most benchmarks are taken from the CFA literature: `mj09` is a running example from the work of Midtgaard and Jensen [91] designed to exhibit a nontrivial return-flow behavior, `eta` and `blur` test common functional idioms, mixing closures and eta-expansion, `kcfa2` and `kcfa3` are two worst-case examples extracted from Van Horn and Mairson's [134] proof of k-CFA complexity, `loop2` is an example from Might's [93, Section 13.3] dissertation that was used to demonstrate the impact of abstract GC, and `sat` is a brute-force SAT-solver with backtracking. This benchmark suite was run on a 2 Core 2.66 GHz OS X machine with 8 Gb RAM.

5.2.1.1 Comparing precision

In terms of precision, the fusion of pushdown analysis and abstract garbage collection substantially cuts abstract transition graph sizes over one technique alone.

Singleton flow sets were also measured as a heuristic metric for precision. Singleton flow sets are a necessary precursor to optimizations such as flow-driven inlining, type-check elimination and constant propagation. Here again, the fused analysis prevails as the best-of- or better-than-both-worlds.

The results of these benchmarks have revalidated hypotheses about the improvements to precision granted by both pushdown analysis [135] and abstract garbage collection [93]. Table 5.1 contains the detailed results on the precision of the analysis. In order to make the comparison fair, the table reports the numbers of *control states*, which do not contain stack components and are the nodes of the constructed DSG. In the case of plain k-CFA, control states are coupled with stack pointers to obtain *configurations*, whose resulting number is significantly bigger.

Table 5.1. Analysis benchmark results for toy programs. The first three columns provide the name of a benchmark, the number of expressions and variables in the program in the ANF, respectively. For each of eight combinations of pushdown analysis, $k \in \{0, 1\}$ and garbage collection on or off, the first two columns in a group show the number of *control states* and transitions/DSG edges computed during the analysis (for both less is better). The third column presents the amount of *singleton* variables, i.e, how many variables have a single lambda flow to them (more is better). Inequalities for some results of the plain k-CFA denote the case when the analysis explored more than 10^5 *configurations* (i.e., control statce coupled with continuations) or did not finish within 30 minutes. For such cases we do not report on singleton variables.

Program	Expressions	Variables	k	k-CFA			k-PDCFA			k-CFA + GC			k-PDCFA + GC		
				States	Edges	Singletons	States	Edges	Singletons	States	Edges	Singletons	States	Edges	Singletons
`mj09`	19	8	0	83	107	4	38	38	4	36	39	4	33	32	4
			1	454	812	1	44	48	1	34	35	1	32	31	1
`eta`	21	13	0	63	74	4	32	32	6	28	27	8	28	27	8
			1	33	33	8	32	31	8	28	27	8	28	27	8
`kcfa2`	20	10	0	194	236	3	36	35	4	35	34	4	35	34	4
			1	970	1935	1	87	144	2	35	34	2	35	34	2
`kcfa3`	25	13	0	272	327	4	58	63	5	53	52	5	53	52	5
			1	> 32662	> 88548	–	1761	4046	2	53	52	2	53	52	2
`blur`	40	20	0	4686	7606	4	115	146	4	90	95	10	68	76	10
			1	123	149	10	94	101	10	76	82	10	75	81	10
`loop2`	41	14	0	149	163	7	69	73	7	43	46	7	34	35	7
			1	> 10867	> 16040	–	411	525	3	151	163	3	145	156	3
`sat`	51	23	0	3844	5547	4	545	773	4	1137	1543	4	254	317	4
			1	> 28432	> 37391	–	12828	16846	4	958	1314	5	71	73	10

The SAT-solving benchmark showed a dramatic improvement with the addition of context-sensitivity. Evaluation of the results showed that context-sensitivity provided enough fuel to eliminate most of the nondeterminism from the analysis.

5.2.1.2 Comparing speed

In the original work on CFA2, Vardoulakis and Shivers [135, Section 6] present experimental results with a remark that the running time of the analysis is proportional to the size of the reachable states. There is a similar correlation in our fused analysis, but it is not as strong or as absolute. Since most of the programs from the toy suite run for less than a second, I do not report on the absolute time. Instead, the histogram on Figure 5.1 presents normalized relative times of analyses' executions. In these benchmarks, the pure machine-style k-CFA is always significantly worse in terms of execution time than either with GC or push-down system, so the plain, nonoptimized k-CFA has been excluded from the comparison.

An earlier implementation of a garbage-collecting pushdown analysis [45] did not fully exploit the opportunities for caching ϵ-predecessors, as described in Section 4.5. This led to significant inefficiencies of the garbage-collecting analyzer with respect to the mere control-flow analyzer, even though the former one observed a smaller amount of states and in some cases found larger amounts of singleton variables. After this issue had been fixed, it became clearly visible that in all cases the GC-optimized analyzer is strictly faster than its nonoptimized pushdown counterpart.

Although caching of ϵ-predecessors and ϵ-summary edges is relatively cheap, it is not free, since maintaining the caches requires some routine machinery at each iteration of the analyzer. This explains the loss in performance of the garbage-collecting pushdown analysis with respect to the GC-optimized mere CFA.

As it follows from the plot, fused analysis is always faster than the mere pushdown analysis, and about a fifth of the time, it beats the plain CFA with garbage collection in terms of performance. When the fused analysis is slower than just a GC-optimized one, it is generally not much worse than twice as slow as the next slowest analysis. Given the already substantial reductions in analysis times provided by collection and pushdown analysis, the amortized penalty is a small and acceptable price to pay for improvements to precision.

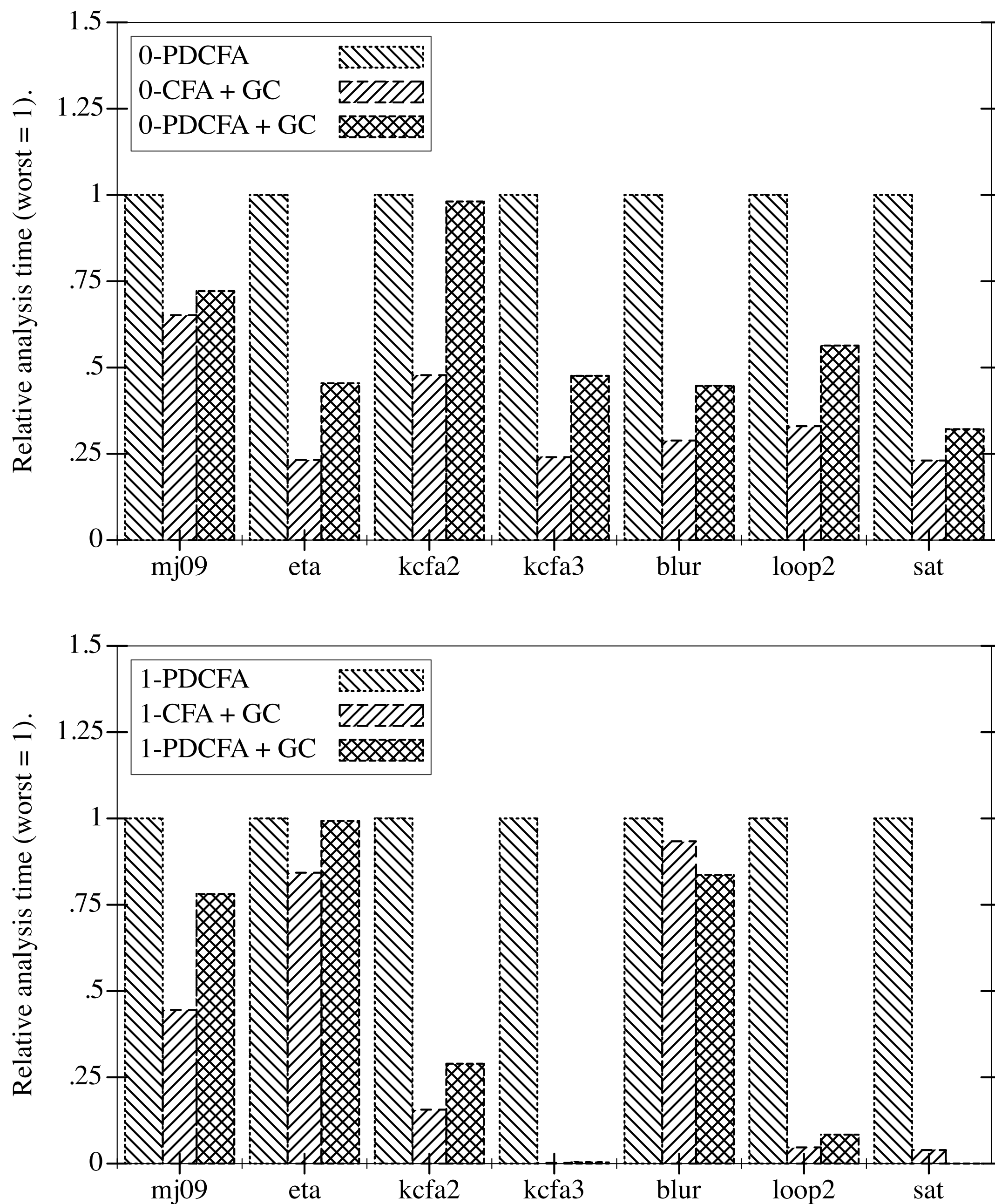


Figure 5.1. Runtime performance of various analyses. Analysis times relative to worst (= 1) in class; smaller is better. On the top is the monovariant 0CFA class of analyses, on the bottom is the polyvariant 1CFA class of analyses. (Non-GC k-CFA omitted.)

5.2.2 Analyzing real-life programs with garbage-collecting pushdown CFA

Even though this prototype implementation is just a proof of concept, this prototype implementation was run on a set of real-life programs, and not just on the suite of toy programs from the previous section, which were tailored for particular functional programming patterns. For this experiment, four programs were chosen, dealing with numeric and symbolic computations:

- `primtest` – an implementation of the probabilistic Fermat and Solovay-Strassen primality testing in Scheme for the purpose of large prime generation;
- `rsa` – an implementation of the RSA public-key cryptography;
- `regex` – an implementation of a regular expression matcher in Scheme using Brzozovski derivatives [95, 106]; and
- `scm2java` – a Scheme to Java compiler.

This benchmark suite was run on a 2.7 GHz Intel Core i7 OS X machine with 8 Gb RAM. Unfortunately, standard k-CFA timed out on most of these examples, so those tests are excluded from the comparison and focus on the effect of a pushdown analyzer only. Table 5.2 presents the results of running the benchmarks for $k = 0, 1$ with a garbage collector on and off. Surprisingly, for each of the six programs, the cases, which terminated within 30 minutes, delivered the same number of found singleton variables. However, the numbers of observed states and runtimes are indeed different in most of the cases except `scm2java`, for which all the four versions of the analysis were precise enough to actually evaluate the program. Time-wise, the results of the experiment demonstrate the general positive effect of the abstract garbage collection in a pushdown setting, which might improve the analysis performance for more than two orders of magnitude.

5.3 Applications

Pushdown control-flow analysis offers more precise control-flow analysis results than the classical finite-state CFAs. Consequently, introspective pushdown control-flow analysis improves flow-driven optimizations (e.g., constant propagation, global register allocation, inlining [121]) by eliminating more of the false positives that block their application.

The more compelling applications of pushdown control-flow analysis are those which are difficult to drive with classical control-flow analysis. Perhaps not surprisingly, the best

Table 5.2. Benchmark results for real-life programs. The first four columns provide the name of a program, the number of expressions and variables in the program in the ANF, and the number of singleton variables revealed by the analysis (same in all cases). For each of four combinations of $k \in \{0, 1\}$ and garbage collection on or off, the first two columns in a group show the number of visited control states and edges, respectively, and the third one shows absolute time of running the analysis (for both less is better). The results of the analyses are presented in minutes ($'$) or seconds ($''$), where ∞ stands for an analysis, which has been interrupted due to the an execution time greater than 30 minutes.

Program	Expressions	Variables	Singletons	$k = 0$, GC off			$k = 0$, GC on			$k = 1$, GC off			$k = 1$, GC on		
				States	Edges	Time	States	Edges	Time	States	Edges	Time	States	Edges	Time
`primtest`	155	44	16	790	955	$14''$	113	127	$1''$	>43146	>54679	∞	442	562	$13''$
`rsa`	211	93	36	1267	1507	$23''$	355	407	$6''$	20746	28895	$21'$	926	1166	$28''$
`regex`	344	150	44	943	956	$54''$	578	589	$45''$	1153	1179	$88''$	578	589	$50''$
`scm2java`	1135	460	63	376	375	$13''$	376	375	$13''$	376	375	$14''$	376	375	$13''$

examples of such analyses are escape analysis and interprocedural dependence analysis. Both of these analyses are limited by a static analyzer's ability to reason about the stack, the core competency of introspective pushdown control-flow analysis.

5.3.1 Escape analysis

In escape analysis, the objective is to determine whether a heap-allocated object is safely convertible into a stack-allocated object. In other words, the compiler is trying to figure out whether the frame in which an object is allocated outlasts the object itself. In higher-order languages, closures are candidates for escape analysis.

Determining whether all closures over a particular λ-term *lam* may be heap-allocated is straightforward: Find the control states in the Dyck state graph in which closures over *lam* are being created, then find all control states reachable from these states over only ϵ-edge and push-edge transitions. Call this set of control states the "safe" set. Now find all control states which are invoking a closure over *lam*. If any of these control states lies outside of the safe set, then stack-allocation may not be safe; if, however, all invocations lie within the safe set, then stack-allocation of the closure is safe.

5.3.2 Interprocedural dependence analysis

In interprocedural dependence analysis, the goal is to determine, for each λ-term, the set of resources which it may read or write when it is called. Might and Prabhu [96] showed that if one has knowledge of the program stack, then one can uncover interprocedural dependencies. This technique can be adapted to work with Dyck state graphs. For each control state, find the set of reachable control states along only ϵ-edges and pop-edges. The frames on the pop-edges determine the frames which could have been on the stack when in the control state. The frames that are live on the stack determine the procedures that are live on the stack. Every procedure that is live on the stack has a read-dependence on any resource being read in the control state, while every procedure that is live on the stack also has a write-dependence on any resource being written in the control state. This logic is the direct complement of "if f calls g and g accesses a, then f also accesses a."

5.4 Summary

The empirical results show that introspective pushdown control-flow analysis and abstract garbage collection are both powerful improvements over k-CFA, especially when paired together. For example, the satisfiability benchmark (sat) clearly shows the advantage of using introspective pushdown control-flow analysis and abstract garbage collec-

tion together. However, the Scheme to Java compiler benchmark (scm2java) shows that introspective pushdown control-flow analysis can perform well without abstract garbage collection.

Beyond the applications discussed in Section 5.3, introspective pushdown control-flow analysis is useful for any application that can use k-CFA, especially those that require reasoning about the stack. Introspective pushdown control-flow analysis is also useful for applications that do not specifically require reasoning about the stack if the input programs contain recursion. k-CFA, especially for low values of k, is limited in the depth of the stack that it keeps precise. Therefore k-CFA does not perform well on arbitrarily-deep recursion because the stack becomes too deep. Introspective pushdown control-flow analysis has no limit on the depth of the stack that it can keep precise, and as such avoids the problems that k-CFA has with deep stacks.

Other than Section 9.2 on the work related to introspective pushdown control-flow analysis, this chapter is the end of the static analysis portion of this dissertation. The next three chapters, along with Section 9.3, present Nebo, a declarative domain-specific language embedded in C++ for discretizing partial differential equations for transport phenomena.

CHAPTER 6

SYNTAX AND SEMANTICS OF NEBO

6.1 Introduction

This chapter and the following two chapters present Nebo: Its design, its implementation, its use, and its performance. This chapter focuses on Nebo's design, specifically its syntax and semantics. The next chapter focuses on Nebo's implementation, and Chapter 8 focuses on the use and performance of Nebo.

Nebo is a domain-specific language for numerically solving PDEs in high-performance simulations. Because Nebo's domain is so limited, Nebo's syntax and semantics are limited. Numerically solving partial differential equations does not require a Turing-complete language, and thus Nebo is not Turing-complete. Because Nebo is not Turing-complete, the syntax and semantics are almost completely limited to assignment statements. Each assignment statement in Nebo is roughly analogous to a mathematical equation over fields.

Nebo is embedded within C++ [125], and as such Nebo's syntax is restricted to C++'s syntax. Thus, standard C++ compilers parse Nebo code without modification. On one hand, Nebo is limited to C++'s basic syntax: Nebo uses C++-style function call syntax, C++-style operator syntax, and only the operators that can be overloaded within C++. On the other hand, Nebo inherited syntax, operators, and operator precedence that are well-defined, well-documented, and well-understood by C++ programmers. Furthermore, Nebo maintains the semantic meaning of the operators it overloads, lifted over fields. For example, addition of two fields represents the pointwise-addition of the elements with those two fields. The sole exception to Nebo maintaining the semantic meaning of its overloaded operators is its assignment operator, which will be discussed in Section 6.3.

Nebo carefully defines what the calculation of a specific Nebo Expression means. This calculation is the same for all valid elements in fields involved in a given Nebo Expression.[1] Nebo intentionally does not define the order in which the calculation of a Nebo Expression happens to individual elements/indices. This unspecified order is the source of Nebo's architecture-independent parallelism. Nebo is able to pass this flexibility on to the

[1] Stencil operations (Section 6.6) invalidate edge elements in well-defined ways.

underlying parallelism constructs to avoid synchronization and communication, which is discussed in detail in Section 7.4.

The full grammar for Nebo Expressions is given in Figure 6.1. The rest of this chapter is laid out as follows: Section 6.2 discusses basic Nebo Expressions. Section 6.3 discusses Nebo assignment. Section 6.4 discusses Nebo reductions. Section 6.5 discusses conditional expressions within Nebo. Finally, Section 6.6 discusses stencil operations in Nebo.

6.2 Basic Nebo Expressions

Expressions are the basic abstraction of Nebo. Unlike traditional expressions, Nebo Expressions represent calculations, not values. Nebo Expressions do produce values, but the values they produce depend on how the expressions are used. The current uses of Nebo Expressions are field assignment and reductions (see Sections 6.3 and 6.4, respectively.)

A Nebo Expression can be:

- a scalar value;
- a field;
- the valid use of supported operators and functions (below), whose arguments themselves are Nebo Expressions;
- a conditional expression, which is discussed in Section 6.5; or
- a stencil operator applied to a Nebo Expression, which is discussed in Section 6.6.

Nebo Expressions support the following operations and functions:

- algebraic operators: addition [$\bullet + \bullet$], subtraction [$\bullet - \bullet$], multiplication [$\bullet * \bullet$], division [$\bullet / \bullet$], and negation [$-\bullet$];
- trigonometric functions: sine [`sin(•)`], cosine [`cos(•)`], tangent [`tan(•)`], and hyperbolic tangent [`tanh(•)`]; and
- other mathematical functions: exponentiation with base e [`exp(•)`], exponentiation with given base [`pow(•, •)`], absolute value [`abs(•)`], square root [`sqrt(•)`], natural logarithm [`log(•)`], minimum of two values [`min(•, •)`], and maximum of two values [`max(•, •)`].

Nebo provides support for these operators and functions through operator (and function) overloading and template metaprogramming.

$$
\begin{array}{rcl}
s \in \mathit{SCALARS} & ::= & \textit{scalar values} \\
f \in \mathit{FIELDS} & ::= & \textit{field objects} \\
b \in \mathit{BOOLEANS} & ::= & \textit{boolean values} \\
\mathit{StOp} \in \mathit{STENCILS} & ::= & \textit{stencil operators} \\
e \in \mathit{EXPRESSIONS} & ::= & s \\
 & | & f \\
 & | & e_1 + e_2 \\
 & | & e_1 - e_2 \\
 & | & e_1 * e_2 \\
 & | & e_1 \ / \ e_2 \\
 & | & - \ e \\
 & | & \texttt{sin(}e\texttt{)} \\
 & | & \texttt{cos(}e\texttt{)} \\
 & | & \texttt{tan(}e\texttt{)} \\
 & | & \texttt{tanh(}e\texttt{)} \\
 & | & \texttt{exp(}e\texttt{)} \\
 & | & \texttt{pow(}e_1\texttt{, } e_2\texttt{)} \\
 & | & \texttt{abs(}e\texttt{)} \\
 & | & \texttt{sqrt(}e\texttt{)} \\
 & | & \texttt{log(}e\texttt{)} \\
 & | & \texttt{min(}e_1\texttt{, } e_2\texttt{)} \\
 & | & \texttt{max(}e_1\texttt{, } e_2\texttt{)} \\
 & | & \texttt{cond(}be_{\texttt{1}}\texttt{, } e_{\texttt{1}}\texttt{)} \ldots \texttt{(}be_{\texttt{n}}\texttt{, } e_{\texttt{n}}\texttt{)(}e_{\texttt{n+1}}\texttt{)} \\
 & | & \mathit{StOp}\texttt{(}e\texttt{)} \\
be \in \mathit{BOOLEXPR} & ::= & b \\
 & | & e_1 == e_2 \\
 & | & e_1 ! = e_2 \\
 & | & e_1 < e_2 \\
 & | & e_1 > e_2 \\
 & | & e_1 <= e_2 \\
 & | & e_1 >= e_2 \\
 & | & be_1 \ \&\& \ be_2 \\
 & | & be_1 \ || \ be_2 \\
 & | & ! \ be
\end{array}
$$

Figure 6.1. Grammar for Nebo Expressions.

6.3 Assignment

Because Nebo calculates the discretized results of partial differential equations, using Nebo Expressions in assignments keeps Nebo's syntax very close to the source equations. Field assignment is the primary use of Nebo Expressions, which is comparable to a foreach operation or Lisp's map operation. With field assignment, Nebo Expressions produce a field (array) of values, which are the values used in the assignment. Nebo uses `operator <<=` for assignment instead of `operator =` because using `operator <<=` makes it explicit when and where Nebo assignment is taking place. This assignment operator is the only operator that Nebo has changed the semantic meaning from the semantics of C++, other than lifting the operations over fields. Because Nebo's target domain does not explicitly use bitwise shifts, let alone compound bitwise shift assignments, Wasatch developers do not confuse Nebo assignment with the compound bitwise left shift assignment operation.

For a concrete example of Nebo assignment, consider the following equation, where a, b, and c are fields:

$$c = a + sin(b)$$

Without Nebo, this equation could be calculated with the following code:[2]

```
Field a, b, c;
//...
Field::iterator ic = c.begin();
Field::iterator const ec = c.end();
Field::const_iterator ia = a.begin();
Field::const_iterator ib = b.begin();
while(ic != ec) {
    *ic = *ia + sin(*ib);
    ++ic;
    ++ia;
    ++ib;
}
```

With Nebo, this same equation can be calculated by:

```
Field a, b, c;
//...
c <<= a + sin(b);
```

When not using Nebo, executable code in C++ loses the simplicity of the mathematical equations being modeled, and becomes bogged down with implementation-specific details such as iterators. When using Nebo, Wasatch developers are able to write code that focuses on what the calculation should do and not how the calculation should be done.

[2]There are many ways to calculate the results of the equations in this domain. The non-Nebo code in this chapter is based on how Wasatch developers wrote code before they started using Nebo.

6.4 Reductions

Reductions, such as sum and max (of a field), use Nebo Expressions both to simplify the implementation of the reductions themselves as well as restrict Nebo's end users to use syntax with which they are already familiar. Reductions, along with field assignment, are currently the only uses of Nebo Expressions. These reductions act much like MapReduce, where the calculation of the Nebo Expression is the map step, and the reduction operation (sum, max, etc.) is the reduce step. Thus, with reductions, Nebo Expressions produce a single scalar value. Currently, Nebo supports the following reductions directly: Finding the minimum value in a field, finding the maximum value of a field, finding the sum of the entire field, and finding the L^2 norm of a field[3].

For example, consider the following expression which returns the summation of all scalar values of the calculated field, where a, b, and c are fields:

$$\texttt{sum}(a + sin(b))$$

Without Nebo, this equation could be calculated with the following code:

```
Field a, b, c;
Scalar sum;
//...
Field::iterator ic = c.begin();
Field::iterator const ec = c.end();
Field::const_iterator ia = a.begin();
Field::const_iterator ib = b.begin();
while(ic != ec) {
    *ic = *ia + sin(*ib);
    ++ic;
    ++ia;
    ++ib;
}

sum = 0;
ic = c.begin();
while(ic != ec) {
    sum += *ic;
    ++ic;
}
```

Or by hand-fusing the loops:

```
Field a, b;
Scalar sum;
//...
sum = 0;
```

[3]Technically, finding the L^2 norm of a field is more than a simple reduction: The L^2 norm is the square root of the sum of the square of each element. Wasatch developers occasionally use the L^2 norm of fields for testing, thus it is convenient for them to have a single function they can call.

```
Field::const_iterator ia = a.begin();
Field::const_iterator const ea = a.end();
Field::const_iterator ib = b.begin();
while(ia != ea) {
    sum += *ia + sin(*ib);
    ++ia;
    ++ib;
}
```

This example is trivial to calculate the sum with a single loop. However, with some operations, such as stencils (see Section 6.6), Wasatch developers would write multiple loops before using Nebo. While hand-fusing loops is technically possible, Wasatch developers found that the hand-fused code was more complex than they were willing to maintain.

With Nebo, this same expression can be calculated by:

```
Field a, b, c;
Scalar sum;
//...
c <<= a + sin(b);
sum = nebo_sum(c);
```

Or simply just:

```
Field a, b;
Scalar sum;
//...
sum = nebo_sum(a + sin(b));
```

Unless Wasatch developers need the calculated field (c in the above example) for some purpose other than the summation, they prefer to merge the loops with Nebo (last code snippet). Avoiding single-use variables is easy with Nebo and has two distinct advantages: First, code with fewer variables and fewer loops is easier for Wasatch developers to maintain. Second, code with fewer loops has better runtime performance in general.

6.5 Conditional expressions

Because of the way C++ is defined, the predefined conditionals `if` and the ternary operator (• ? • : •) cannot be overloaded. Thus, to have point-wise conditional evaluation over fields, Nebo needed a different type of conditional expression.

We considered two main options: `WHERE`, as introduced in Fortran 90, and `cond`, as used in many functional languages (introduced by LISP). The main advantage of `WHERE` is that many Wasatch developers have experience with Fortran and therefore would not need to be trained on its use. However, the syntax of `WHERE` in Fortran does work easily with the syntax of C++. The main advantage of `cond` is that it can fit into the syntax of C++ using operator overloading and template metaprogramming, which Nebo already exploits. `cond`'s

main disadvantage is that Wasatch developers were unfamiliar with its syntax.

Ultimately, we decided to use `cond` because `cond` because fits into Nebo's familiar declarative syntax. We overcame `cond`'s disadvantage by tightly controlling the valid syntax of `cond`. With our simple syntax, we found that Wasatch developers were quickly able to learn and use `cond`.

For conditional expressions to be useful, we must have expressions that express boolean values. As most languages, Nebo uses expressions that evaluate to true or false; that is, Nebo includes boolean expressions, called Nebo Boolean Expressions. Nebo Boolean Expressions are similar to Nebo Expressions in that they represent calculations, not values. However, when Nebo Boolean Expressions are evaluated they produce boolean values rather than scalar values, which Nebo Expressions produce. A Nebo Boolean Expression can be:

- a boolean value;
- the numeric comparison of two Nebo Expressions, using any of the C++ numeric comparison operators (==, ! =, <, >, <=, and >=); or
- a logical connective of Nebo Boolean Expressions, using any of the C++ logical connective operators (&&, ||, and !).

Many functional languages that implement some form of `cond` provide support for a wide range of clause styles. In effect these languages trust and enable the end users to select the proper clause style for their current task. In these languages, if the end user does not understand how certain clause styles work and misuses them, that end user is expected to figure it out for themselves. Nebo's average end user views learning new syntax as a burden. Thus, Nebo restricts syntax to focus the user's attention on as few new syntactic forms as possible with as little surprise as possible.

With these goals in mind, Nebo's implementation of `cond` has the following restrictions: Every nonfinal clause must contain exactly two arguments, and the last clause must contain exactly one argument. The second argument of each nonfinal clause and the single argument of the final clause must be a valid Nebo Expression. The first argument of each nonfinal clause must be a Nebo Boolean Expression, which are explained above. Of course, any use of `cond` that does not follow these restrictions will cause compiler errors and will not compile.

The semantics of `cond` mimic a nested if-statement, lifted pointwise over fields. For each element, the Nebo Boolean Expression of the first clause is evaluated. If that boolean expression returns true, the second argument (a Nebo Expression) is evaluated and returned

as the result of the `cond`. If that boolean expression returns false, evaluation skips to the next clause and evaluates its boolean expression, again branching on the resulting value. If the final clause is reached, its only argument (a Nebo Expression) is evaluated and returned as the result of the conditional expression.

These semantics imply that every `cond` expression represents a well-defined and user-specified calculation. For each point, the conditional expression returns the value associated with the first true Nebo Boolean Expression. If none of the Nebo Boolean Expressions evaluate to true the conditional expression returns the value of the final clause.

Thus there is no undefined behavior or runtime errors that can surprise an end user. Furthermore, specifying only two clause styles that must be used in each `cond` expression[4] limits the syntax that end users must learn to the syntax that end users will actually use.

For a concrete example, consider the following code:

```
bool d;
Field a, b;
//...
Field::iterator ib = b.begin();
Field::iterator const eb = b.end();
Field::const_iterator ia = a.begin();
while(ib != eb) {
    if(*ia > 0.0)
        *ib *= *ia;
    else if(*ia < 0.0)
        *ib *= -(*ia);
    else if(d)
        *ib *= *ib;
    ++ib;
    ++ia;
}
```

With Nebo, this code can be rewritten:

```
bool d;
Scalar s;
Field a, b;
//...
b <<= b * cond(a > 0.0, a)
              (a < 0.0, -a)
              (d, b)
              (1.0);
```

In the above example, the code not using Nebo implicitly does not change the value of the current element of `b`, if the value of the current element of `a` is exactly zero and the double

---

[4]Technically, `cond` expressions only containing a final clause are valid, such as: `a <<= cond(b)`, where `a` and `b` are fields. However, these single-clause cases are useless and needlessly complex. The `cond` in each single-clause `cond` can be removed without changing the meaning of the expression. The previous example could be rewritten as: `a <<= b`. Thus, any practical use of `cond` will contain at least two clauses.

`d` is false. In the code that uses Nebo, this case must be explicitly handled, forcing the user to think about the final case and what should happen if all the previous cases fail.

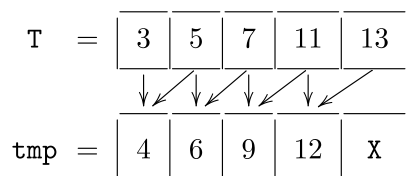
6.6 Stencil operations

Computational fluid dynamics heavily uses stencils, a nonpoint-wise array access pattern. A stencil is a fixed set of array offsets describing what array cells are needed for the current numeric computation. Some other implementations of stencils allow the stencil to change shape at runtime. However, in this domain stencil shapes are dependent upon compile-time and type information, and therefore do not change at runtime.

Consider the following 1-dimensional stencil example, which calculates a gradient of field T and uses traditional C array access (`array[•]`) for clarity:

```
int total;
Field1 T;
Field2 tmp;
//...
int cur = 0;
while(cur < total) {
    tmp[cur] = 0.5 * (T[cur] + T[cur + 1]);
    ++cur;
}
```

Graphically, with arbitrary values in field `T`, this gradient calculation looks like:



The values in field `tmp` come directly from field `T`: $0.5 * (3 + 5) = 4$; $0.5 * (5 + 7) = 6$; $0.5 * (7 + 11) = 9$; and $0.5 * (11 + 13) = 12$.

The last cell of field `tmp` does not contain a valid value because there is no further cell in field `T`. With any stencil there are edge cases that cannot be computed. There are various ways to handle such edge conditions. Uintah and Wasatch use “ghost cells,” a layer of cells surrounding the fields on all sides, so that stencils can fill the entire result field with valid results.

Using these operations before the use of Nebo was so complex that Wasatch developers developed their own abstraction prior to Nebo development and use. Each stencil operation Wasatch developers use contains 2, 3, 4, 9, or 27 points, with constant coefficients. Wasatch developers do not think about stencil calculations as stencils with definite shapes. Instead they think in terms of operations that happen to use stencils: Taking the gradient of a

field, interpolating one field type to another, taking the divergence of field, etc. So they developed an object-oriented approach where each operation is its own object, and each operation object supports an `apply_to_field` method.

For example, the gradient operation becomes:

```
Field1 T;
Field2 tmp;
//...
GradX.apply_to_field(T, tmp);
```

Unfortunately, this approach limits its application to a single stencil operation which cannot be combined with any other calculation. For example, consider applying a divergence operation to the result of gradient operation:

$$\phi = \nabla \cdot \nabla(T)$$

where ∇ is the gradient operation and $\nabla\cdot$ is the divergence operation. With Wasatch developers' syntax, this calculation becomes:

```
Field1 phi, T;
Field2 tmp;
//...
GradX.apply_to_field(T, tmp);
DivX.apply_to_field(tmp, phi);
```

(The operations are called GradX and DivX because they are gradient and divergence operations respectively, along the X-axis.) While this approach works, the syntax looks nothing like the mathematical equation, and requires two loops that cannot be fused.

To allow loop fusion, Nebo uses stencil points as the basic stencil abstraction, which fits well into Nebo's declarative structure. For example, the above gradient operation could theoretically be written in Nebo as:

```
Field1 T;
Field2 tmp;
//...
tmp <<= 0.5 * (StPt<0>(T) + StPt<1>(T));
```

Each stencil point describes a computation that has been shifted by the given offset. This approach treats each stencil point as a function that modifies the calculation by shifting its values. Thus, stencil points can be arbitrarily composed. However, using stencil points forces Wasatch developers to consider explicitly the stencil shape rather than just the operation they wish to calculate. For example, in addition to the above gradient operation, consider the divergence operation defined using stencil points:

```
Field1 phi;
```

```
Field2 tmp;
//...
phi <<= 0.5 * (StPt<-1>(tmp) + StPt<0>(tmp));
```

Performing these stencil operations in sequence becomes:

```
Field1 T, phi;
Field2 tmp;
//...
tmp <<= 0.5 * (StPt<0>(T) + StPt<1>(T));
phi <<= 0.5 * (StPt<-1>(tmp) + StPt<0>(tmp));
```

Chaining these stencil operations together into a single assignment becomes:

```
Field1 T, phi;
//...
phi <<= 0.5 * (StPt<-1>(0.5 * (StPt<0>(T) +
                                StPt<1>(T))) +
               StPt< 0>(0.5 * (StPt<0>(T) +
                                StPt<1>(T))));
```

With basic algebra and the reasoning that moving to the right (StPt<1>) and then moving to the left (StPt<-1>) is the same as not moving (StPt<0>):

```
Field1 T, phi;
//...
phi <<= 0.25 * StPt<-1>(T) +
        0.5  * StPt< 0>(T) +
        0.25 * StPt< 1>(T);
```

While the stencil point approach may be more efficient than using the object oriented approach, the stencil point approach is much more error-prone as well as being more difficult to read and to maintain.

Thus, Nebo actually provides the use of stencil operations as Nebo functions that take a Nebo Expression as an argument and return a Nebo Expression. These functions use the following syntax:

```
Field1 T, phi;
Field2 tmp;
//...
tmp <<= GradX(T);
phi <<= DivX(tmp);
```

The shape and coefficients of GradX and DivX are determined by compile-time type information. The resulting code is much easier to understand and maintain:

```
Field1 T, phi;
//...
phi <<= DivX(GradX(T));
```

Additionally, this code is about as close as one can get to the mathematical equation in

ASCII-based source files.

Furthermore, the shape of a stencil operation depends upon the types of the fields involved. Consider taking the gradient of a volume field, T, resulting in a X-face field, tmp, using the stencil operation, GradX:[5]

```
tmp <<= GradX(T);
```

There are four types of volume fields, and each volume field type has a X-face field type associated with it. Each of these pairs of types has a different interpolant operation, each with a potentially different stencil shape. Using Nebo's functional stencil operation syntax in a function templated over the field types, the above code will work with any pair of volume and X-face field types. The type system is able to determine the correct shape of the stencil operation at compile-time. Using the stencil point syntax, each pair of volume and X-face types would require a separate function, rather than having a single templated function. Section 7.3 has more information about field types and stencil operations.

This stencil syntax is another example of how we have restricted Nebo's syntax. While stencil points are not directly supported by Nebo, common stencil usage patterns are available through stencil operations. By chaining the provided stencil operations, end users can create new stencil shapes quickly and without error. While it is straightforward to add new stencil shapes to Nebo, doing so requires some knowledge of Nebo's implementation. The average Wasatch developer does not create new Nebo stencil shapes. However, once a Nebo developer (or possibly a Wasatch power-developer) creates a new stencil shape, all Wasatch developers can use it in the same way as any other Nebo stencil shape.

6.7 Summary

This chapter deliberately presents what can be done with Nebo without discussing how it is done. This distinction is the heart of Nebo's main design goal: What is to be calculated should be easy to write, easy to read, and contain no notion of how that calculation should be done. Thus, Nebo has a clean declarative syntax. While code written in Nebo is declarative and simple, the code that implements Nebo is not. The next chapter discusses the details of Nebo's implementation. In particular, the next chapter explains how the syntax of Nebo from this chapter is translated, in standard C++ compilers, into runnable C++ (and CUDA) code.

---

[5]Each element of a volume field contains a scalar value measuring a quantity of the volume of the cell at that index. Each element of a X-face field contains a scalar value measuring a quantity at a face (normal to the X-axis) of the cell at that index.

CHAPTER 7

IMPLEMENTATION OF NEBO

7.1 Introduction

This chapter presents the implementation of Nebo and is divided into three main sections: Parsing (Section 7.2), the type system (Section 7.3), and the backends (Section 7.4). This implementation can be downloaded using git from:

`git://software.crsim.utah.edu/SpatialOps.git`

Parsing is handled by a standard C++ compiler and in particular through template metaprogramming. The C++ template system is Turing-complete, and template metaprogramming is the technique of using the C++ template system to perform arbitrary computation within a standard C++ compiler, during compilation. Every Nebo Expression is parsed into a type that represents an abstract syntax tree. An abstract syntax tree represents the calculation of its originating Nebo Expression. Section 7.2 discusses how Nebo is parsed with template metaprogramming in abstract syntax trees.

Nebo uses the same types and data structures as Wasatch, and in any given assignment the types of the fields used must match. For example, when adding two fields, they must have the same type, and they return the same type. Likewise, applying sine to a Nebo Expression results in an expression of the same type as the input expression. Stencils complicate this straightforward model: Every stencil operation takes a field or Nebo Expression of a certain type and returns a Nebo Expression of a different type. Using a stencil operation, values from a field of one type can be assigned to values of a different type. Section 7.3 goes into more detail how stencil operations change types.

Section 7.4 discusses how an abstract syntax tree is converted into running code for each of Nebo's backends. Every Nebo Expression can be run on a single-core chip (Section 7.4.1), on a multicore chip (Section 7.4.2), and on a many-core/GPU chip (Section 7.4.3). The only exception to this universality of Nebo Expressions is reductions (Section 7.4.4). Currently reductions are only implemented for single-core execution.

7.2 Template metaprogramming

Nebo uses function and operator overloading in C++ to implement Nebo Expressions, which are then evaluated by Nebo assignment statements or Nebo reductions. Nebo *could* use C++ class inheritance to build Nebo Expressions from the overloaded functions and operators. This approach would use a Nebo Expression base class, from which specific operations would inherit. Operations, such as addition, which have subexpressions, would contain base class pointers to the operation's subexpressions. When evaluating a specific element, an operation would call virtual member functions on its subexpressions to determine their values for the current element. Any call to a virtual member function requires a lookup in a virtual table of function pointers. Unfortunately, this approach would require a lookup in a virtual table for every subexpression in a Nebo Expression for each element during every evaluation. These lookups are an unacceptable amount of overhead at runtime. Thus, Nebo does not use class inheritance, but instead uses C++ template metaprogramming.

The C++ template system is a completely-pure, Turing-complete functional language with burdensome syntax. There are several implementations C++ template metaprograms that serve as proofs of concept [133, 136]. While the Turing-completeness of C++ templates is interesting, it is not immediately useful. Unless the compile-time computation affects the generated code, there are more straightforward tools that can compute the same results. Since this system is part of the type system of C++, compile-time computation can remove runtime overhead, by informing the compiler about constant values, functions, and control flow paths. Thus, using the C++ template system, we can inform the compiler exactly what the subexpressions in a given Nebo Expression are and avoid using virtual tables altogether. Furthermore, since all Nebo functions are marked as inline, the compiler often can inline nested calls to subexpressions, avoiding the runtime overhead of calling these functions.

Nebo Expressions are template objects, whose template parameters are the types of the expression's subexpressions. Thus, when analyzing the function calls to subexpressions, the compiler knows the specific type and therefore the specific function that will be called to evaluate a specific element at runtime. Therefore, the type of a Nebo Expression is an abstract syntax tree (AST) of the calculation that Nebo Expression is to perform. Consider the following Nebo code:

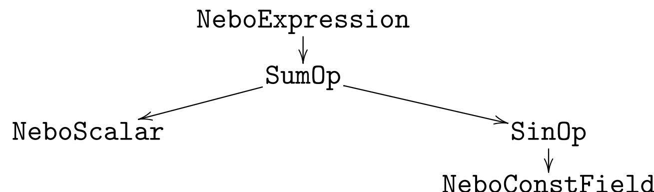
```
Field a, b;
//...
b <<= 3.14159 + sin(a);
```

Building this AST framework is rather straightforward. C++ operator overloading allows functions and operators to return any type. The `+` operator generates an object of type

SumOp<Arg1, Arg2>, and the sin function generates an object of type SinOp<Arg>. A simplified[1] version of the right-hand side's return type in the above example is:

```
NeboExpression<SumOp<NeboScalar, SinOp<NeboField> > >
```

Graphically, this AST is:



This AST model works well for simple Nebo assignment and stencils. Unlike operators and functions, Nebo conditionals do not have a fixed number of operands, complicating this AST framework. Just as functional programming provided the idea of cond, functional programming techniques provide the most simple framework: We represent each clause as its own type/object, and the entire conditional expression as a list of clauses. To build this list, we use an auxiliary structure CondBuilder that has function call, operator (), overloaded. When a clause is applied to this structure, CondBuilder returns a new version of itself with the given clause added to the list of clauses. When the final clause is applied to this structure, CondBuilder returns the Nebo Expression form of cond.

The syntax of cond and C++'s parsing method create some complications to this model. The clauses are added to the structure CondBuilder in the order they appear (and the order in which they are to be evaluated). CondBuilder adds the clauses to its list in the order that it parses them, which creates a list of clauses in the reverse order in which they need to be evaluated at runtime. When the final clause has been parsed, CondBuilder reverses its list of clauses. Fortunately, reversing a proper list in a pure system/language is straightforward.

We also use this template/type AST approach to generate different backends. Depending on available resources and run-time conditions, Nebo is able to run on a single thread, on multiple threads, or on a GPU. Each backend requires different yet related functionality to run on its target architecture. To keep each backend separate and distinct, Nebo again uses templates, or rather more template parameters. Each specific Nebo Expression has a template parameter for mode. A mode is either a backend or an intermediate step towards a backend. When a use of a Nebo Expression, such as Nebo assignment, is executed, an

[1]Information about the Field type and the mode have been left out. See Section 7.3 for information about Field type information, and see below in this same section and Section 7.4 for information about modes.

instance of the Nebo Expression AST is constructed in the `Initial` mode. Then, based on compile-time and runtime conditions, a specific backend is chosen, and the Nebo Expression AST is constructed with the appropriate mode.

Consider again the example from the beginning of this section:

```
Field a, b;
//...
b <<= 3.14159 + sin(a);
```

The type of the Nebo Expression in this assignment starts out as:

```
NeboExpression<SumOp<Initial,
                     NeboScalar<Initial>,
                     SinOp<Initial, NeboField<Initial> > > >
```

When a specific backend has been chosen, the top-level `NeboExpression` type is dropped. If the single-core/thread backend is chosen, a new instance of the AST is created in the `SeqWalk` mode (short for sequential walk). The type of this AST is:

```
SumOp<SeqWalk, NeboScalar<SeqWalk>, SinOp<SeqWalk, NeboField<SeqWalk> > >
```

There are a total of five modes: `Initial`, `SeqWalk`, `Resize`, `GPUWalk`, and `Reduction`. Each mode is a type that is declared but not defined. Thus, each mode can be used as a template argument but cannot be used as the type of a variable. For example, `Initial` declaration is simply:

```
struct Initial;
```

Currently, there are 32 different classes that define specific Nebo Expression terms and take a mode template parameter. These are the classes that define how fields, scalar values, operators, functions, conditional expressions, and stencil operations as defined in Chapter 6 actually operate. Since each of these classes behaves differently for each of the five modes, there are 160 different class definitions relating to how each specific Nebo Expression term behaves in each mode. Each of these class definitions is a partial template specialization, specialized on the mode with template parameters remaining for the field type and any operands that the term takes. For example, the full template signature of SumOp is `SumOp<Mode, Arg1, Arg2, FieldType>`, and for SinOp is `SinOp<Mode, Arg, FieldType>`. How each mode/backend is implemented is covered in Section 7.4.

7.3 Field type system and stencils

To numerically solve partial differential equations, Wasatch calculates scalar fields, vector fields, and tensor fields. Simply put, a scalar field has single scalar value for each

element of the field. A vector field has a vector for each element of the field. These vectors are represented by their scalar components, so each vector field contains three scalar values for each element of the field. A tensor field has a tensor for each element of the field. Like the vector field, these tensors are represented by their scalar components, so each tensor field contains nine scalar values for each element of the field. Wasatch stores all of its fields as scalar fields. Thus, Wasatch uses three scalar fields to represent one vector field and nine scalar fields to represent one tensor field.

Stencil operations, such as gradient and interpolant, convert scalar fields into vector (or tensor) fields. Other stencil operations, such as divergence and interpolant, convert vector (or tensor) fields back to scalar fields. Since Wasatch stores vector and tensor fields as scalar fields, a single operation like divergence, generally requires three or nine stencil operations respectively, one for each scalar value. To help keep everything straight, Wasatch uses four types of scalar fields: A volume field, and three face fields, each named for a different axes (X, Y, or Z).[2] In general, a volume field represents a scalar field, and together the three different face fields represent a vector field. A tensor field is represented by nine face fields, three of each type.

To further complicate matters, Wasatch uses staggered fields (or staggered meshes), where the elements are conceptually offset by half a cell width. There are four types of staggering: No offset, and offset along each of the different axes (X, Y, and Z). When comparing an unstaggered field with a field staggered along the X axis, the cell center of the staggered field lies directly on the boundary between two cells of the unstaggered field. Again to help keep everything straight, each staggered field is represented as a different type. Thus, there are 16 different scalar field types used in Wasatch and Nebo: For each of the four different staggers, there is a volume field and three face fields.

Nebo enforces the difference between these field types at compile-time. That is, all Nebo Expressions, except stencils, must be calculated from (and assign to) the same field type. For example, consider a modified version of the example from the last section, where VolField and XFaceField are unstaggered volume and X-face fields, respectively:

```
VolField a;
XFaceField b;
//...
b <<= 3.14159 + sin(a);
```

[2]Each face field represents some scalar value on the face that is normal, or perpendicular, to the axis it is named after. For example, an X-face field represents scalar values on the face normal to the X-axis.

This code will not compile, but instead raise compile-time errors about incompatible types. The original example, where the fields a and b had the same type, will compile for any of the sixteen field types.

Stencil operations are the only way in Nebo to calculate results from fields of different types. In fact, in the case of some interpolants, the stencil operation is only used to overcome Nebo's type restrictions. In these cases, the center of a volume field lines up with the center of a face field with a different stagger. When the centers line up in this way, interpolating from one type to the other is simply a copy, and no stencil points are needed. These stencils are called null stencils, precisely because the stencil is a direct copy of its argument.

However, these interpolants are the exception, not the rule. All other stencils in Nebo, including interpolants where field centers do not line up, use stencil points. As discussed in Section 6.6, Nebo stencil operations are implemented in terms of stencil points. In fact, a Nebo stencil is defined by four pieces of information: The field type of its argument, the field type that it returns, the list of stencil points it uses, and the list of coefficients that are multiplied to the value from each stencil point. For efficiency, all of this information, except the coefficients, is compile-time.

Just before evaluating a Nebo Expression that contains stencils, each stencil operation in the expression creates a shallow copy of its argument for each of its stencil points. Each shallow copy is passed a stencil point offset, which it uses to shift its memory accesses to the correct data. Shallow copies are used to avoid allocating space for and storing multiple copies of the same scalar values. During execution of an individual element, the argument copies evaluate their shifted point. Then, the stencil operation multiplies the values of each stencil point to the correct coefficients, and sums the result.

Nested stencils, where a stencil is part of the argument to another stencil, follow the same basic process. The outer stencil operation creates copies of its argument, which includes another stencil operation. Within each copy of the outer stencil operation's argument, the inner stencil creates copies of its argument. However, the inner stencil does not directly pass its stencil points to the copies of its argument. Instead the inner stencil combines the shift/stencil point it receives from the outer stencil with its own stencil points to create the shifts it passes to the copies of its arguments. In Nebo, this process of nesting stencils is generalized to work with arbitrary nesting of stencils.

Finally, as mentioned in Section 6.6, stencil operations have edge cases where they cannot evaluate elements because the stencil shape would go outside of the field. Nebo stencil operations automatically account for these invalidated ghost cells. Wasatch uses

ghost cells to cover these edge cases. Ghost cells are populated with values, generally from bordering partitions of the simulation. Thus, Nebo skips ghost cells in assignment where the shape of the stencils used cannot produce valid values. Furthermore, Nebo raises an error when there are not enough ghost cells to cover the edge cases. These invalidated ghost cells are determined directly from the largest extents of the stencil points (including nested stencils) in the expression.

7.4 Backends

This section discusses how Nebo decides which backend to use during execution and how each backend of Nebo works. As discussed in Section 7.2, Nebo uses different modes to implement different backends to Nebo. The different modes are represented by a template argument to a Nebo Expression term. Each mode for each Nebo Expression term is a partial template specialization. Each partial template specialization has no restrictions on what it can and cannot contain, beyond the limits of a C++ class. However, by convention in Nebo's implementation, each mode provides a uniform interface. For example, every term's `Initial` mode implements an `init`() method, which takes arguments for the valid number ghost cells remaining and the shift it needs to perform and returns the same Nebo Expression but in the SeqWalk mode. Similarly, every term's SeqWalk mode provides an `eval`() method, which takes no arguments and returns the value of the Nebo Expression at the current element. Thus, Nebo's partial template specializations behave much like C++ classes that have inherited some basic interface. This convention creates a uniform way for Nebo Expression terms to interact with their subexpressions. On one hand, Nebo does not benefit from the compile-time error checking that traditional C++ inheritance provides. That is, if some part of Nebo breaks a mode's interface convention, any resulting errors generally do not clearly state how the interface convention has been violated. On the other hand, Nebo's interface convention avoids the use of virtual tables for function lookups.

Nebo uses a mix of compile-time and runtime parameters to determine which backends to use. For the compile-time flags, Nebo uses the C preprocessor macros `#def` and `#ifdef` add or ignore Nebo's various backends. By default, Nebo only compiles the single-core CPU backend, and regardless of how flags are set this backend is always available. The thread-parallel and GPU backends are compiled by defining `ENABLE_THREADS` and `ENABLE_CUDA`, respectively. Furthermore, for the GPU backend to be used, the code must be compiled by NVidia's CUDA compiler, nvcc.

At runtime, the `Initial` mode of a Nebo Expression is constructed first. If the Nebo

Expression is used in a reduction, the expression switches to the reduction mode, and continues as explained in Section 7.4.4. If the Nebo Expression is used in an assignment, Nebo then chooses which particular backend to use based on choices implicitly made by the user/developer. Assuming all backends are compiled, Nebo first checks the location of the memory for the result. If the memory is located on a GPU, then Nebo uses its GPU backend. If the memory is located on a CPU, then Nebo checks the number of active threads in the thread-pool it uses. If there is more than one active thread, Nebo uses its thread-parallel backend. Otherwise, Nebo uses its single-core CPU backend. Of course, if a particular backend of Nebo is not compiled, Nebo will skip the check to use that backend.

Because of its scope, Nebo leaves the decisions of how many threads to use and where to allocate memory up to end users. Nebo only considers how to efficiently compute the calculation in a single Nebo Expression. Determining how many threads to use depends on how many cores are present and available, the effectiveness of the current hardware's hyper-threading, and what other calculations are running concurrently. Likewise, determining whether to run on a CPU or GPU depends on the speed of the CPU and GPU available, the current load on each processor, the location of the input data to the current calculation, and whether or not the current calculation's result is needed for future calculations that must run on the CPU. Nebo has no way of knowing any of this information, let alone determining the best approach. These are important issues that determine the efficiency of the overall application, and as such Nebo's syntax and different backends make it easy and simple to change scheduling and memory locality decisions.

With Nebo's declarative syntax, each backend is able to specify its own execution model. These execution models are described in the following sections.

7.4.1 Single-core implementation

The `SeqWalk` mode[3] implements Nebo's default single-core execution implementation. The `SeqWalk` mode uses an interface for Nebo Expressions which is almost identical to the standard interface for forward iterators. There are two differences between the single-core interface and forward iterators' interface: First, the single-core interface allows for arbitrary evaluation rather than just dereference. Second, rather than using C++'s increment operator (++), Nebo uses the method name `next`().

The interface has two functions for the right-hand side (Nebo Expression) of an assignment: A `next`() method that moves the underlying iterators to the next element in

[3] `SeqWalk` is short for sequential walk.

the fields, and a `eval()` method which evaluates the value of the Nebo Expression at the current element. The constructor for the `SeqWalk` mode initializes all underlying iterators. The interface has three functions for the left-hand side of an assignment (a `NeboField` object): A `next()` method that moves the underlying iterator to the next element in the field, a `ref()` method which returns a reference to the current element of the underlying field, and an `at_end()` method that returns true if and only if the underlying iterator is at the end of the field. The interface for the right-hand side (Nebo Expression) is public, but the interface for the left-hand side is private. To execute the assignment, Nebo calls the `assign` method on the left-hand side, passing the right-hand side as an argument:

```
template<typename RhsType>
inline void assign(RhsType rhs) {
  while(!at_end()) {
    ref() = rhs.eval();
    next();
    rhs.next();
  }
}
```

Since the `assign` method is part of the left-hand side, it has access to the necessary methods for both the left-hand and right-hand sides of the assignment. The while loop in the `assign` method iterates over all the elements in the underlying fields and is simple intentionally. The simplicity of this loop has two main advantages. First, the simplicity is easy to maintain and debug. Second, with all the methods marked as inline, optimizing compilers can recoginize the pattern of the loop and apply standard optimizations, such as loop unrolling.

For example, consider the single-core execution of the example from Section 7.2:

```
Field a, b;
//...
b <<= 3.14159 + sin(a);
```

Ignoring the `SeqWalk` mode and the field type template arguments, the type of the left-hand side of this assignment becomes `NeboField`. The type of the right-hand side of this assignment becomes `SumOp` < `NeboScalar`, `SinOp` < `NeboField` > >.

Each call to `lhs.ref()` and `lhs.next()` behave exactly like the methods for forward iterators over mutable data. Each call to `rhs.eval()`, `SumOp`'s evaluate method, calls the evaluate method on both of its arguments, adds the values from these evaluate calls together, and returns the result. Each call to `NeboScalar`'s evaluate method simply returns its scalar value, which is in this case 3.14159. Each call to `SinOp`'s evaluate method calls the evaluate method on its argument, applies the sine function to the value from this evaluate call, and returns the result. Each call to `NeboConstField`'s evaluate method dereferences the value

from the iterator it contains and returns that value.

Each call to `rhs.next()`, `SumOp`'s increment method, simply calls the increment method on both of its arguments. Each call to `NeboScalar`'s increment method does nothing—a scalar value has nothing to change. Each call to `SinOp`'s increment method calls the evaluate method on its argument. Each call to `NeboConstField`'s increment method increments the iterator it contains.

While there are a lot of function calls for each iteration of the above while loop, all of these functions are declared with the C++ keyword `inline`. Defining a function with the `inline` keyword does not guarantee that that function will be inlined. Fortunately, these function calls are textbook examples of functions to inline: First, they are short and simple (other than calls to other short and simple functions). Second, each function is used in exactly one location. Thus, when compiling Nebo with standard optimizations, such as gcc's `-O3` optimizations, gcc generally inlines most evaluate and increment function calls.

7.4.2 Multicore implementation

Nebo's strategy for multithread execution is to divide the fields underlying the current Nebo Expression into subfields. Each subfield is then assigned to a thread and executed sequentially on that thread. Nebo's semantics define that elements in Nebo assignment can be evaluated and assigned in any order. Thus, there is no need for communication between threads other than to signal that a subfield has finished execution.

When Nebo has determined that it is to use the multicore backend, Nebo determines the active number of threads. Then Nebo partitions the fields underlying the current Nebo Expression into subfields. Nebo partitions the fields along a single axis, which is determined by the shape of the original fields. Nebo picks an axis, based on size of the original field and on how contiguous the underlying memory will be for each field. Thus, given the typical memory layout in Wasatch, Nebo prefers to partition fields along the Z axis. Once Nebo has determined its partitioning scheme, Nebo creates an instance of the Nebo Expression in `Resize` mode for each subfield. Each instance of the Nebo Expression is passed information about the location and size of its subfield as well as a pointer to a semaphore. The original Nebo Expression passes each `Resize` instance to a FIFO thread pool to schedule the calculation. The original instance of the Nebo Expression in `Initial` mode then waits on the semaphore for each `Resize` instance to post to the semaphore. Once a Nebo Expression in `Resize` mode is started in a different thread, it creates an instance of the Nebo Expression in `SeqWalk` mode. After the `SeqWalk` instance has finished its assignment, the `Resize` instance posts to the semaphore.

7.4.3 Many-core (GPU) implementation

Because GPUs use a SIMD model of execution, Nebo's GPU backend is very different from Nebo's other backends. Nebo's GPU backend sets up a "plane" of threads, such that each thread has a unique pair of X-axis and Y-axis indices. Then all the threads together iterate through all the Z indices. At each Z index, each thread calculates the result for its unique combination of X, Y, and Z indices. For example, consider a field whose dimensions are 3 by 4 by 5 (X, Y, Z, respectively). In this case, Nebo's GPU backend would use 12 threads (three times four), and each thread would calculate five different elements.

When Nebo determines that it is to execute on the GPU, it builds an instance of the Nebo Expression in `GPUWalk` mode. Nebo then calls a templated CUDA kernel:

```
template<typename LhsType, typename RhsType>
__global__ void gpu_assign_kernel(LhsType lhs, RhsType rhs) {
  lhs.assign(rhs);
};
```

The code of the `assign` method for the left-hand side is deceptively like the code for sequential execution:

```
template<typename RhsType>
__device__ inline void assign(RhsType rhs) {
  const int ii = blockIdx.x * blockDim.x + threadIdx.x;
  const int jj = blockIdx.y * blockDim.y + threadIdx.y;

  start(ii, jj);
  rhs.start(ii, jj);

  while(!at_end()) {
    if(valid()) { ref() = rhs.eval(); };
    next();
    rhs.next();
  };
};
```

Despite the differences between execution model, the code to execute Nebo assignment on a GPU looks very similar: The first obvious difference between the sequential CPU code and the GPU code is the use of the `start` method. Because the CUDA constructs the Nebo Expression for each thread exactly the same for all threads, except for a few indexing variables (`blockIdx`, `blockDim`, and `threadIdx`), each thread must determine what its assigned X-axis and Y-axis indices are. The next obvious difference is the guard method, `valid`(). For the sake of execution speed and regularity, sometimes threads are assigned X-axis and Y-axis indices that are outside the bounds of the fields. Thus, the `valid`() check is needed to ensure that the given X-axis and Y-axis indices point to a valid element of the fields.

Because Nebo uses asynchronous kernel invocations, some synchronization is handled by the end user. When the `ENABLE_CUDA` C preprocessor flag is defined at compile-time, each field contains a `cudaStream_t`, which defaults to the default CUDA stream (0). A `cudaStream_t` informs the CUDA software and GPU hardware what CUDA kernels must be executed in order and which can be run in parallel. Two CUDA kernels invoked with the same CUDA stream are executed in the order they were invoked. Conversely, two CUDA kernels invoked with different CUDA streams can be executed in any order, even concurrently. When Nebo invokes the templated CUDA kernel above, Nebo passes the CUDA stream in the field on the left-hand side of the assignment (the assignee). When Wasatch is compiled with Nebo's GPU backend enabled, it copies the same CUDA stream into all fields that are assigned in the same task. Furthermore, each task has its own CUDA stream object. Thus, all calls to Nebo in a Wasatch task are in the same stream, and calls to Nebo from different Wasatch tasks are in different streams.

Finally, just before a Wasatch task finishes, it calls `cudaStreamSynchronize`() with the task's CUDA stream. This synchronization point forces all the Nebo CUDA kernels invoked in a task to finish before the Wasatch task can finish. While Nebo could handle all of the synchronization internally, having the users of Nebo handle the synchronization reduces the synchronization overhead.

7.4.4 Reduction implementation

The `Reduction` mode implements Nebo's reduction operations. Currently, Nebo reductions only work for single-core execution. Thus, the `Reduction` mode uses an interface for Nebo Expressions which is almost identical to the interface for `SeqWalk` mode. The major difference between a reduction and a single-core assignment is that there is no left-hand-side/assignee in a reduction. So, in addition to increment and evaluate methods, a Nebo Expression in `Reduction` mode provides an end condition method (`at_end`). Like single-core assignment, the constructor for the `Reduction` mode is required to initialize all underlying iterators.

The reduction-core backend uses a while similar to the loop in the single-core backend:

```
while(!(expr.at_end())) {
  result = proc(result, expr.eval());
  expr.next();
};
```

The increment, evaluate, and end condition methods all behave the same way as they do in the single-core backend. However, the reduction backend does have the sensible restriction

that a reduction must be over a Nebo Expression that has a definite size. Any Nebo Expression that contains at least one field has a definite size. Any Nebo Expression whose terminals are only scalars does not have a definite size. Fortunately, a Nebo Expression made up of only scalars is itself a scalar, and so a reduction is unnecessary.

7.5 Summary

This chapter presents the "magic" behind Nebo. If Nebo were implemented without any one part discussed in this chapter, Nebo would not be nearly as easy to use or as useful as it is now. Without template metaprogramming to do the parsing, Nebo would require a stand-alone compiler. Other DSL projects use stand-alone compilers, and stand-alone compilers do have advantages. On one hand, with a stand-alone compiler, optimizations are not limited by constraints imposed by pre-existing language design. While Nebo is limited in its optimizations, the next chapter will show that it does not suffer for this limitation. On the other hand, stand-alone compilers complicate build systems. In large existing software projects, build systems are complicated enough without another component to maintain. Since Nebo is implemented as a header file, adding Nebo fits into the standard C++ framework for including simple libraries.

Without stencil operations and Nebo's type system, Nebo would be limited to point-wise functions over fields of the same type, such as algebraic, trigonometric, and conditional calculations. While Nebo could be still useful with such a restriction, Nebo could not fulfill its purpose: Numerically solving partial differential equations requires stencil operations to implement various vector operators. Other abstractions exist for these vector operators, such as `apply_to_field` discussed in Section 6.6. Unfortunately, these abstractions limit loop fusion, other optimizations, and performance as will be seen in Section 8.2 of the next chapter.

Finally, without Nebo's multiple backends, Nebo loses much of its appeal to application developers. When refactoring existing code or writing new code, application developers need a clear benefit to using a new tool. Beyond Nebo's simple and clean syntax, Nebo's backends provide two clear benefits. First, Nebo's single-core implementation performs at least as well and sometimes significantly better than the hand-written C++ code that it replaces.[4] Second, Nebo's multiple backends mean that developers can target multiple architectures without maintaining a separate implementation for each architecture. The next chapter demonstrates these two benefits of Nebo.

---

[4]Section 8.2 contains an example of this benefit.

CHAPTER 8

CASE STUDIES OF THE USE AND PERFORMANCE OF NEBO

8.1 Introduction

All the case studies in this chapter, with the exception of the first (Section 8.2), are taken directly from Wasatch [103]. As such, the five case studies taken from Wasatch were written by Wasatch developers, and were written for their own uses. The case studies from Wasatch show how Nebo's end users (who are not Nebo developers) actually use Nebo. Wasatch can be downloaded using svn from:

`https://gforge.sci.utah.edu/svn/uintah/trunk`

There are a few caveats about the Wasatch code presented here as well as the results. First, other than the Taylor-Green vortex [130, 31] tests (Sections 8.6 and 8.7), these tests were run outside of Wasatch. Each benchmark is essentially a single Wasatch task, except again for the Taylor-Green vortex tests. Thus, these results reflex Nebo's performance without the overhead Wasatch would bring to such small benchmarks.

Second, the case studies which include older Wasatch code as originally written will no longer run in Wasatch. Before Nebo was introduced, compound assignment operators were implemented as a "one-operation" version of Nebo. For example, the following compound addition assignment,

```
Field a, b;
//...
b += a;
```

is equivalent to the following Nebo assignment:

```
Field a, b;
//...
b <<= b + a;
```

Also, the traditional C++ assignment operator (=) was implemented to initialize fields, either from another field or from a scalar value. Since Nebo's adoption, this syntax has been removed, and there have been major improvements to underlying structures in the interim.

In the interest of a fair comparison of coding styles, in the benchmarks I have replaced all uses of the compound assignment and traditional assignment with the line-for-line Nebo equivalent. Thus the obsolete code can take advantage of updates and improvements to the underlying structures, just as the new Nebo-based code can.

Finally, Wasatch is not a static software project. Since Nebo's adoption, Wasatch code has changed. Thus, pre-Nebo Wasatch code will have different flags to do slightly different tasks than the current Wasatch code based on Nebo. Each time there is a difference, the benchmarks were run such that they calculate the same result, with perhaps some rounding error due to a different order of operations. Furthermore, I specify what code is executed for each benchmark.

The first four case studies, those run outside of Wasatch, were run on all of Nebo's backends. The following tables summarize the results: Table 8.1 shows the improved performance of using Nebo over older coding styles. Table 8.2 shows the performance of using Nebo with 2, 4, 6, 8, 10, or 12 threads as well as the improved performance of the GPU backend with a field size of 64^3. Table 8.3 likewise shows the performance of using Nebo with its various backends but with a field size of 128^3.

The simple heat equation in Section 8.2 is the only case study not written by Wasatch developers. It was written to demonstrate the simple syntax of nesting stencil operations. The Scalar right-hand side term in Section 8.3 is a good example of how Nebo has been used as it has been developed and contains three different versions of the same computation. The detailed conditional expression in Section 8.4 shows the utility of having a nestable conditional expression in Nebo. Section 8.5 presents a complex use of Nebo and shows how complex point-wise calculations in Nebo can become. Sections 8.6 and 8.7 both present different aspects of the Taylor-Green vortex test. Section 8.6 discusses comparing Wasatch and Nebo to other components of Uintah, which do not use a DSL for their numeric calculations. Section 8.7 discusses weakly scaling the Taylor-Green vortex test on the supercomputer Titan [25]. All tests were run on two Intel Xeon E5-2620 (six processors each, 2.00 GHz) with 16 Gb RAM.

8.2 Simple heat equation

Consider the following parabolic PDE:

$$\frac{\partial T}{\partial t} = \underbrace{\nabla \cdot (-\lambda \nabla T)}_{\phi} \tag{8.1}$$

In the right-hand side of this equation (ϕ), the fields, λ and T, are input, and several stencil operators are applied before assigning the result to ϕ. Namely, the gradient operator

Table 8.1. Increases in performance from better use of Nebo. Heat equation compares the single-core runtime performance of the original code and the Nebo code from Section 8.2, in Figures 8.1 and 8.2, respectively. First scalar RHS compares the single-core runtime performance of the original code and the initial Nebo version from Section 8.3, in Figures 8.3 and 8.4, respectively. Second scalar RHS compares the single-core runtime performance of the original code and the current Nebo version from Section 8.3, in Figures 8.3 and 8.5, respectively.

Size	Heat equation	First scalar RHS	Second scalar RHS
64^3	0.94x	1.00x	1.88x
128^3	0.95x	1.03x	1.91x

Table 8.2. Nebo speedup on real uses in Wasatch on problem size 64^3. Heat equation is the Nebo code from Section 8.2, in Figure 8.2. First scalar RHS is the initial version of the scalar right-hand side code that uses Nebo from Section 8.3, in Figure 8.4. Second scalar RHS is the current version of the scalar right-hand side code that is in Wasatch from Section 8.3, in Figure 8.5. Conditional is the benchmark from Section 8.4, in Figure 8.6. Complex is the benchmark from Section 8.5, in Figure 8.7.

Benchmark	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	GPU
Heat equation	1.31x	2.41x	2.45x	2.59x	3.26x	3.28x	13.78x
First scalar RHS	1.65x	1.61x	1.99x	1.85x	2.14x	2.07x	12.49x
Second scalar RHS	1.81x	2.93x	2.93x	3.41x	3.63x	3.60x	16.25x
Conditional	1.93x	3.42x	5.31x	3.91x	5.33x	5.51x	23.12x
Complex	1.84x	2.33x	3.61x	4.89x	5.88x	5.52x	27.27x

Table 8.3. Nebo speedup on real uses in Wasatch on problem size 128^3. Heat equation is the Nebo code from Section 8.2, in Figure 8.2. First scalar RHS is the initial version of the scalar right-hand side code that uses Nebo from Section 8.3, in Figure 8.4. Second scalar RHS is the current version of the scalar right-hand side code that is in Wasatch from Section 8.3, in Figure 8.5. Conditional is the benchmark from Section 8.4, in Figure 8.6. Complex is the benchmark from Section 8.5, in Figure 8.7.

Benchmark	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	GPU
Heat equation	1.91x	3.86x	4.91x	6.84x	7.76x	5.98x	25.99x
First scalar RHS	1.85x	3.03x	2.99x	3.29x	2.80x	3.08x	10.43x
Second scalar RHS	2.00x	3.63x	4.95x	6.53x	6.08x	4.81x	13.51x
Conditional	1.92x	3.87x	5.39x	6.13x	6.97x	6.65x	33.57x
Complex	1.90x	3.60x	5.19x	6.00x	7.33x	6.63x	37.42x

(∇) is applied to T, and there is an implicit interpolant operator applied to λ.

The results of these two operators are multiplied together and then negated. Finally, the divergence operator ($\nabla\cdot$) is applied to the negated multiplication.

To complicate matters, each of these operators has an implied directionality to it. When the directions are explicit, the right-hand side of this equation becomes:

$$\phi = \nabla_x \cdot (-\lambda_x \nabla_x T) \tag{8.2}$$
$$+ \nabla_y \cdot (-\lambda_y \nabla_y T) \tag{8.3}$$
$$+ \nabla_z \cdot (-\lambda_z \nabla_z T) \tag{8.4}$$

The code necessary to calculate the right-hand side of this equation is in Figure 8.1. Before Nebo was available, this equation required 16 loops to implement.

The code for each direction's calculation is detailed and yet repetitive. The difference between calculating the X direction (lines 1-4), calculating the Y direction (lines 6-9), and the Z direction (lines 11-14) is limited to the names of variables, and stencil operations[1]. Of course, the internal implementations of the stencils are different; however, from the end user's perspective, these stencils are only slight variations of the same operation. Additionally, because of the use of the `apply_to_field` method, this code cannot easily be simplified or optimized through loop fusion.

However, when using Nebo, the code mirrors the directionally-explicit mathematical terms (equations 8.2-8.4), as seen in Figure 8.2.

While this code is still repetitious in its use of stencils, it fuses the 16 loops into a single loop. Additionally, the Nebo version of this simple heat equation is easier to write, read, and maintain than the pre-Nebo version.

In this code, there is no branching, so it is obvious how the code was executed.

Because Nebo currently forces nested stencil expressions to recalculate all intermediate results, nested stencil expression can cause minor slow down for single-core execution, as seen in Table 8.1. While this recalculation is a current problem, it is not a fundamental part of Nebo. Optimizations could be added to Nebo to save intermediate results and use them rather than recalculating them each time. Nebo could break its single calculation loop into several, mimicing the old syntax, but this would require saving all intermediate results. Alternatively, in a single calculation loop, Nebo could do a form of memoization. Nebo could calculate the intermediate results the first time they are needed, and then store the value until it is no longer needed.

---

[1]The field types are also different, but discussion of that aspect of Nebo has been left out for clarity and space constraints.

```
 1 InterpX.apply_to_field(lambda, lambdaX);
 2 GradX.apply_to_field(T, gradTX);
 3 lambdaX *= gradTX;
 4 DivX.apply_to_field(lambdaX, phiX);
 5
 6 InterpY.apply_to_field(lambda, lambdaY);
 7 GradY.apply_to_field(T, gradTY);
 8 lambdaY *= gradTY;
 9 DivY.apply_to_field(lambdaY, phiY);
10
11 InterpZ.apply_to_field(lambda, lambdaZ);
12 GradZ.apply_to_field(T, gradTZ);
13 lambdaZ *= gradTZ;
14 DivZ.apply_to_field(lambdaZ, phiZ);
15
16 phi = 0.0;
17 phi -= phiX;
18 phi -= phiY;
19 phi -= phiZ;
```

Figure 8.1. Code to evaluate equation (8.1). Lines 1-4, 6-9, and 11-14 correspond to calculating equations (8.2), (8.3), and (8.4), respectively. The final block of code (lines 16-20) calculates the final result from the intermediate results of the previous code.

```
phi <<=   DivX(-InterpX(lambda) * GradX(T))
        + DivY(-InterpY(lambda) * GradY(T))
        + DivZ(-InterpZ(lambda) * GradZ(T));
```

Figure 8.2. Nebo code to evaluate equation (8.1).

Table 8.3 show that the Nebo code scales linearly (with greater than 90% efficiency) up to four processors, with fields sized 128^3. Table 8.2 show that the Nebo code scales linearly (with greater than 90% efficiency) only up to two processors, with fields sized 64^3. Both tables show greater than 13x speed up with Nebo's GPU backend. The simple heat equation example is a very simple calculation, and therefore is memory-bound, limiting its scalability.

8.3 Scalar right-hand side term

The scalar right-hand side Wasatch task is a great example of how Nebo has improved code development in Wasatch. The original code in Figure 8.3 predates Nebo's adoption, and uses the compound assignment operator as well as the original object-oriented method syntax for stencil operations. This code is old enough that it predates the field type restrictions on stencil operators. Quite simply, this code checks first for how to

```
 1  rhs = 0.0;
 2
 3  if( doXDir_ ){
 4    if( haveConvection_ ){
 5      divOpX_->apply_to_field( xConvFlux_, tmp );
 6      rhs -= tmp;
 7    }
 8    if( haveDiffusion_ ){
 9      divOpX_->apply_to_field( xDiffFlux_, tmp );
10      rhs -= tmp;
11    }
12  }
13
14  if( doYDir_ ){
15    if( haveConvection_ ){
16      divOpY_->apply_to_field( yConvFlux_, tmp );
17      rhs -= tmp;
18    }
19    if( haveDiffusion_ ){
20      divOpY_->apply_to_field( yDiffFlux_, tmp );
21      rhs -= tmp;
22    }
23  }
24
25  if( doZDir_ ){
26    if( haveConvection_ ){
27      divOpZ_->apply_to_field( zConvFlux_, tmp );
28      rhs -= tmp;
29    }
30    if( haveDiffusion_ ){
31      divOpZ_->apply_to_field( zDiffFlux_, tmp );
32      rhs -= tmp;
33    }
34  }
```

Figure 8.3. Original code for Scalar right-hand side term.

handle dimensionality (do the fields have one, two, or three dimensions), checking each axis independently. For each axis that is valid, the code checks for convection and diffusion, computing them and adding them in independent of one another. For the benchmarks, every branch is taken; that is, the code calculates convection and diffusion in three dimensions.

When Nebo was first introduced, stencils were not an acceptable part of the language, and as such Wasatch solely used the `apply_to_field` operator. While the code after refactor in Figure 8.4 follows much the same pattern as the original version, it does improve upon its use of the divergence operator. Convection and diffusion still are handled separately, but divergence is applied afterwards (divergence distributes over subtraction). This control flow means exactly three divergence operators are executed rather than a flexible number (zero to six) of the original. The three-dimensional case is the most common in this domain,

```
rhs <<= 0.0;

if( doXDir_ ){
  if( haveConvection_ ) tmpx <<= - xConvFlux_;
  else                  tmpx <<= 0.0;
  if( haveDiffusion_ )  tmpx <<= tmpx - xDiffFlux_;
  if( haveXAreaFrac_ ){
    xAreaFracInterpOp_->apply_to_field( xareafrac_, xAreaFracInterp );
    tmpx <<= tmpx * xAreaFracInterp;
  }
  divOpX_->apply_to_field( tmpx, rhs );
}

if( doYDir_ ){
  if( haveConvection_ ) tmpy <<= yConvFlux_;
  else                  tmpy <<= 0.0;
  if( haveDiffusion_ )  tmpy <<= tmpy + yDiffFlux_;
  if( haveYAreaFrac_ ){
    yAreaFracInterpOp_->apply_to_field( yareafrac_, yAreaFracInterp );
    tmpy <<= tmpy * yAreaFracInterp;
  }
  divOpY_->apply_to_field( tmpy, tmp );
  rhs <<= rhs - tmp;
}

if( doZDir_ ){
  if( haveConvection_ ) tmpz <<= zConvFlux_;
  else                  tmpz <<= 0.0;
  if( haveDiffusion_ )  tmpz <<= tmpz + zDiffFlux_;
  if( haveZAreaFrac_ ){
    zAreaFracInterpOp_->apply_to_field( zareafrac_, zAreaFracInterp );
    tmpz <<= tmpz * zAreaFracInterp;
  }
  divOpZ_->apply_to_field( tmpz, tmp );
  rhs <<= rhs - tmp;
}
```

Figure 8.4. Initial version of Nebo code for Scalar right-hand side term.

which this control flow optimizes for. Despite the control flow, this first refactor does not use Nebo very efficiently. Finally, this version adds an area fracture interpolant. For the benchmarks, every branch but the area fracture interpolants are taken; thus, the code calculates convection and diffusion in three dimensions, as did the original code.

The current version in Figure 8.5 makes full use of Nebo. In fact, this code takes manual inlining to an extreme not generally seen elsewhere in Wasatch. Rather than independently handling the dimensions, the code checks for the most common cases, three-dimensional fields with convection and diffusion, with or without the area fracture interpolant. For the benchmarks, again the simulation is on three-dimensional fields with convection and diffusion but not the area fracture interpolant, which means lines 9-11 are the only ones

```
if( doXDir_ && doYDir_ && doZDir_ && haveConvection_ && haveDiffusion_ ){
  // inline everything
  if( haveXAreaFrac_ ){
    rhs <<= -(divOpX_)( (xAreaFracInterpOp_)(xareafrac_) *
                        ( xConvFlux_ + xDiffFlux_ ) )
            -(divOpY_)( (yAreaFracInterpOp_)(yareafrac_) *
                        ( yConvFlux_ + yDiffFlux_ ) )
            -(divOpZ_)( (zAreaFracInterpOp_)(zareafrac_) *
                        ( zConvFlux_ + zDiffFlux_ ) );
  }
  else{
    rhs <<= -(divOpX_)( xConvFlux_ + xDiffFlux_ )
            -(divOpY_)( yConvFlux_ + yDiffFlux_ )
            -(divOpZ_)( zConvFlux_ + zDiffFlux_ );
  }
}
else{
  // handle 2D and 1D cases - not quite as efficient since we won't be
  // running as many production scale calculations in these configurations
  // ...
```

Figure 8.5. Current version of Nebo code for Scalar right-hand side term. The code to handle the Y dimension has been shortened and code to handle the Z dimension has been omitted due to space constraints. The omitted code is almost identical to the code from Figure 8.4, allowing for any subset of the dimensions to be inactive.

run for the benchmarks.

Table 8.1 shows that there is no real improvement in using the first version of Nebo over the original code for the scalar right-hand side term. Given the similarity of structure between the original code and the first Nebo version of the code, this is not surprising. On the other hand, Table 8.1 also shows that the current Nebo version of the code performs 1.9x faster than the original version of the code, with Nebo's single-core backend. This is a prime example of how loop fusion, through Nebo's syntax, clearly benefits performance.

Tables 8.2 and 8.3 show that the first Nebo version of the scalar right-hand side term does not scale. However, these tables do show that the current Nebo version of the code scales much better, especially with fields sized 128^3. While the scaling of the current version is not ideal, this code has not be modified in anyway for parallel execution. The scalar right-hand side term is about as complex as the simple heat equation example of the last section, and likewise is memory-bound, limiting its scalability.

8.4 A detailed conditional expression

A Wasatch task to calculate the effective viscosity of a precipitate uses a complex conditional expression in Nebo as seen in Figure 8.6. Interestingly, this conditional contains

```
1  result <<= cond(volumeFraction < 1e-10,
2                  baseViscosity)
3                 (1.0 > (1 + 2.5 * corrFac * volumeFraction) *
4                        pow(2.0 * strainMagnitude, power/2),
5                  baseViscosity)
6                 (sqrt(2.0 * strainMagnitude) < minStrain &&
7                    1.0 > (1 + 2.5 * corrFac * volumeFraction) *
8                          pow(minStrain, power),
9                  baseViscosity)
10                (sqrt(2.0 * strainMagnitude) < minStrain,
11                 (1 + 2.5 * corrFac * volumeFraction) *
12                  pow(minStrain, power) * baseViscosity)
13                ((1 + 2.5 * corrFac * volumeFraction) *
14                 pow(2.0 * strainMagnitude, power/2) * baseViscosity);
```

Figure 8.6. A complex use of Nebo conditionals.

several values that are scalar. For example, volumeFraction and baseViscosity are both scalar, which makes the first clause completely scalar in practice. Furthermore, the first half of the third clause's conditional is the same as the whole conditional for the fourth clause.

Tables 8.2 and 8.3 show that this conditional use of Nebo is complex enough to scale to six processors with 90% efficiency with Nebo's multicore backend. This conditional case is more computationally complex than the scalar right-hand side term or the simple heat equation example of the last two sections, and as such scales better than the other benchmarks. Unfortunately, this benchmark is still a memory-bound calculation, and as such does not scale as much further than the previous two benchmarks on Nebo's multicore backend. Yet, this benchmark's performance is much better than the previous two benchmarks on Nebo's GPU backend. This benchmark's GPU performance is surprising given that this benchmark is based on branching behavior. However, Nebo's implementation of `cond` lifts as much as possible out of its branches, which aids the GPU performance by spending as little time as possible in the `cond` branches.

8.5 A complex use of Nebo

The code in Figure 8.7 implements a homogeneous nucleation term. Other than being a highly complex calculation, this code is interesting because it could be represented as just two Nebo assignments (one for each branch of the if statement). However, the Wasatch developer who wrote it decided to break it into separate terms. By doing so, the developer avoided verbosely repeating the definitions of the assignee fields. For the benchmarks, I chose to follow the then branch (lines 3-13 and 30-39).

```
if ( surfaceEngTag_ != Expr::Tag() ) {

  delG <<= 16.0 * PI / 3.0 * molecularVolume * molecularVolume *
           surfaceEng * surfaceEng * surfaceEng / (KB * KB *
           temperature * temperature * log(superSat) *
           log(superSat)) + KB * temperature * log(superSat) -
           surfaceEng * pow(36.0 * PI * molecularVolume *
           molecularVolume, 1.0 / 3.0);

  iC <<= 32.0 * PI / 3.0 * molecularVolume * molecularVolume *
         surfaceEng * surfaceEng * surfaceEng / (KB * KB * KB *
         temperature * temperature * temperature * log(superSat) *
         log(superSat) * log(superSat));
}
else {

  delG <<= 16.0 * PI / 3.0 * molecularVolume * molecularVolume *
           surfaceEnergy * surfaceEnergy * surfaceEnergy / (KB *
           KB * temperature * temperature * log(superSat) *
           log(superSat)) + KB * temperature * log(superSat) -
           surfaceEnergy * pow(36.0 * PI * molecularVolume *
           molecularVolume, 1.0 / 3.0);

  iC <<= 32.0 * PI / 3.0 * molecularVolume * molecularVolume *
          surfaceEnergy * surfaceEnergy * surfaceEnergy / (KB *
          KB * KB * temperature * temperature * temperature *
          log(superSat) * log(superSat) * log(superSat));
}

z <<= sqrt( delG / (3.0 * PI * KB * temperature * iC * iC));

kF <<= diffusionCoef * pow(48.0 * PI * PI * molecularVolume *
       iC, 1.0 / 3.0);

N1 <<= NA * eqConc * superSat;

result <<= cond(superSat > 1.0,
                z * kF * N1 * N1 * exp( - delG / KB / temperature))
               (0.0);
```

Figure 8.7. A complex use of Nebo.

While Table 8.2 shows that this benchmark struggles to scale past two processors with fields sized 64^3, Table 8.3 shows that these benchmarks scale linearly, with at least 86% efficiency, up to six processors with fields sized 128^3 on Nebo's multicore backend. Furthermore, both tables show that this benchmark performs better than any of the other benchmarks on Nebo's GPU backend. These results are hardly surprising because this benchmark, despite being broken up into six loops, is more computationally bound than the other benchmarks. Finally, it bears mentioning again that this benchmark was not optimized for parallel execution.

8.6 Comparing Wasatch to Arches and ICE

The Taylor-Green vortex [130, 31] is a classic two-dimensional fluid dynamics problem, which has an analytic solution. Because the Taylor-Green vortex has a closed-form solution, it is often used to verify general algorithms for fluid dynamics.

In this section, I am not using the Taylor-Green vortex for verification but rather for comparison. Since the Taylor-Green vortex is a classic problem, many software packages for computational fluid dynamics implement the Taylor-Green vortex. In particular, Arches [120] and ICE [59] are components of Uintah, just as Wasatch is. As components of Uintah, Arches and ICE use the same interface and framework for communication and data storage as Wasatch does. Arches, ICE, and Wasatch take different approaches and target different problem domains. While this comparison does not validate Uintah's framework, this comparison does show that Wasatch's approach and use of Nebo is competitive. In particular, Nebo does not create overhead that prevents Wasatch from performing better than both Arches and ICE and even helps Wasatch's performance. It also bears mentioning that the timings reported in this section exclude the linear solver, which is used in the same manner across all three components. Thus, the differences in performance are in the direct control of the developers of each component.

Table 8.4 presents the results from comparing Wasatch, Arches, and ICE on a single processor using a single thread. While using small domain sizes, Wasatch, Arches, and ICE perform roughly the same with Wasatch doing slightly better. As the domain size grows, Wasatch performs increasingly well compared to Arches and ICE. At the largest size, Wasatch runs roughly six times faster than Arches and roughly an order of magnitude faster than ICE.

Nebo is only a part of Wasatch and can only take part of the credit for Wasatch's performance. However, Nebo does perform most of Wasatch's numeric calculations. Nebo has a fixed amount of overhead for the Taylor-Green vortex code. As the domain size grows, overhead becomes a smaller part of the execution of each simulation.

Table 8.4. Taylor-Green vortex test results comparing Wasatch with Arches and ICE. Each test ran on a single processor using a single thread. In these simulations of the Taylor-Green vortex, Wasatch is faster than Arches and ICE for all domain sizes.

Component	8^3	16^3	32^3	64^3	128^3
Arches	1.3x	2.5x	4.2x	5.3x	5.8x
ICE	1.7x	3.2x	6.2x	9.0x	9.8x

8.7 Weakly scaling Wasatch on Titan

The tests in this section run the same code as in the last section. The difference between the Wasatch tests of the last section and the test of this section is that these tests were run on Titan [25] using various numbers of processors. For these tests, Nebo was only used with its single-core implementation, and communication between processors was handled by Uintah's MPI framework.

Figure 8.8 shows how the Taylor-Green vortex code in Wasatch weakly scales from 32 to 262,144 processors. Each plot denotes a different domain size per processor, ranging from

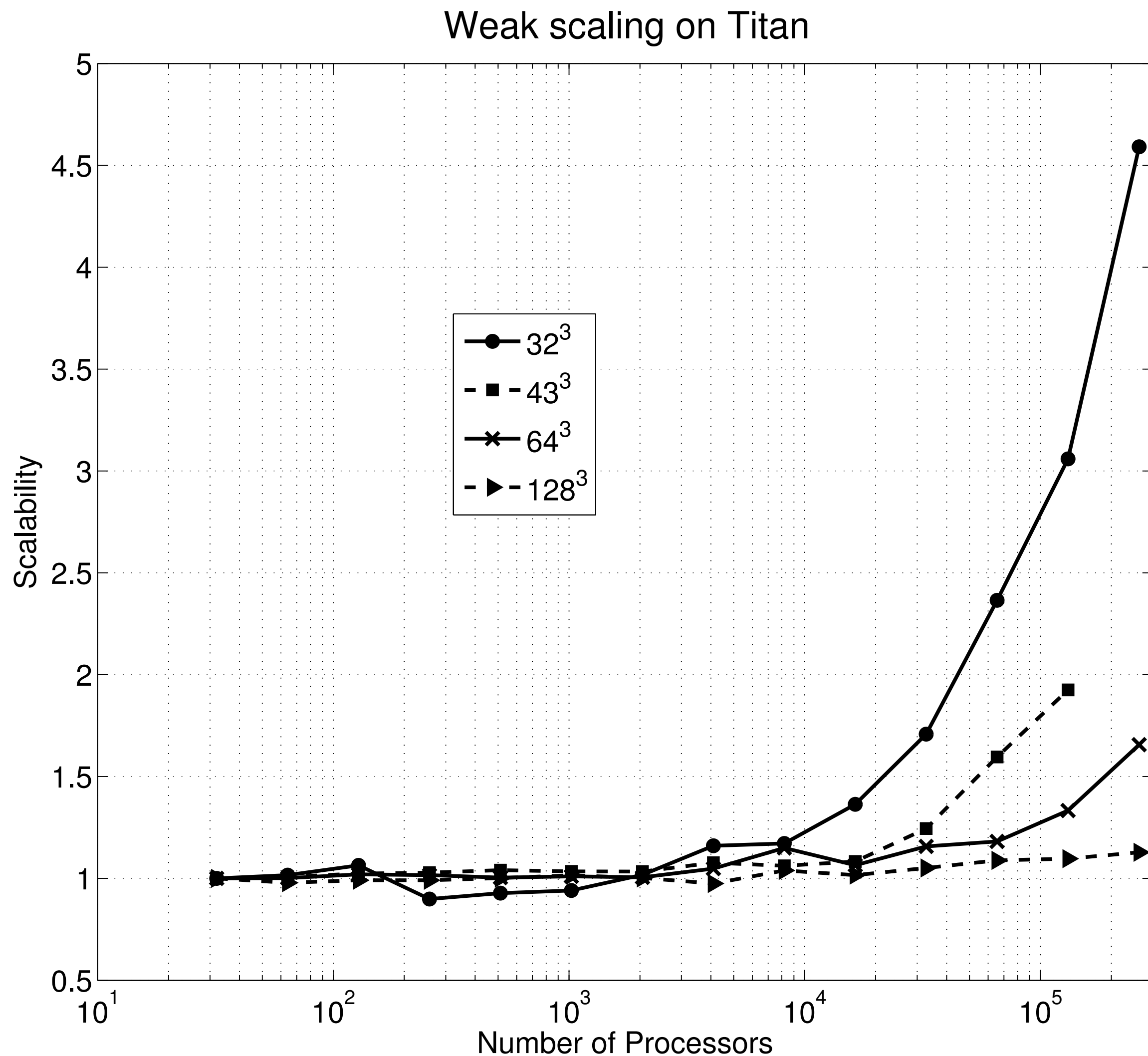


Figure 8.8. Weak scaling of the Taylor-Green vortex on Titan. Values closer to 1 are better: They represent better scaling. With these tests, the problem size per processor was fixed, and the number of processors was increased. Number of processor ranges from 2^5 to 2^{18}.

32^3 to 128^3 elements per processor. Because these tests measure weak scalability, values closer to 1 are better. Thus, for 32^3 elements per processor, weak scaling starts to break down at 4096 processors. For 43^3 and 64^3 elements per processor, weak scaling starts to break down at 32,768 processors. For 128^3 elements per processor, weak scaling stays strong through the largest tests run, with 262,144 processors. Again, as in the last section, we see that Wasatch and Nebo perform better with more elements per processor.

Just as with the tests from the last section, Nebo cannot take all of the credit for these results. Much time and effort has gone into developing Uintah, Wasatch, and Nebo. All three software packages must perform efficiently and scalably for the Taylor-Green vortex problem to scale to 262K processors. For example, if Uintah handled the MPI communication poorly, these tests would not scale. Alternatively, if Nebo had more overhead or was less efficient, these tests may scale even better than they do, by hiding communication latency behind inefficient numeric calculations. However, in this case, efficiency would suffer and Wasatch would not perform favoriably in comparison tests with other components of Uintah. Fortunately, the previous section shows that Wasatch is efficient, and so Wasatch, using Nebo, is scalable and efficient.

8.8 Summary

This chapter presents evidence that Nebo's backends are efficient and useful. Nebo's single-core implementation performs at least as well as the hand-written C++ that it replaces and often better. Nebo's multicore implementation can scale linearly up to six threads. Nebo's many-core (GPU) implementation generally performs at least an order of magnitude faster than its single-core implementation, and can be as much as 37 times faster. Moreover, Wasatch with its heavy use of Nebo compares favorably to Arches and ICE, and can be up to an order of magnitude faster. While Uintah has weakly scaled to 262K processors with other components before, Uintah scaling to that same point with Wasatch and Nebo proves that Nebo (and Wasatch) are efficient and scalable as well.

This chapter concludes the contributions of this dissertation. The next chapter discusses work related to introspective pushdown control-flow analysis and Nebo.

CHAPTER 9

RELATED WORK

9.1 Introduction

This chapter discusses work related to introspective pushdown control-flow analysis as well as work related to Nebo. Section 9.2 discusses other approaches to approximating program behavior, either as a pushdown system or as another context-free model. Section 9.3 discusses other approaches to designing languages and systems for large-scale parallel computation. Section 9.3.1 discusses parallel paradigms, including shared memory, distributed memory, and PGAS along with other models. Section 9.3.2 discusses the various approaches to designing parallel languages. Sections 9.3.1.4.7 and 9.3.2.4 specifically discuss domain-specific languages similar to Nebo.

9.2 Control-flow analysis

The static analysis portion of this dissertation is based on an earlier work: *Introspective pushdown analysis of higher-order programs*, in the Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, © ACM, 2012. http://doi.acm.org/10.1145/2364527.2364576. Included here by permission.

The complete development of pushdown analysis from first principles stands as a new contribution, and it constitutes an alternative development of CFA2. It goes well beyond the ICFP 2012 work (and work on CFA2) by specifying specific mechanisms for reducing the complexity to polynomial time as well. An immediate advantage of the complete development is its exposure of parameters for controlling polyvariance and context-sensitivity. The development of introspective pushdown systems is also more complete and more rigorous. The complete development of pushdown analysis also exposes the critical monotonicity constraint absent from the ICFP 2012 work. More importantly, this work uses additional techniques to improve the performance of the implementation and discusses those changes.

Garbage-collecting pushdown control-flow analysis draws on work in higher-order control-flow analysis [121], abstract machines [47] and abstract interpretation [42].

9.2.1 Context-free analysis of higher-order programs

The motivating work for the pushdown analysis presented here is Vardoulakis and Shivers' [135] very recent discovery of CFA2. CFA2 is a table-driven summarization algorithm that exploits the balanced nature of calls and returns to improve return-flow precision in a control-flow analysis. Though CFA2 exploits context-free languages, context-free languages are not explicit in its formulation in the same way that pushdown systems are explicit in our presentation of pushdown flow analysis. With respect to CFA2, the pushdown flow analysis in this dissertation is also polyvariant/context-sensitive (whereas CFA2 is monovariant/context-insensitive), and it covers direct-style.

On the other hand, CFA2 distinguishes stack-allocated and store-allocated variable bindings, whereas my formulation of pushdown control-flow analysis does not: It allocates all bindings in the store. If CFA2 determines a binding can be allocated on the stack, that binding will enjoy added precision during the analysis and is not subject to merging like store-allocated bindings. While I could incorporate such a feature in our formulation, it is not necessary for achieving "pushdownness," and in fact, it could be added to classical finite-state CFAs as well.

9.2.2 Calculation approach to abstract interpretation

Midtgaard and Jensen [92] systematically calculate 0CFA using the Cousot-Cousot-style calculation-based approach to abstract interpretation [41] applied to an ANF λ-calculus. Like the present work, Midtgaard and Jensen start with the CESK machine of Flanagan et al. [48] and employ a reachable-states model.

The analysis is then constructed by composing well-known Galois connections to reveal a 0CFA incorporating reachability. The abstract semantics approximate the control stack component of the machine by its top element. Midtgaard and Jensen remark monomorphism materializes in two mappings: "one mapping all bindings to the same variable," the other "merging all calling contexts of the same function." [92, p. 293] Essentially, the pushdown 0CFA of Section 2.3 corresponds to Midtgaard and Jensen's analysis when the latter mapping is omitted and the stack component of the machine is not abstracted.

9.2.3 CFL-reachability and pushdown-reachability techniques

This work also draws on CFL-reachability and pushdown-reachability analysis [30, 79, 112, 113]. For instance, ϵ-closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. For my analysis, I implicitly

invoked these methods as subroutines. When these algorithms showed their weaknesses (as with their enumeration of control states), I developed Dyck state graph construction.

CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [90] and type-based polymorphic control-flow analysis [110]. These analyses should not be confused with pushdown control-flow analysis, which is computing a fundamentally more precise kind of CFA. Moreover, Rehof and Fahndrich's [110] method is cubic in the size of the *typed* program, but the types may be exponential in the size of the program. Finally, the technique presented here is not restricted to typed programs.

9.2.4 Model-checking higher-order recursion schemes

There is terminology overlap with work by Kobayashi [77] on model-checking higher-order programs with higher-order recursion schemes, which are a generalization of context-free grammars in which productions can take higher-order arguments, so that an order-0 scheme is a context-free grammar. Kobayashi exploits a result by Ong [105], which shows that model-checking these recursion schemes is decidable (but ELEMENTARY-complete) by transforming higher-order programs into higher-order recursion schemes.

Given the generality of model-checking, Kobayashi's technique may be considered an alternate paradigm for the analysis of higher-order programs. For the case of order-0, both Kobayashi's technique and the one presented in this dissertation involve context-free languages, though mine is for control-flow analysis and his is for model-checking with respect to a temporal logic. After these surface similarities, the techniques diverge. In particular, higher-order recursion schemes are limited to model-checking programs in the simply-typed lambda-calculus with recursion.

9.3 Parallel processing languages for large-scale parallel computation

Computer science attempts at parallel processing languages for large-scale parallel computation have failed to gain traction over the last 20 years because of hidden costs and adopter uncertainty. Creating a language is a fairly inexpensive and fast process; however, building a highly optimized and efficient compiler for a new language is an expensive process, in terms of developers' time. Many language designers consider their work done when a prototype implementation is complete. Thus, many language designers do not spend the time necessary to optimize for large scale computation. Many of the languages discussed below have no current support. Researchers and members of industry interested in adopting new parallel technologies are justifiably concerned that their projects may be orphaned if

the trendy new parallel programming language used by their projects loses support.

The rest of this chapter is laid out as follows: Section 9.3.1 discusses the main types of parallel models used by parallel programming languages. Section 9.3.2 discusses the various methods of designing a new parallel programming model as well as which languages are still actively used. Section 9.3.3 discusses why so many of the languages discussed in Sections 9.3.1 and 9.3.2 have failed to gain traction and why the few that have been successful found success.

9.3.1 Models of parallelism

There are many models of parallel computation. This section divides the models generally upon how each language handles memory. Shared memory models view memory as a single globally addressed entity. Distributed memory models view memory as locally partitioned. Message passing models allow computation units to share information with each other through the use of explicit messages. Data parallel models force all computation units to execute the same instructions. Because of the interrelation of these models, I combine distributed memory, message passing, and data parallel models into a single section. PGAS (partitioned global address space) models combine shared and distributed memory models, for a globally addressed, yet locally positioned memory model. Finally, I discuss other models, which generally abstract memory management from the programmer's direct control in various ways.

9.3.1.1 Shared memory

Within the shared memory model, each processor/computation unit has access to every address in memory. Traditionally, the shared memory model implies uniform access. With uniform access, each address takes as long to access as every other. In practice, caches and prefetchers optimize memory access times for spatial and temporal locality. That is, memory recently accessed or memory near recently accessed memory is faster to access than other memory. For sequential computation, caches and prefetchers can significantly improve performance. For parallel computation, caches and prefetchers can improve performance but also complicate the memory model. A cache keeps a local copy of some memory, which can change independent of any other copies in other caches or in main memory. However, cache coherence for shared memory models is generally handled by hardware.

The explicit or implicit use of computational units, or threads, mainly defines shared memory models. For example, ACE [44], Pthreads [6], and Presto [20] all give explicit control of threads to the programmer. When controlling threads directly, the programmer

must determine when it is best to create and destroy threads as well as how best to manage them. For a diligent programmer, this control can allow better performing code, but it also forces more work onto the programmer.

Implicitly controlled, or rather language/compiler-controlled, threading requires less work from the programmer but also require the compiler to determine the best means for managing the threads. Fork [60], Cilk [53], ParC [55], COOL (Concurrent object oriented language) [36], FX [126], and OpenMP [8] provide parallel structures/syntax for the programmer to use. Generally, these parallel structures/syntax inform the compiler of parallel opportunities but are not required to be executed in parallel or even to be scheduled for parallel execution. Fork and Cilk provide “spawn points” after which execution may be performed in parallel. ParC, among other languages, provides parallel loop structures, that allows different iterations of the loop to be executed in parallel. ParC also provides a split block, in which each statement can be executed in parallel. The function definition syntax of COOL specifies if a function can be executed in parallel or must be computed atomically. Users of the FX library describe tasks, which FX schedules on threadpools that it manages. The OpenMP library allows for both parallel loop execution as well as task descriptions for parallel execution.

Synchronization also varies among languages that use shared memory. For example, COOL uses mutex functions to specify sole access to objects. The Pthread library and ParC provide synchronization primitives, such as barriers, semaphores, and mutex locks. Similarly, Presto provides synchronization classes, which can be expanded through class inheritance.

9.3.1.2 Nonshared memory

A distributed memory model has memory divided into sections, each of which is near a computation unit. This model reflects clusters and networks more accurately than shared memory models. With clusters and networks, memory accesses between physical chips is not identical to memory accesses on the same physical chip. Memory accesses across chips takes longer to complete and must use communication channels that are physically different that on-chip memory accesses. If computation units are to communicate information, it cannot be through memory. Most distributed memory models focus on either eliminating communication or providing mechanisms for computation units to pass information between each other.

The distributed memory models that focus on information sharing, generally use explicit message passing. CSP [64] was an early and influential message passing language, and

Occam [89] is heavily based off of it. MPI [138] is a widely used message passing library with several implementations for several languages. ISIS [24] is a message passing library for Fortran, and Fortran-M [51] adds message passing and task parallelism to Fortran77. MpC [84] implements a subset of MPI as part of its core language. SuperPascal [63] is Pascal [139] extended with message passing. POOL-I [11], Sina [132] and DC++ [34] pass messages between objects. Furthermore, DC++ is built upon C++ and uses special classes for communication, which act like channels, and uses other specialized classes for messages. P++ [85] is also built upon C++ but does not allow inheritance from its channels and messages. V [81] uses streams over shared or distributed memory models.

ACT++ [70] and Joyce [62] use the actor model. In the actor model, everything is an actor, which acts based upon the messages it receives. Computation follows a cascade of messages. Actors are concepts more general than threads because they can respond to multiple messages at once in parallel. Thus more than one thread may be needed to implement a single actor.

Charm [71] and Charm++ [73] use chares for message passing. A chare is similar to an actor; however, a chare must sequentially process a single message at a time. Thus a single thread can implement a chare.

The distributed memory models that attempt to eliminate communication usually focus on data parallelism for problems where there are no or few dependencies between data. Thus different computation units can run the same computation on different data. This parallelism is very similar to the loop parallelism in the shared memory model languages, such as pC++ [28]. However, generally data parallel models still provide some form of communication for problems with data dependencies. In the strictest data parallel models, all the computation units must execute the same instruction in lock-step. Generally, though, data parallel models only force the computation units to execute the same block of instructions or function.

P3L (Pisa Parallel Programming Language) [16] models both data and task parallelism. Data parallelism comes through explicit parallel loop and functional map constructs. Task parallelism comes through the pipe functionality, which essentially chains computational blocks together creating streams of computed values.

DOME [13] is an object oriented language that defines parallelism through classes that contain arrays of data. Similarly, pC++ [28] and Correlate [116] extend C++ and Java, respectively, with data parallel constructs. Fortran D [52] adds data parallelism to Fortran77.

NESL [26] allows for nested data parallelism. Data parallelism is nested when data parallel functions can be performed in parallel. NESL also builds measures of the complexity of the work and the depth of the nested data parallelism, which helps to estimate the amount of parallelism for analysis.

C* [117] provides the ability to define shapes, which are regularly- or irregularly-shaped arrays. C* decomposes the user specified shapes to exploit data parallelism. The compiler then handles any information sharing that arises from the decomposition of the shapes. Vic* [40] extends C* with virtual memory. While not directly related, C** [83] extends C++ with classes called aggregates that are essentially shapes from which other classes can inherit.

9.3.1.3 PGAS

Partitioned global address space (PGAS) is the best of both shared and distributed memory models. Programs can uniformly access memory on hardware that is physically distributed. Uniform memory access simplifies the structure of programs. Compilers are able to hide the difference between local and nonlocal memory accesses. However, nonlocal (off-chip) memory accesses must still travel longer distances than local (on-chip) memory accesses and therefore have more latency. This latency directly hinders performance. Thus several PGAS languages, especially the later ones, distinguish between local and nonlocal memory in the type system. This type-system distinction on memory allows memory accesses to have a uniform syntax, while still forcing the programmer to consider the latency. Furthermore, the type-system distinction allows compilers to better optimize local memory accesses, because there does not have to be a runtime check for nonlocal memory access.

9.3.1.3.1 Early PGAS languages. Many PGAS languages predate the terminology, "partitioned global address space." These languages describe themselves as working on distributed memory systems while providing the syntax of shared memory systems. Split-C [86], XPC (explicitly parallel C) [109], and UC [17] extend C with global pointers and synchronization primitives. CC++ [38] and MPC++ similarly extend C++ with global pointers and synchronization (inheritable) classes. KOAN (Fortran-S) [82] extends Fortran77 with a virtual shared memory. Likewise, HPF (High Performance Fortran) [1, 2] extends Fortran90 and Fortran95 with a virtual shared memory. Modula-2* [131] extends Modula-2 with a virtual shared memory. Also, Midway [21], Millipede [66], TreadMarks [12], Lparx [80], and Orca [18] all provide a virtual shared memory space. Beyond just providing a virtual shared memory, Munin [19] analyzes the types of nonlocal memory accesses. Munin uses this analysis to intelligently select different distributed information sharing techniques

for better performance. DSM-Threads [99] can replace Pthreads [6] in existing applications to use these applications in a distributed memory setting with virtual shared memory. GA (Global Arrays) [102] is a library for Fortran, C, C++, and Python and provides virtual shared memory and automatically distributed arrays.

Not all early PGAS-like languages provided uniform memory access. Some provided a mix of shared memory in certain contexts and distributed memory information sharing techniques everywhere else. While this two-level memory distinction disqualifies these languages from being true PGAS languages, they represent a first step towards unifying shared and distributed memory models. The desire for unification began with the use of multicore chips in clusters. Here traditionally shared memory devices were being used in distributed memory settings. Without some way of addressing the difference, such systems must use distributed models on the multicore chips, which introduces unnecessary latency for communication between computation units on the same chip. Thus, these hybrid systems were proposed. HPC++ (High Performance C++) [69], ABC++ (Abstract Base Class) [14] and QPC++ (Quasi-Parallel C++) [29] extend C++ with shared memory within objects but message passing between objects. Interestingly, QPC++ also allows variables to be shared between objects when the variables are explicitly scoped and labeled as shared. Nexus [49] is a library communication between CC++ [38] and Fortran-M [51]. Nexus adds its own concept of shared versus distributed memory through contexts. Within a context a shared memory model is used, and message passing is used between them.

9.3.1.3.2 Modern PGAS languages. There are also several relatively recent languages that use PGAS explicitly and intensionally. Co-Array Fortran [104] extends Fortran95 with PGAS. Co-Array Fortran creates images, which are essentially data parallel copies of the program. Synchronous primitives are included to simplify communication between images. Co-Array Fortran has become part of standard Fortran (as of the 2008 standard) [111].

UPC (Unified Parallel C) [4] extends C with PGAS. UPC uses explicit threads for parallelism as well as explicit synchronization primitives. UPC requires variables that are to be shared between threads to be explicitly marked with the keyword, shared. For shared variables, UPC provides the same pointer arithmetic as local (standard C) pointers and extra support for programmers to discover what parts of shared variables are local. Currently there is ongoing research with and support for UPC, unlike most of the other languages that extend C with parallel capabilities.

Titanium [141] extends Java with PGAS. Titanium uses threads to model parallelism.

Titanium does not require any special annotations for variables and data to be shared across threads. Titanium is easily ported between systems of various memory layouts. However, some systems, especially distributed memory ones, generally need tuning for optimal performance, which is usual for distributed memory systems. Titanium, unlike most other parallel languages, provides garbage collection, which it inherits from Java. The Titanium project appears to no longer be maintained, but some of the applications built upon Titanium are still used for research. Thus Titanium lives on in one form or another.

X10 [119] uses syntax similar to C and C++ but does not directly inherit from either language. X10 divides data among places, which act very similar to threads. Like threads, places can pass messages to each other to simplify communication. X10 provides activities, which asynchronously execute statements, which threads generally cannot do. For synchronization, X10 provides clocks, which are dynamic barriers, and hierarchical relationships between activities. One activity can spawn a child activity and thus become the new activity's parent. A child activity cannot wait for its parent to finish. However, a parent can wait for its children to finish. X10 also provides support for certain blocks of statements to be executed on NVidia's GPUs, through a CUDA backend. X10 also provides syntax for compiler annotations, which allows programmers to selectively use compiler add-ons, for specialized features and optimizations. This ability to grow the language through annotations is rather unique among parallel languages. X10 was created and is currently supported by IBM.

Chapel [9] is a procedural PGAS language with syntax common to many languages, such as C++, Fortran, and Matlab. Chapel defines a locale as its local memory unit/basic computational unit. Chapel is designed around the idea of incremental implementation. Incremental implementation provides high-level simple commands to perform common complex tasks and allows for gradual replacement of those commands with more specialized and more detailed commands, if and when better performance is required. Incremental implementation also guides data allocation among locales: Easy and fast-to-code commands for data allocation exist, as do complex, detailed, and optimized commands for the same tasks. Similar to X10's activities, Chapel uses task-based constructions to exploit task parallelism. Chapel is supported by Cray, and is under steady if slow development. To wit, version 0.4 was released in 2005 [3], and the current version, 0.91 was released last year [9]. There is no information as to when the first full version (1.0) will be released.

Fortress [10] is a PGAS language that incorporates many recent concepts and designs from the programming language research community. Fortress is object oriented but uses

traits instead of inheritance. Fortress is statically typed and can use type inference to determine the type of various statements and expressions. With type inference, programmers can skip specifying many of the type annotations that otherwise generally plague statically typed programs. Fortress also employs components, which are essentially modules. A module or component defines functions, objects, types, and values which can be exported to other modules for use. Thus a module is a lot like a user-defined library. In fact, many libraries are written as modules when supported by a given language. However, unlike most nonmodule libraries, modules can selectively chose what definitions they wish to provide, thus allowing global definitions in the module to be private to it. While all of these features make Fortress attractive to potential users, Fortress has come to a crucial and potentially fatal point in its development. Fortress was originally designed by Sun Microsystems and has been maintained by Oracle. Unfortunately, Oracle just last year announced that it decided to pull support from Fortress [124]. Loss of corporate support is not the end of Fortress, but Fortress must now rely on its open source community for long-term support and use. Since Fortress began as a corporate project, the open source community around Fortress is largely untested and now must for the first time bear Fortress's full support commitments. Fortress's future is very unclear at the moment.

It is worth noting here that the last three languages discussed, X10, Chapel, and Fortress, all were begun by corporations as part of DARPA's HPCS (High Productivity Computing Systems) project [88]. This project lasted until 2010, although corporate support continued afterward. It is not surprising that these three languages are the biggest and most interesting projects of their kind in the last few years. However, only time will tell if these projects are directly useful or through derivative languages indirectly useful to future parallel programming needs.

9.3.1.4 Other models

In this section, I discuss parallel processing languages that do not fit into the previous categories. First, I will discuss a few unique languages and then some small categories, all of which handle memory quite differently than the above languages.

μCRL (*micro*-Common Representation Language) [58] is a language built to mathematically study communicating processes. Alternatively, μCRL can also be described as an algebraic language of communicating processes. The syntax and semantics of μCRL essentially represents a term rewriting system. This approach to syntax and semantics translates very directly into mathematical notation for proofs of correctness, termination, and properties. Of course, μCRL is Turing-complete and as such generally is not decidable.

However, under certain well-defined conditions, μCRL is terminating. This result and other results about program properties makes μCRL interesting to those studying the computational theory behind parallel programs. Conversely, μCRL is not designed for ease of use or efficiency and thus is not interesting to those wishing to build applications not relating to the computational theory behind parallel programs.

Mentat [57] is a language loosely based on C++, designed to explore how far compiler-driven parallelism can be taken. Mentat's basic work flow is as follows: A programmer writes object-oriented yet sequential code and provides hints/annotations about which computations, tasks, functions, and data are worth parallelizing. The compiler then analyzes the code and determines all data and task dependencies. Using this dependency information and the programmer's annotations, the compiler automatically parallelizes the code. Theoretically, this heavy reliance on the compiler is ideal. The programmer focuses on what is to be computed and hints what is difficult and contains parallelization potential, while the compiler focuses on how to do the computation. If the compiler is properly built, it could support any hardware configuration and efficiently map the computation to the hardware's strengths without any extra input or tuning from the programmer. However, in practice, a compiler of this sort is limited to what it has been designed to analyze. Thus the compiler is limited to the experience and knowledge of the designers and implementors of the compiler. That said, no other language focuses as much as Mentat does on both ease of programming and performance optimization. The other languages focus on providing easy programming at the cost of performance or high performance at the cost of dealing with minute but influential details that complicate the source code.

Unity [37] is a pure research language of a very different vein than μCRL. Where μCRL focuses on mathematical rigor, Unity focuses on convergence theory. A program written in Unity is primarily composed of a series of statements. These statements are executed randomly in parallel repeatedly. The computation continues infinitely until a fixed point is reached. In this context, a fixed point is when executing any statement does not change anything, i.e., no data/state change. Unity models infinite parallel computations as well as parallel fixed point computations. Much like μCRL, Unity's primary interest is theoretic and not practical. Unity was implemented as a teaching tool, and not as a language for practical applications [56].

Falcon [101] focuses on providing a common, broad and efficient toolbox of techniques and styles. Falcon contains six programming language paradigms: Procedural, functional, class-based object orientation, prototype-based object orientation, message orientation, and

tabular. With this collection of paradigms, a programmer can use procedures, first-class functions, inheritable objects, dynamically composed objects, messages, or table-based computing. Rather than restricting programmers to one or two paradigms, Falcon allows programmers to pick the right paradigm—or tool—for the current task or subtask without changing languages. Falcon provides only limited parallelism through its message passing mechanism, but its approach to programmer freedom is what makes Falcon interesting.

9.3.1.4.3 Coordination languages. Coordination models use a logical tuple-space, or global heap, that coordinates computation units through their interaction with the tuples. Coordination models can be considered an indirect message passing model. Linda [54] is the original coordination language, and as such all the other languages in this section derive from Linda. Objective Linda [76] replaces the tuple-space with a global object store. The main difference between Objective Linda and classic Linda is that methods are now passed along with data. Structured Dagger [72] combines Linda with Charm [71] to add direct communication between threads/computation units. Eilean [35] is an implementation of Linda that uses MPI for its basic communication primitives. ISETL-Linda [118] combines the ISETL approach to imagine processing with Linda-based communication. ParLin [122] combines C with Linda-based communication.

9.3.1.4.4 Graph-based languages. Almost all the languages discussed so far in this chapter focused on data parallelism: The placement of data, communication of data, etc., etc. Unlike those languages, graph-based languages focus mostly on task parallelism. Graph-based languages decompose computation into graphs, where edges are data-dependencies. Compiler-based schedules are able to use the graph structure to determine what nodes of computation are ready to execute. Similarly, these schedulers can focus on critical paths and bottlenecks to increase parallelism. GLU [67] is a rather straightforward graph-based language. Jade [115] is a shared memory extension to C that uses task graphs to exploit task parallelism. Code [100] is an Ada/C based graph-based language that employs lazy evaluation. Lazy evaluation is a model of evaluation where a value is only evaluated when it is actually needed for another computation. PSDM (Parallel Software Design Model) [74] is a graph-based language that tightly ties task nodes to processors.

9.3.1.4.5 Logic and functional parallel languages. The best known logic programming language is Prolog. Logic programming is declarative. Facts, rules, and information are declared. Statements posed as queries then drive computation. The logical system tries to prove or disprove these queries. There is no notion of memory or state in logic programming. Parlog [39], Strand [50], and Aurora [87] are all Prolog-derived languages

that parallelize the logical proof system.

Functional languages pose computation as the composition of functions with no (or little) global state. Even with global state, there is no notion of memory in a functional language. Lisp is one of the oldest functional languages, and multilisp [61] is a parallel variation of lisp. Multilisp introduces the concept of futures. A future is the result of a concurrently-performed computation/evaluation that may or may not be ready. Any use of a future before it is ready blocks. Also, multilisp introduces a parallel function call that causes parallel execution/evaluation of the given call's arguments.

9.3.1.4.6 GPU languages. GPUs are hardware processors that are specifically designed to work over massive amounts of data in parallel. Unlike traditional CPUs, GPUs tend to have a large number of registers, lock-stepped instruction execution across many processors, and explicitly managed memory and cache. CUDA [5] is a low-level language designed by NVidia, the makers of a large portion of GPUs on the market. CUDA attempts to be the C of GPUs, with a great deal of processor control and optimization directly given to the programmer. CUDA can be used within C, C++, and Fortran. On the other hand, OpenCL [7] focuses on a broader range of processors, targeting both CPUs and GPUs.

9.3.1.4.7 Domain-specific languages. A recent approach to designing parallel processing languages is to limit the scope of the language to a specific domain or area of application. On one hand, limiting a language to a single application does have the obvious drawback of restricting the potential users of a language to those interested in its target domain. On the other hand, a new domain-specific language can focus on the abstractions and the potential parallelism of its domain. Nebo clearly fits into this category.

There are many other DSLs that have functionality and domains similar to Nebo, but none contain all of Nebo's features. POOMA [114] and Blitz++ [137] are the only other DSLs to allow incremental adoption in the same general domain. POOMA provides support for message-passing and thread-based parallelism but not GPU support. Out of all the languages discussed in this chapter, Blitz++ does not support parallel execution but is included because of its similarity to Nebo.

Liszt [43] solves PDEs by abstracting based on geometry and spatial reasoning rather than mathematical equations as Nebo does. While Liszt supports both CPU- and GPU-based parallelism, Liszt generates only code that supports one type of parallelism at a time. Furthermore, Liszt does not support incremental adoption.

DSLs, such as OptiMesh [128] and OptiML [127], developed with the Delite compiler [33], offer CPU- and GPU-based parallel backends within the same runtime environment, like Nebo. The Delite language/compiler is a metadomain-specific language/tool. Delite fa-

cilitates the creation of DSLs that can interact with other DSLs created within Delite. OptiMesh uses the same abstractions and much of the same syntax as Liszt for solving PDEs. In general, OptiMesh performs better than Liszt because Delite provides support for more aggressive optimizations, and OptiMesh is able to work with other Delite-based DSLs. OptiML is a domain-specific language for machine learning.

The Pochoir stencil compiler [129] supports stencil calculations very similar to the stencils Nebo provides. However, the shape of Pochoir's stencils are not polymorphic, as Nebo's are. (See Section 7.3 for more information on Nebo's polymorphic stencils.) Pochoir does more optimizations than Nebo; however, Pochoir must analyze the entire time-step function, which currently limits it to simple time-step functions. By comparison, Wasatch regularly runs time-step functions that use dozens and sometimes hundreds of variables.

Pochoir and DSLs from Delite support forms of incremental adoption. Generally, partial adoption of these languages requires extra code/syntax for DSL and host code to interact. Partial adoption for these languages does require changing the build system to use a new language-specific compiler. In comparison, Nebo works with the host representation of data (and so does not require interface code) and is included with a header file as any C++ library is.

9.3.2 Language design approaches

When designing a new parallel programming language, there are four basic approaches that language designers take. First, they can design a novel language, inheriting little if anything from previous languages. Second, they can heavily borrow syntax and model designs from existing languages. Often heavily relying on another language means creating a new language that contains the original as a subset of the new language. Third, they can design libraries that are linked into existing languages' syntax and compilers. The second and third approaches are very similar but can be distinguished by implementation. When a new parallel language is contains another established language as a subset (second approach), often the new language features and semantics require the implementation and use of a new compiler and tool chain for the new parallel language. When a new parallel language is designed as a library for use with existing languages (third approach), the new language features and semantics require only modifying existing compilers and tools, often requiring only extra compiler flags to enable the new parallel language. Fourth, language designers can build a domain-specific language. A new language targeting a specific domain generally takes one of two paths: A new domain-specific language can stand alone or be embedded in another language. A stand-alone domain-specific language generally is a novel

language with its own compiler. An embedded language, like Nebo, either expands an existing language or is linked in like a library.

Each of these four approaches has its own benefits and drawbacks. It should be noted that languages listed as no longer active, are listed so because they have had no recent publications, recent support updates, or publicly known applications currently in use. Thus a language listed as inactive may be active, but it is unlikely.

9.3.2.1 Novel language approach

Novel language design is mainly useful for language designers who are building a language on a novel parallelism model or a very new one. μCRL [58], CSP [64], GLU [67], Linda [54], Unity [37], Code [100], Falcon [101], PSDM [74], and Chapel [9] all fit into this category. Out of these, only Falcon and Chapel are still in use. It should be noted, however, that Linda still has derivative languages that are still in use.

9.3.2.2 Language extension

Language extension is a better option for a language designer. The language's users, that is programmers, will need to learn less to adopt the language.

There are at least 15 languages that derive from C [75]: Split-C [86], Jade [115], Midway [21], MpC [84], ParC [55], UC [17], C* [117], Vic* (builds on C*) [40], Charm [71], Cilk [53], Fork [60], ParLin (combines C and Linda) [122], XPC [109], UPC [4], and OpenCL [7]. Out of these 15, Cilk, UPC and OpenCL are clearly still supported and actively developed.

There are at least 19 languages that derive from C++ [125]: ACE [44], Correlate [116], HPC++ [69], pC++ [28], CC++ [38], MPC++ [65], ABC++ [14], Mentat [57], Presto [20], Munin (builds on Presto) [19], Parallel-C++ [68], DC++ [34], QPC++ [29], C** [83], Charm++ (builds on Charm) [73], COOL [36], DOME [13], P++ [85], and P3L [16]. Out of these 19, ACE is still used in industrial applications.

Java is the basis of: A version of ACE [44], Correlate (shifted from C++ toward the end of its support) [116], Titanium [141], and DPJ (Deterministic Parallel Java) [27]. Of these, ACE and DPJ are still used in industrial applications.

Fortran had at least six derivatives: KOAN/Fortran-S [82], Polaris [107], Fortran-M [51], Co-Array Fortran [104], Fortran-D [52], and HPF [2]. Out of these, Co-Array Fortran is part of the latest Fortran standard [111], and HPF is still actively used and supported.

None of the following parallel derivative languages are known to be currently used or supported: Modula-P [32], Modula-2* [131], and Orca [18] derived from Modula-2.

NESL [26] derives from ML. Linda [54] is the basis of Objective Linda [76], Structured Dagger (builds on Charm and C) [72], Eilean (uses MPI) [35], and ISETL-Linda [118]. Multilisp multilisp [61] derives from Lisp. Prolog is the basis of Parlog [39], Strand [50], and Aurora [87]. CSP [64] is the core of Occam [89]. SuperPascal [63] is a superset of Pascal.

9.3.2.3 Language as library

Next, a language designer can build their parallel model as an library to be added onto both existing languages and existing compilers. This approach will not work for parallel models that are great departures from traditional programming paradigms. However, if the parallel model can be captured in a library, this is generally a language designer's best chance of success. Rather than tying their design to a specific definitive language, a library allows any language/compiler community to pick it up and implement a convenient interface for their language. Thus a language designer can build a prototype of their library as just a proof of concept, and let interested communities build the practical and useful implementations onto already successful compilers and languages.

MPI [138], OpenMP [8], and Pthreads [6] are probably the most successful libraries and works with C, C++, and Fortran. CUDA [5], while not technically a library, behaves very much like a library when interacting with C, C++, and Fortran and is used with most GPUs. The Meta-Chaos library [46] allows HPF [2], pC++ [28] and other languages to communicate in the same application; however, this library is no longer in use. POOMA [114] and POET [15] are libraries for C++ that are now defunct but did spark greater interest in using C++ template metaprogramming for parallel applications.

9.3.2.4 Domain-specific languages

As mentioned above, a domain-specific language can either be embedded within an existing language, or it can stand alone with its own compiler. POOMA [114], POET [15], Blitz++ [137], and Nebo are all embedded within C++. Stand-alone domain-specific languages, such as Liszt [43], OptiMesh [128], and Pochoir [129], have their own compilers, which often target other common languages, especially C, CUDA, and OpenCL. Entire applications can be written in stand-alone domain-specific languages, whereas embedded domain-specific languages generally require use of the host language for many tasks not directly related to the target domain. Since many stand-alone domain-specific languages are compiled to a common language, such as C, they can be used in existing domain projects by being compiled separately and then linked into other parts of the project. Adding a

stand-alone domain-specific language to an existing project requires the addition of a compiler to the project's build system as well as refactoring code to match the domain-specific language's abstractions and capabilities. Adding an embedded domain-specific language to an existing project requires the project to use the host language and the abstractions of the domain-specific language. All of the languages other than POOMA and POET are currently under development.

9.3.3 Failure to gain traction

Out of all the languages discussed in this chapter, only a few are in use today, and many of those are recent language that are still in their initial development effort. So why did most of these languages fail? The most prominent reason is lack of compiler support. Many of these languages and libraries never were implemented as more than prototypes and proofs-of-concepts. If there is no working compiler for a language, no one will write meaningful and useful applications in it for that simple reason.

Furthermore, the more complex the parallel model, language, and resulting implementation are, the more difficult the work to build a useful and efficient implementation. Even for the language designers, in many cases, it is not worth the money, the time, or the effort to build a useful and efficient implementation.

Next, even when there is an actively supported compiler, future support is not guaranteed. For example, POOMA [114] and POET [15] lost their institutional support[1] just about the time that interest was growing in them. Likewise, both ACE [44] and DPJ [27] have no active support but do have active applications based upon them. This climate of short-lived parallel languages is bad for adoption or increased support. Application developers become wary of using any new language, even if it has a currently supported useful and reasonably efficient compiler, if they fear that support will vanish while they still need to support their application. Likewise, open source contributors become wary of volunteering their time and effort to supporting a compiler project that may disappear in the next few years.

Finally, there was a conceptual fallacy among many of the language and model designers of the 1990s. Many designers felt that previous parallel languages had failed because their model was flawed, too detail-oriented, or not detail-oriented enough. Thus these designers were convinced that their superior model would solve the parallel programming problem. This attitude was apparent in the literature from the various designers' discussions of other previous parallel languages that had already failed or were in decline. For example, the

[1] Both were developed at National Laboratories.

designers of HPC++ [69] felt justified in creating another C++-based parallel language because in their number where designers of pC++ [28], CC++ [38], and MPC++ [65], all of which had already failed. With their combined learned lessons from these projects they would create a new language that avoided past mistakes in HPC++'s design. However, HPC++ ended up almost indistinguishable from these other languages.

9.4 Summary

While Section 9.3 is far from exhaustive of the parallel processing languages that have been developed over the last 20 years, it is a representative sample of these languages. Despite the mass of defunct parallel processing languages, the final message is not all doom and gloom. Languages and libraries, such as CUDA [5], OpenMP [8], MPI [138], Pthreads [6], and OpenCL [7] have strong and vibrant communities supporting them. Everyone heavily using these languages and libraries has an invested interest in keeping these projects supported and active. CUDA provides a clear and strong financial incentive for NVidia to continue supporting the project. Other projects, such as OpenMP and MPI, have a broad base of users, many of whom are willing to donate time and effort, in varying amounts, to supporting the underlying implementations.

These successful projects do pose something of a chicken-and-egg problem: A successful project will sustain itself making it more successful. However, there are a few commonalities among these libraries. All fit rather seamlessly into existing successful compiler tool-chains. None of them require a fundamental shift in the core paradigm of the host language. Language extensions that do shift the core paradigm of the host language require vast refactoring to add them to existing compiler projects. Also, the successful parallel libraries provide a mix of low-level control and simple structures. True, these simple structures can easily create hard-to-detect data races, misaligned synchronous checks, and other common parallel pitfalls. But these simple parallel structures do allow high-performance applications to take advantage of whatever performance gains are available by hand-writting optimizations.

Domain-specific languages are better suited to meet the challenges general-purpose parallel processing languages face than the general-purpose languages themselves. The cost of creating a new compiler can be mitigated by the language being embedded in an established language or by the domain-specific language's compiler generating code for an established language. With less work needed to produce a production-ready implementation of a language, it is far easier to produce and then maintain a domain-specific language than a general-purpose parallel processing language. Most domain-specific languages allow some

form of incremental adoption, which in turn reduces the cost of adopting these languages. That is, these languages can be used in pre-existing projects in their target domain. Many domain-specific languages target multiple architectures, such as CPUs and GPUs. Without some sort of abstraction over multiple architectures, current and future projects will need to maintain implementations of the same functionality for each architecture. Domain-specific languages with abstractions well-suited for their domains are the future for large-scale parallel computation.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

10.1 Introspective pushdown analysis

Introspective pushdown control-flow analysis approximates programs directly as pushdown systems, unlike CFA2 which uses a summarization technique to handle changes to the stack. CFA2's summarization technique obscures the stack, making CFA2 incompatible with static analysis techniques which need to reason about the stack. Abstract garbage collection is one such technique that requires access to the stack. Introspective pushdown control-flow analysis allows for the power of CFA2 to be combined with abstract garbage collection. The results in Chapter 5 show that pushdown control-flow analysis and abstract garbage collection are powerful static analysis tools in their own right. However, these results also show that there can be great benefit to using these two tools together by improving the precision of the analysis as well as lowering the execution time beyond what each tool can do on their own. In practice, more programs are tractable with the two techniques together, which means that the compilation of more programs can benefit from static analysis. Unfortunately, these algorithms are still exponential. Section 3.5 presents an algorithm with polynomial-time complexity ($O(n^6)$) for pushdown control-flow analysis. This polynomial-time algorithm can be used on programs that are intractable with the exponential-time algorithms, and the results would still benefit from perfectly precise return-flow.

Looking forward, research on introspective pushdown control-flow analysis is progressing in two intertwined directions. First, further improvements to precision and more informative techniques are being developed. For example, CFA2 includes a way of allocating local variables on the abstract stack frame to avoid merging in the abstract store (heap). With the summarization technique, CFA2 needs this local stack allocation. With an explicit pushdown system, this local stack allocation is not necessary but would often be useful. Similarly, techniques for more informative uses of pushdown control-flow analysis are being investigated. The main goal of static analysis is to understand algorithmically program behavior. Algorithmically understanding program behavior leads to two benefits. Compilers

can check for undesirable behavior (bugs and malware) and can check for inefficient behavior, for which there is a known equivalent version that is more efficient. Traditionally, static analysis has focused on mistakes, such as bugs and inefficient code. However, as more code is written and made publicly available it is becoming increasingly difficult to trust the source of the code and the code itself. Thus static analysis, and pushdown control-flow analysis in particular, are being used to detect malware in untrusted code.

10.2 Nebo

Nebo is an expressive, portable, efficient, and scalable language for numerically solving partial differential equations. Nebo's declarative syntax is shorter and easier to read and to maintain than the hand-written C++ that Nebo code replaces. Nebo's multiple functioning backends prove that Nebo is architecture-independent. Comparisons between Nebo and the code it has replaced in Wasatch as well as comparisons between Wasatch and other components of Uintah proves Nebo's efficiency. The Uintah scaling tests prove that Nebo with a good framework can scale to 262K cores.

While Nebo is all of these things, Nebo is not a static language but is continuing to change and expand. Nebo is expanding on three fronts. First, new architectures are being studied to see if they would be a good fit for Nebo and Wasatch. For example, Intel's new Xeon Phi many-core chip, in theory, would be a good fit for Nebo. However, it is not clear yet if the Xeon Phi and its associated software is mature enough to be useful, nor is it clear if any of Wasatch's end users want to run their code on Xeon Phi chips.

Second, since Nebo has proved that it can run efficiently on GPUs, Wasatch developers have become more interested in using Nebo everywhere possible. For Wasatch to use GPUs efficiently, as many numeric calculations must be moved to the GPU as possible. There are many calculations that Nebo cannot handle. To support Wasatch developers in their quest to move everything to GPUs, new features for Nebo are under development. For example, mask/filter operations and a generalized catamorphism[1] are currently being developed.

Finally, as seen from the code in Section 7.4, the loops that Nebo generates for execution are very simple and unoptimized. Thus far, this code, optimized only by the C++ compiler in use, is good enough. In fact, the case studies of Chapter 8 show that Nebo's performance is very good, especially compared to similar projects that do not use any domain-specific languages. Despite the good performance Nebo has now, further improvements can and

[1]This generalized catamorphism in theory could handle arbitrary procedures; however, it is not yet known how many such procedures would be executable on a GPU. This generalized catamorphism will work on Nebo's other backends without issue.

will be made. For example, Nebo currently does not make use of any SSE instructions in its CPU backends. Given the nature of computations in Nebo, SSE instructions could significantly increase performance on some architectures.

APPENDIX

PUSHDOWN PRELIMINARIES

The literature contains many equivalent definitions of pushdown machines, so I adapt definitions from Sipser [123].

A.1 Syntactic sugar

When a triple (x, ℓ, x') is an edge in a labeled graph:

$$x \overset{\ell}{\rightarrowtail} x' \equiv (x, \ell, x').$$

Similarly, when a pair (x, x') is a graph edge:

$$x \rightarrowtail x' \equiv (x, x').$$

I use both string and vector notation for sequences:

$$a_1 a_2 \ldots a_n \equiv \langle a_1, a_2, \ldots, a_n \rangle \equiv \vec{a}.$$

A.2 Stack actions, stack change and stack manipulation

Stacks are sequences over a stack alphabet Γ. To reason about stack manipulation concisely, I first turn stack alphabets into "stack-action" sets; each character represents a change to the stack: Push, pop or no change.

For each character γ in a stack alphabet Γ, the *stack-action* set $\Gamma_{\pm}$ contains a push character γ_+; a pop character γ_-; and a no-stack-change indicator, ϵ:

$$\begin{aligned} g \in \Gamma_{\pm} ::= {} & \epsilon && && \text{[stack unchanged]} \\ \mid {} & \gamma_+ && \text{for each } \gamma \in \Gamma && \text{[pushed } \gamma\text{]} \\ \mid {} & \gamma_- && \text{for each } \gamma \in \Gamma && \text{[popped } \gamma\text{]}. \end{aligned}$$

In this dissertation, the symbol g represents some stack action. In Haskell, we can turn any data type in a stack-action alphabet:

```
data StackAct frame = Push { frame :: frame }
                    | Pop  { frame :: frame }
                    | Unch
```

Chapter 4, which develops introspective pushdown systems, uses the following formalisms for easily manipulating stack-action strings and stacks. Given a string of stack actions, we can compact it into a minimal string describing net stack change. I do so through the operator $\lfloor\cdot\rfloor : \Gamma_{\pm}^{*} \to \Gamma_{\pm}^{*}$, which cancels out opposing adjacent push-pop stack actions:

$$\lfloor \vec{g}\,\gamma_{+}\gamma_{-}\,\vec{g}\,' \rfloor = \lfloor \vec{g}\,\vec{g}\,' \rfloor \qquad\qquad \lfloor \vec{g}\,\epsilon\,\vec{g}\,' \rfloor = \lfloor \vec{g}\,\vec{g}\,' \rfloor,$$

so that $\lfloor \vec{g} \rfloor = \vec{g}$, if there are no cancellations to be made in the string $\vec{g}$.

We can convert a net string back into a stack by stripping off the push symbols with the stackify operator, $\lceil\cdot\rceil : \Gamma_{\pm}^{*} \rightharpoonup \Gamma^{*}$:

$$\lceil \gamma_{+}\gamma'_{+}\ldots\gamma_{+}^{(n)} \rceil = \langle \gamma^{(n)}, \ldots, \gamma', \gamma \rangle,$$

and for convenience, $[\vec{g}] = \lceil \lfloor \vec{g} \rfloor \rceil$. Notice the stackify operator is defined for strings containing only push actions.

A.3 Pushdown systems

A *pushdown system* is a triple $M = (Q, \Gamma, \delta)$ where:

1. Q is a finite set of control states;
2. Γ is a stack alphabet; and
3. $\delta \subseteq Q \times \Gamma_{\pm} \times Q$ is a transition relation.

The set $Q \times \Gamma^{*}$ is called the *configuration-space* of this pushdown system. I use $\mathbb{PDS}$ to denote the class of all pushdown systems.

For the following definitions, let $M = (Q, \Gamma, \delta)$.

- The labeled *transition relation* $(\longmapsto_{M}) \subseteq (Q \times \Gamma^{*}) \times \Gamma_{\pm} \times (Q \times \Gamma^{*})$ determines whether one configuration may transition to another while performing the given stack action:

$$(q, \vec{\gamma}) \overset{\epsilon}{\underset{M}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\epsilon}{\rightarrowtail} q' \in \delta \qquad \text{[no change]}$$

$$(q, \gamma : \vec{\gamma}) \overset{\gamma_{-}}{\underset{M}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\gamma_{-}}{\rightarrowtail} q' \in \delta \qquad \text{[pop]}$$

$$(q, \vec{\gamma}) \overset{\gamma_{+}}{\underset{M}{\longmapsto}} (q', \gamma : \vec{\gamma}) \text{ iff } q \overset{\gamma_{+}}{\rightarrowtail} q' \in \delta \qquad \text{[push]}.$$

- If unlabelled, the transition relation ($\longmapsto$) checks whether *any* stack action can enable the transition:

$$c \underset{M}{\longmapsto} c' \text{ iff } c \overset{g}{\underset{M}{\longmapsto}} c' \text{ for some stack action } g.$$

- For a string of stack actions $g_1 \ldots g_n$:

$$c_0 \overset{g_1 \cdots g_n}{\underset{M}{\longmapsto}} c_n \text{ iff } c_0 \overset{g_1}{\underset{M}{\longmapsto}} c_1 \overset{g_2}{\underset{M}{\longmapsto}} \cdots \overset{g_{n-1}}{\underset{M}{\longmapsto}} c_{n-1} \overset{g_n}{\underset{M}{\longmapsto}} c_n,$$

  for some configurations $c_0, \ldots, c_n$.

- For the transitive closure:

$$c \overset{*}{\underset{M}{\longmapsto}} c' \text{ iff } c \overset{\vec{g}}{\underset{M}{\longmapsto}} c' \text{ for some action string } \vec{g}\,.$$

In Haskell, I need to two functional encodings of δ:

```
type Delta control frame =
 (TopDelta control frame, NopDelta control frame)
type TopDelta control frame =
 control -> frame -> [(control,StackAct frame)]
type NopDelta control frame =
 control -> [(control,StackAct frame)]
```

If we only want to know push and no-change transitions, we can find these with a `NopDelta` function without providing the frame that is currently on top of the stack. If we want pop transitions as well, we can find these with a `TopDelta` function, but of course, it must have access to the top of the stack. In practice, a `TopDelta` function would suffice, but there are situations where only push and no-change transitions are needed, and having access to `NopDelta` avoids extra computation.

Some texts define the transition relation δ so that $\delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$. In these texts, $(q, \gamma, q', \vec{\gamma}) \in \delta$ means, "if in control state q while the character γ is on top, pop the stack, transition to control state q' and push $\vec{\gamma}$." Clearly, we can convert between these two representations by introducing extra control states to my representation when it needs to push multiple characters.

A.4 Rooted pushdown systems

A *rooted pushdown system* is a quadruple (Q, Γ, δ, q_0) in which (Q, Γ, δ) is a pushdown system and $q_0 \in Q$ is an initial (root) state. $\mathbb{RPDS}$ is the class of all rooted pushdown systems.

For a rooted pushdown system $M = (Q, \Gamma, \delta, q_0)$, I define the *reachable-from-root transition relation*:

$$c \overset{g}{\underset{M}{\longmapsto\!\!\!\rightarrow}} c' \text{ iff } (q_0, \langle\rangle) \overset{*}{\underset{M}{\longmapsto}} c \text{ and } c \overset{g}{\underset{M}{\longmapsto}} c'.$$

In other words, the root-reachable transition relation also makes sure that the root control state can actually reach the transition.

I overload the root-reachable transition relation to operate on control states:

$$q \overset{g}{\underset{M}{\longmapsto\!\!\!\rightarrow}} q' \text{ iff } (q, \vec{\gamma}) \overset{g}{\underset{M}{\longmapsto\!\!\!\rightarrow}} (q', \vec{\gamma}\,') \text{ for some stacks } \vec{\gamma}, \vec{\gamma}\,'.$$

For both root-reachable relations, if we elide the stack-action label, then, as in the unrooted case, the transition holds if *there exists* some stack action that enables the transition:

$$q \underset{M}{\longmapsto\!\!\!\rightarrow} q' \text{ iff } q \overset{g}{\underset{M}{\longmapsto\!\!\!\rightarrow}} q' \text{ for some action } g.$$

A.5 Computing reachability in pushdown systems

A pushdown flow analysis can be construed as computing the *root-reachable* subset of control states in a rooted pushdown system, $M = (Q, \Gamma, \delta, q_0)$:

$$\left\{ q : q_0 \underset{M}{\longmapsto\!\!\!\rightarrow} q \right\}.$$

Reps et al. and many others provide a straightforward "summarization" algorithm to compute this set [30, 79, 112, 113]. Chapter 3 develops a complete alternative to summarization.

A.6 Pushdown automata

A *pushdown automaton* is an input-accepting generalization of a rooted pushdown system, a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ in which:

1. Σ is an input alphabet;
2. $\delta \subseteq Q \times \Gamma_{\pm} \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation;
3. $F \subseteq Q$ is a set of accepting states; and
4. $\vec{\gamma} \in \Gamma^*$ is the initial stack.

I use $\mathbb{PDA}$ to denote the class of all pushdown automata.

Pushdown automata recognize languages over their input alphabet. To do so, their transition relation may optionally consume an input character upon transition. Formally, a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ recognizes the language $\mathcal{L}(M) \subseteq \Sigma^*$:

$$\begin{aligned}
\epsilon \in \mathcal{L}(M) &\text{ if } q_0 \in F \\
aw \in \mathcal{L}(M) &\text{ if } \delta(q_0, \gamma_+, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \gamma : \vec{\gamma}) \\
aw \in \mathcal{L}(M) &\text{ if } \delta(q_0, \epsilon, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma}) \\
aw \in \mathcal{L}(M) &\text{ if } \delta(q_0, \gamma_-, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma}') \\
&\text{ where } \vec{\gamma} = \langle \gamma, \gamma_2, \ldots, \gamma_n \rangle \text{ and } \vec{\gamma}' = \langle \gamma_2, \ldots, \gamma_n \rangle,
\end{aligned}$$

where a is either the empty string ϵ or a single character.

A.7 Nondeterministic finite automata

In this disseration, I need a finite description of all possible stacks at a given control state within a rooted pushdown system. I exploit the fact that the set of stacks at a given control point is a regular language. Specifically, I extract a nondeterministic finite automaton accepting that language from the structure of a rooted pushdown system. A *nondeterministic finite automaton* (NFA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$:

- Q is a finite set of control states;
- Σ is an input alphabet;
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation.
- q_0 is a distinguished start state.
- $F \subseteq Q$ is a set of accepting states.

I denote the class of all NFAs as $\mathbb{NFA}$.

In Haskell, I represent an NFA as a set of labeled forward edges, the inverse of those edges (for convenience), a start state and an end state:

```
type NFA state char =
 (NFAEdges state char, NFAEdges state char, state, state)
type NFAEdges state char = state :-> ℙ(Maybe char, state)
```

A.8 Transliterating formalism into Haskell

Where it is critical to understanding the details of the analysis, I have transliterated the formalism into Haskell. I make use of a two extensions in GHC:

```
-XTypeOperators -XTypeSynonymInstances
```

All code is in the context of the following header:

```
import Prelude hiding ((!!))

import Data.Map as Map hiding (map,foldr)
import Data.Set as Set hiding (map,foldr)
import Data.List as List hiding ((!!))

type ℙ s = Set.Set s
type k :-> v = Map k v

(==>) :: a -> b -> (a,b)
(==>) x y = (x,y)

(//) :: Ord a => (a :-> b) -> [(a,b)] -> (a :-> b)
(//) f [(x,y)] = Map.insert x y f

set x = Set.singleton x
```

REFERENCES

[1] *High Performance Fortran language specification version 1.0*, tech. rep., High Performance Fortran Forum (HPFF), 1993.

[2] *High Performance Fortran language specification version 2.0*, tech. rep., High Performance Fortran Forum (HPFF), 1997.

[3] *Chapel Specification 0.4*, tech. rep., Cray Inc, Feb. 2005.

[4] *UPC language specifications, version 1.2*, tech. rep., Lawrence Berkeley National Laboratory, 2005.

[5] *Nvidia CUDA compute unified device architecture: Programming guide version 1.0*, tech. rep., 2007.

[6] *Base specifications, issue 7*, tech. rep., The Open Group, 2009.

[7] *The OpenCL specification, version 1.2, revision 19*, tech. rep., Khronos OpenCL Working Group, 2011.

[8] *OpenMP specifications version 3.1*, tech. rep., OpenMP Architecture Review Board, 2011.

[9] *Chapel Specification 0.91*, tech. rep., Cray Inc, 2012.

[10] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al., *The Fortress language specification*, Sun Microsystems, 2005.

[11] P. America and F. van der Linden, *A parallel object-oriented language with inheritance and subtyping*, in Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications, New York, NY, USA, 1990, ACM, pp. 161–168.

[12] C. Amza, A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, *Treadmarks: Shared memory computing on networks of workstations*, IEEE Computer, 29 (1996), pp. 18–28.

[13] J. N. C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, *Dome: Parallel programming in a distributed computing environment*, in Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International, IEEE, 1996, pp. 218–224.

[14] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. C. Eigler, and G. G. Gao, *ABC++: Concurrency by Inheritance in C++*, IBM Systems Journal, 34 (1995), pp. 120–137.

[15] R. Armstrong, *POET (Parallel Object-oriented Environment and Toolkit) and Frameworks for Scientific Distributed Computing*, in Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture - Volume 1, HICSS '97, Washington, DC, USA, 1997, IEEE Computer Society.

[16] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, *P3L: A structured high-level parallel language, and its structured support*, Concurrency: Practice and experience, 7 (1995), pp. 225–255.

[17] R. Bagrodia, M. Chandy, and M. Dhagat, *UC: A set-based language for data-parallel programming*, Journal of Parallel Distributed Computing, 28 (1995), pp. 186–201.

[18] H. E. Bal and A. S. Tanenbaum, *Distributed programming with shared data*, Computer Languages, 16 (1991), pp. 129–146.

[19] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, *Munin: Distributed shared memory based on type-specific memory coherence*, SIGPLAN Not., 25 (1990), pp. 168–176.

[20] B. N. Bershad, E. D. Lazowska, and H. M. Levy, *PRESTO: A system for object-oriented parallel programming*, Softw. Pract. Exper., 18 (1988), pp. 713–732.

[21] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, *The Midway Distributed Shared Memory System*, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[22] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson, *Uintah: A scalable framework for hazard analysis*, in Proceedings of the 2010 TeraGrid Conference, TG '10, New York, NY, USA, 2010, ACM, pp. 3:1–3:8.

[23] M. Berzins, Q. Meng, J. Schmidt, and J. C. Sutherland, *DAG-based software frameworks for PDEs*, in Euro-Par 2011: Parallel Processing Workshops, Springer, 2012, pp. 324–333.

[24] K. P. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood, *The ISIS system manual, version 2.0*, tech. rep., Upson Hall, Ithaca, NY, USA, 1990.

[25] B. Bland, *Titan - Early experience with the Titan system at Oak Ridge National Laboratory*, SC Companion: High Performance Computing, Networking Storage and Analysis, (2012), pp. 2189–2211.

[26] G. E. Blelloch and J. Greiner, *A provable time and space efficient implementation of NESL*, in Proceedings of the first ACM SIGPLAN International Conference on Functional Programming, ICFP '96, New York, NY, USA, 1996, ACM, pp. 213–225.

[27] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, *A type and effect system for deterministic parallel java*, in Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, New York, NY, USA, 2009, ACM, pp. 97–116.

[28] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, *Distributed pC++: Basic ideas for an object parallel language*, Scientific Programming, 2 (1993), pp. 7–22.

[29] D. Boles, *Parallel object-oriented programming with QPC++*, Structured Programming, 14 (1993), pp. 14–158.

[30] A. Bouajjani, J. Esparza, and O. Maler, *Reachability analysis of pushdown automata: Application to model-checking*, in Proceedings of the 8th International Conference on Concurrency Theory, CONCUR '97, Springer-Verlag, 1997, pp. 135–150.

[31] M. E. Brachet, D. I. Meiron, S. A. Orszag, B. Nickel, R. H. Morf, and U. Frisch, *Small-scale structure of the Taylor-Green vortex*, J. Fluid Mech, 130 (1983), pp. 411–452.

[32] T. Bräunl, *Parallel programming: An introduction*, Prentice Hall, 1993.

[33] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, *A heterogeneous parallel framework for domain-specific languages*, in 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2011, pp. 89–100.

[34] H. Carr, *Distribued C++*, PhD thesis, University of Utah, 1994.

[35] J. Carreira, L. Silva, J. G. Silva, and J. G. Silva, *On the design of Eilean: A Linda-like library for MPI*, tech. rep., Proceedings of the Second Scalable Parallel Libraries Conference, 1994.

[36] R. Chandra, A. Gupta, and J. L. Hennessy., *COOL: A language for parallel programming*, tech. rep., Stanford, CA, USA, 1989.

[37] K. Chandy and J. Misra, *Parallel program design: A foundation*, Computer Science Series, Addison-Wesley Pub. Co., 1988.

[38] K. M. Chandy and C. Kesselman, *CC++: A declarative concurrent object oriented programming notation*, tech. rep., Pasadena, CA, USA, 1993.

[39] K. Clark and S. Gregory, *PARLOG: Parallel programming in logic*, ACM Trans. Program. Lang. Syst., 8 (1986), pp. 1–49.

[40] A. Colvin, *ViC*: Running Out-Of-Core Instead Of Running Out Of Core*, PhD thesis, Dartmouth College, 1999.

[41] P. Cousot, *The calculational design of a generic abstract interpreter*, in Calculational System Design, M. Broy and R. Steinbrüggen, eds., NATO ASI Series F. IOS Press, Amsterdam, 1999.

[42] P. Cousot and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, ACM Press, 1977, pp. 238–252.

[43] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al., *Liszt: A domain specific language for building portable mesh-based PDE solvers*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 9.

[44] J. Dorband and M. Aburdene, *Architecture-adaptive computing environment: A tool for teaching parallel programming*, Frontiers in Education, 3 (2002).

[45] C. Earl, I. Sergey, M. Might, and D. Van Horn, *Introspective pushdown analysis of higher-order programs*, in Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012), ICFP '12, ACM, 2012, pp. 177–188.

[46] G. Edjlali, A. Sussman, and J. Saltz, *Interoperability of data parallel runtime libraries with Meta-Chaos*, in Proceedings of the Eleventh International Parallel Processing Symposium, Society Press, 1997.

[47] M. Felleisen and D. P. Friedman, *A calculus for assignments in higher-order languages*, in POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1987, pp. 314+.

[48] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, *The essence of compiling with continuations*, in PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, ACM, 1993, pp. 237–247.

[49] I. Foster, C. Kesselman, and S. Tuecke, *The Nexus task-parallel runtime system*, in Proc. 1st Intl Workshop on Parallel Processing, 1994, pp. 457–462.

[50] I. Foster, S. Taylor, et al., *Strand: A practical parallel programming language*, tech. rep., Argonne National Lab., IL (USA), 1989.

[51] I. T. Foster and K. M. Chandy, *Fortran M: A language for modular parallel programming*, Journal of Parallel and Distributed Computing, 26 (1995), pp. 24–35.

[52] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, *Fortran D language specification*, tech. rep., Center for Research on Parallel Computation, Rice University, 1990.

[53] M. Frigo, C. E. Leiserson, and K. H. Randall, *The implementation of the Cilk-5 multithreaded language*, in Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI '98, New York, NY, USA, 1998, ACM, pp. 212–223.

[54] D. Gelernter and N. Carriero, *Coordination languages and their significance*, Commun. ACM, 35 (1992), pp. 97–107.

[55] R. Govindarajan, L. Guo, S. Yu, and P. Wang, *ParC project: Practical constructs for parallel programming languages*, in Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International, 1991, pp. 183–189.

[56] A. Granicz, D. M. Zimmerman, and J. Hickey, *Rewriting Unity*, in Proceedings of the 14th International Conference on Rewriting Techniques and Applications, RTA'03, Berlin, Heidelberg, 2003, Springer-Verlag, pp. 138–147.

[57] A. Grimshaw, *Easy-to-use object-oriented parallel processing with Mentat*, IEEE Computer, 26 (1993), pp. 39–51.

[58] J. F. Groote and A. Ponse, *The syntax and semantics of μCRL*, Springer, 1995.

[59] J. E. Guilkey, T. B. Harman, and B. Banerjee, *An Eulerian-Lagrangian approach for simulating explosions of energetic devices*, Comput. Struct., 85 (2007), pp. 660–674.

[60] T. Hagerup, A. Schmitt, and H. Seidl, *Fork: A high-level language for prams*, Future Generation Computer Systems, 8 (1992), pp. 379–393.

[61] R. H. Halstead Jr, *Multilisp: A language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems (TOPLAS), 7 (1985), pp. 501–538.

[62] P. B. Hansen, *Joycea programming language for distributed systems*, Software: Practice and Experience, 17 (1987), pp. 29–50.

[63] ——, *The programming language SuperPascal*, Software: Practice and Experience, 24 (1994), pp. 467–483.

[64] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM, 21 (1978), pp. 666–677.

[65] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota, *Design and implementation of metalevel architecture in C++-MPC++ approach*, in Proceedings of Reflection, vol. 96, 1996, pp. 153–166.

[66] A. Itzkovitz, A. Schuster, and L. Shalev, *Supporting multiple programming paradigms for distributed clusters on top of a single virtual parallel machine–the millipede concept*, in Proceedings, Second International Workshop on High-Level Programming Models and Supportive Environments, 1997.

[67] R. Jagannathan, C. Dodd, and I. Agi, *GLU: A high-level system for granular data-parallel programming*, Concurrency - Practice and Experience, 9 (1997), pp. 63–83.

[68] C.-H. Jo, C.-H. Lee, and J. G. Son, *A realization of a concurrent object-oriented programming*, in Proceedings of the 1998 ACM symposium on Applied Computing, ACM, 1998, pp. 558–563.

[69] E. Johnson and D. Gannon, *HPC++: Experiments with the parallel standard template library*, in Proceedings of the 11th International Conference on Supercomputing, ACM, 1997, pp. 124–131.

[70] D. G. Kafura, M. Mukherji, and G. R. Lavender, *ACT++ 2.0: A class library for concurrent programming in C++ using actors*, tech. rep., 1992.

[71] L. Kalé, B. Ramkumar, A. Sinha, and A. Gürsoy, *The Charm parallel programming language and system: Part I—description of language features*, IEEE TPDS, 12 (1994).

[72] L. V. Kalé and M. A. Bhandarkar, *Structured Dagger: A coordination language for message-driven programming*, in Euro-Par'96 Parallel Processing, Springer, 1996, pp. 646–653.

[73] L. V. Kale and S. Krishnan, *Charm++: A portable concurrent object oriented system based on C++*, in Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications, 1993, pp. 91–108.

[74] K. K. Kee and S. Hariri, *Efficient communication algorithms for pipeline multicomputers*, in Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Supercomputing '94, Los Alamitos, CA, USA, 1994, IEEE Computer Society Press, pp. 468–477.

[75] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C Programming Language, Second Edition*, Prentice-Hall Englewood Cliffs, 1988.

[76] T. Kielmann, *Object-oriented distributed programming with Objective Linda*, in Proceeding of the First International Workshop on High Speed Networks and Open Distributed Platforms, 1995.

[77] N. Kobayashi, *Types and higher-order recursion schemes for verification of higher-order programs*, in Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 44 of POPL '09, ACM, Jan. 2009, pp. 416–428.

[78] J. Kodumal and A. Aiken, *The set constraint/CFL reachability connection in practice*, SIGPLAN Not., 39 (2004), pp. 207–218.

[79] ——, *The set constraint/CFL reachability connection in practice*, in PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM, June 2004, pp. 207–218.

[80] S. R. Kohn and S. B. Baden, *A robust parallel programming model for dynamic non-uniform scientific computations*, in Proceedings of the Scalable High-Performance Computing Conference, IEEE, 1994, pp. 509–517.

[81] S. Kusakabe and M. Amamiya, *A dataflow-based massively parallel programming language V and its implementation on a stock parallel machine*, in Theory and Practice of Parallel Programming, Springer, 1995, pp. 457–471.

[82] Z. Lahjomri and T. Priol, *Koan: A shared virtual memory for the iPSC/2 hypercube*, in Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing, CONPAR '92/ VAPP V, Springer, 1992, pp. 441–452.

[83] J. R. Larus, B. Richards, and G. Viswanathan, *C**: A large-grain, object-oriented, data-parallel programming language*, in Languages and Compilers for Parallel Computing (5th International Workshop), vol. 757, 1992, pp. 326–341.

[84] A. L. Lastovetsky, *mpC: A multi-paradigm programming language for massively parallel computers*, ACM SIGPLAN Notices, 31 (1996), pp. 13–20.

[85] M. Lemke and D. J. Quinlan, *P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications*, in Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing, CONPAR '92/ VAPP V, London, UK, UK, 1992, Springer-Verlag, pp. 121–126.

[86] S. S. Lumetta, A. Krishnamurthy, and D. E. Culler, *Towards modeling the performance of a fast connected components algorithm on parallel machines*, in Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), 1995.

[87] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. Warren, A. Calderwood, P. Szeredi, S. Haridi, et al., *The Aurora or-parallel Prolog system*, New Generation Computing, 7 (1990), pp. 243–271.

[88] E. Lusk and K. Yelick, *Languages for high-productivity computing: the darpa hpcs language project*, Parallel Processing Letters, 17 (2007), pp. 89–102.

[89] D. May, *Occam*, ACM Sigplan Notices, 18 (1983), pp. 69–79.

[90] D. Melski and T. W. Reps, *Interconvertibility of a class of set constraints and context-free-language reachability*, Theoretical Computer Science, 248 (2000), pp. 29–98.

[91] J. Midtgaard, *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*, PhD thesis, University of Aarhus, 2007.

[92] J. Midtgaard and T. P. Jensen, *Control-flow analysis of function calls and returns by abstract interpretation*, in ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ACM, 2009, pp. 287–298.

[93] M. Might, *Environment Analysis of Higher-Order Languages*, PhD thesis, Georgia Institute of Technology, June 2007.

[94] M. Might, B. Chambers, and O. Shivers, *Model checking via Gamma-CFA*, in Proceedings of Verification, Model Checking, and Abstract Interpretation, 2007, pp. 59–73.

[95] M. Might, D. Darais, and D. Spiewak, *Parsing with derivatives: A functional pearl*, in ICFP '11: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ACM, 2011, pp. 189–195.

[96] M. Might and T. Prabhu, *Interprocedural dependence analysis of higher-order programs via stack reachability*, in Proceedings of the 2009 Workshop on Scheme and Functional Programming, 2009.

[97] M. Might and O. Shivers, *Environment analysis via Delta-CFA*, in Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), ACM, 2006, pp. 127–140.

[98] ——, *Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting*, in Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), ACM, 2006, pp. 13–25.

[99] F. Mueller, *Distributed shared-memory threads: DSM-threads*, in Workshop on Run-Time Systems for Parallel Programming, 1997, pp. 31–40.

[100] P. Newton and J. C. Browne, *The CODE 2.0 graphical parallel programming language*, in Proceedings of the 6th International Conference on Supercomputing, ACM, 1992, pp. 167–177.

[101] G. Niccolai, *The Falcon programming language in a nutshell*, Linux Journal, 2008 (2008), p. 4.

[102] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, *Advances, applications and performance of the global arrays shared memory programming toolkit*, International Journal of High Performance Computing Applications, 20 (2006), pp. 203–231.

[103] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland, *Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software*, ACM Transactions on Mathematical Software (TOMS), 39 (2012).

[104] R. W. Numrich and J. Reid, *Co-Array Fortran for parallel programming*, in Proceedings of ACM Sigplan Fortran Forum, vol. 17, 1998, pp. 1–31.

[105] C. H. L. Ong, *On model-checking trees generated by higher-order recursion schemes*, in 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06), IEEE, 2006, pp. 81–90.

[106] S. Owens, J. Reppy, and A. Turon, *Regular-expression derivatives re-examined*, Journal of Functional Programming, 19 (2009), pp. 173–190.

[107] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin, *Polaris: A new-generation parallelizing compiler for MPPs*, in CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign, 1993.

[108] S. G. Parker, *A component-based architecture for parallel multi-physics PDE simulation*, in Computational ScienceICCS 2002, Springer, 2002, pp. 719–734.

[109] M. J. Phillip, *Unification of Synchronous and Asynchronous Models for Parallel Programming Languages*, Master's Thesis, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, (1989).

[110] J. Rehof and M. Fähndrich, *Type-based flow analysis: From polymorphic subtyping to CFL-reachability*, in POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2001, pp. 54–66.

[111] J. Reid, *The new features of Fortran 2008*, in Proceedings of ACM SIGPLAN Fortran Forum, vol. 27, ACM, 2008, pp. 8–21.

[112] T. Reps, *Program analysis via graph reachability*, Information and Software Technology, 40 (1998), pp. 701–726.

[113] T. Reps, S. Schwoon, S. Jha, and D. Melski, *Weighted pushdown systems and their application to interprocedural dataflow analysis*, Science of Computer Programming, 58 (2005), pp. 206–263.

[114] J. Reynders, *The POOMA framework: A templated class library for parallel scientific computing*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[115] M. C. Rinard and M. S. Lam, *The design, implementation, and evaluation of Jade*, ACM Transactions on Programming Languages and Systems (TOPLAS), 20 (1998), pp. 483–545.

[116] B. Robben, W. Joosen, F. Matthijs, B. Vanhaute, and P. Verbaeten, *A metaobject protocol for correlate*, Lecture Notes in Computer Science, 1543 (1998), pp. 367–368.

[117] J. Rose and G. L. Steele Jr, *C*: An extended C language for data parallel programming*, tech. rep., Technical Report PL87-5, Thinking Machines Corporation, 1987.

[118] A. Rowstron and A. Wood, *Implementing mathematical morphology in ISETL-Linda*, in Image Processing and its Applications, 1995., Fifth International Conference on, IET, 1995, pp. 847–851.

[119] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, *X10 language specification*, tech. rep., IBM, 2012.

[120] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, *Large scale parallel solution of incompressible flow problems using uintah and hypre*, IEEE International Symposium on Cluster Computing and the Grid, (2013), pp. 458–465.

[121] O. G. Shivers, *Control-Flow Analysis of Higher-Order Languages*, PhD thesis, Carnegie Mellon University, 1991.

[122] J. G. Silva, J. Carreira, and F. Moreira, *ParLin: From a centralized tuple space to adaptive hashing*, Transputer Applications and Systems 94, (1993), pp. 91–104.

[123] M. Sipser, *Introduction to the theory of computation*, Cengage Learning, 2 ed., 2005.

[124] G. Steele, *Fortress wrapping up.* https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up, July 2012.

[125] B. Stroustrup, *The C++ programming language*, Addison-Wesley Publishing Company, 1997.

[126] J. Subhlok, D. R. O'Hallaron, and T. Gross, *Task parallel programming in Fx*, tech. rep., DTIC Document, 1994.

[127] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, *OptiML: An implicitly parallel domain-specific language for machine learning*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 609–616.

[128] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, et al., *Composition and reuse with compiled domain-specific languages*, in Proceedings of ECOOP, 2013.

[129] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, *The Pochoir stencil compiler*, in Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2011, pp. 117–128.

[130] G. Taylor and A. Green, *Mechanism of the production of small eddies from large ones*, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 158 (1937), pp. 499–521.

[131] W. F. Tichy and C. G. Herter, *Modula-2*: An extension of Modula-2 for highly parallel*, tech. rep., University of Karlsruhe, 1990.

[132] A. Tripathi and M. Akşit, *Communication, scheduling, and resource management in SINA*, Journal of Object-Oriented Programming, 1 (1988), pp. 24–31.

[133] E. Unruh, *Prime number computation*, tech. rep., ANSI X3J16-94-0075/ISO WG21-462, 1994.

[134] D. Van Horn and H. G. Mairson, *Deciding kCFA is complete for EXPTIME*, in ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ACM, 2008, pp. 275–282.

[135] D. Vardoulakis and O. Shivers, *CFA2: A context-free approach to control-flow analysis*, in European Symposium on Programming (ESOP), vol. 6012 of LNCS, Springer, 2010, pp. 570–589.

[136] T. Veldhuizen, *Template metaprograms*, C++ Report, 7 (1995), pp. 36–43.

[137] T. L. Veldhuizen, *Arrays in Blitz++*, in Computing in Object-Oriented Parallel Environments, Springer, 1998, pp. 223–230.

[138] D. W. Walker, D. W. Walker, J. J. Dongarra, and J. J. Dongarra, *MPI: A standard message passing interface*, Supercomputer, 12 (1996), pp. 56–68.

[139] N. Wirth, *The programming language Pascal*, Acta informatica, 1 (1971), pp. 35–63.

[140] A. K. Wright and S. Jagannathan, *Polymorphic splitting: An effective polyvariant flow analysis*, ACM Transactions on Programming Languages and Systems, 20 (1998), pp. 166–207.

[141] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al., *Titanium: A high-performance Java dialect*, Concurrency Practice and Experience, 10 (1998), pp. 825–836.