The Design and Implementation of User-Schedulable Languages

Alexander Reinking

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-271
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-271.html

December 18, 2022

The Design and Implementation of User-Schedulable Languages

by

Alex Reinking

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Jonathan Ragan-Kelley, Chair
Professor Katherine Yelick
Professor Sarah Chasins
Professor Per-Olof Persson

Fall 2022

The Design and Implementation of User-Schedulable Languages

Abstract

The Design and Implementation of User-Schedulable Languages

by

Alex Reinking

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Jonathan Ragan-Kelley, Chair

This thesis details an emerging class of programming language designs, called *user-schedulable languages*, that provide a safe and productive performance engineering environment for modern, heterogeneous hardware. The defining trait of user-schedulable languages is the division of program specification into two key parts: the *algorithm*, which defines functionally *what* is to be computed, and the *schedule*, which defines *how* the computation should be carried out. Importantly, algorithms have semantics independent of any schedule, and schedules are semantics-preserving with respect to the algorithm. Thus, programmers are freed from a large class of bugs. Because algorithm languages tend to be functional and domain-specific, the scheduling language can be very expressive. Existing scheduling languages include many program transformations, from accelerator offloading to tricky tiling and interleaving strategies. Multiple schedules can be written for different sets of hardware targets and can be maintained independently from one another.

Here, we formally analyze the design of an existing and widely deployed user-schedulable language, Halide, and find and correct several serious bugs and design flaws through this analysis. We also detail both the design and implementation of a new user-schedulable language, Exo, whose design is informed by the lessons learned analyzing Halide. Unlike Halide, which models scheduling as a parameter to a monolithic lowering process, Exo uses a rewrite-based scheduling system. This system doubles as an instruction selection process for custom accelerator hardware; importantly, these instructions can be specified in user programs, rather than inside the compiler itself. We then discuss a novel, high-performance, reference-counting memory management strategy suitable for recursive programs with highly non-local control flow over a (co-)inductive data domain. Finally, practical considerations for language design are discussed; these are lessons learned from maintaining and deploying these systems in practice.

To my parents, Susan and Ricardo, who tirelessly sought new opportunities to educate me.

Contents

List of Figures

List of Tables

Acknowledgments

I doubt anyone who has known me long will be surprised to read these words: I finished my Ph.D.! And so over twenty years of effort has reached a seemingly inevitable inflection point. What's next? We'll see. But getting here certainly wasn't easy, and it wouldn't have been possible without the many people I've had in my corner offering their mentorship, support, love, and friendship throughout this journey.

I'll start by thanking my advisor, Jonathan Ragan-Kelley, among whose students I am the first to graduate, for his unwavering confidence in me, even when things didn't go according to plan. Through a move to MIT and a global pandemic, Jonathan has never steered me wrong. I hope we can continue to collaborate far into the future.

I've been blessed to have amazing collaborators across several projects and industries. I'd like to thank Gilbert Bernstein, Yuka Ikarashi, Hasan Genc, Daan Leijen, Ningning Xie, Leonardo de Moura, Dougal Maclaurin, Adam Paszke, Alexey Radul, Ankush Desai, and Shaz Qadeer for working with me on such awesome projects. I'm honored that the Halide team, most notably Steven Johnson, Andrew Adams, Shoiab Kamil, and Zalman Stern, have entrusted me with so much of this project. I'd also like to thank Miguel Nuñez for his mentorship while I was at Microsoft.

I'd like to thank many important people from my time at Yale. I have Paul Hudak, Dan Spielman, and especially Ruzica Piskac to thank for training me in the fundamentals of academic research. Ruzica and I published my first paper together; I doubt I would have become a graduate student without her. Stanley Eisenstat taught me how to teach and gave me far too much of his time in office hours and beyond. I learned so much from all my other wonderful professors, but I'd like to specifically thank Asaf Hadari, Nicholas Christakis, and Zhong Shao for their lessons that shaped both my character and approach to research. I am lucky to have kept classmates Dani and Jessica as friends for so many years.

I would not have kept my sanity through the pandemic, let alone graduate school, without my family in Minnesota. My parents, my grandma, my brother Eric, and my honorary sister Hanna have all given me so much love and support. I can only hope to pay it forward.

I want to thank my many friends at Berkeley for so many fun memories. Bad movie nights and Oakland excursions with Gautam Gunjala, Alain Anton, Alan Dong, Kieran Peleaux, Stan Smith, Chandan Singh, and Phong Nguyen were always a blast. Jeremy Warner and I should have played guitar together much sooner. Traveling to Mexico with Ben Brock was unforgettable. Raucous barbeques hosted by Arya Reais-Parsi and Zoe Cohen were almost worth breaking my leg for. Game nights with Ben and Aviva Mehlow, Rick Brewster, and Cristina Teodoropol somehow always found a touch of philosophy.

Shout out to all my neighbors who made West Berkeley feel like home: Alex, Angelica, Jonathan, David, Greg, Ron, Hugh, Allie, Adam, and Jen.

Last, but not least, I thank Rachel Lawrence for a decade of love and companionship.

Chapter 1

Introduction

For most of computing history, single-core CPU performance grew exponentially. As a result, the general-purpose programming systems of that era prioritized programmer productivity over performance. Given the context in which hardware upgrades could double performance every two years or so, it made more sense to allocate developer time to shipping new features rather than optimization.

However, at the turn of the century, physical limitations at small transistor sizes slowed this growth. In response, hardware engineers began adding more CPU cores to their designs. In 2005, both AMD and Intel released their first consumer-grade multi-core chips. Around the same time, general-purpose GPU programming became more attractive when researchers discovered that pixel shaders could be used to perform linear algebra. With the release of CUDA in 2007, GPU acceleration became the norm for numerical workloads such as image editing, scientific computing, and computer-aided design.

In more recent years, the rise of deep neural networks and other machine learning systems has led to an explosion in demand for processing increasingly large data sets. Model training often occurs in large data centers, and the trained models are deployed on a wide range of devices, from servers to "edge" devices like mobile phones. Because these systems are highly compute- and power-intensive, there has never been a greater demand for efficiency at all scales. In response, hardware engineers have designed purpose-built accelerators for these applications. Modern hardware now includes a wide variety of accelerators, such as Apple's Neural Engine and Qualcomm's Hexagon DSP.

However, the industry-standard programming languages for high-performance computing, such as C and C++, still model execution as running on a single thread on a single CPU core. Even though C gained a parallel memory model in 2011, fewer than half of C programmers today can use the C11 standard due to the slow pace of adoption in the industry. Furthermore, modern compilers are still unable to bridge the gap between simple code and fast code. Even in the case of matrix multiplication on a single CPU core, there are several orders of magnitude difference in both code size and throughput between a naive implementation and a fast one. The fact that over 50 years of compilers research has not yielded a satisfactory solution to this simpler problem suggests that traditional languages are not equipped to handle the new era of highly heterogeneous machines.

In this thesis, we examine an increasingly popular approach to language design called *user scheduling*. These user-schedulable languages strike a balance between abstraction and control in high-performance computing by separating the specification of *what* a program should compute (known as the "algorithm") from a schedule for *how* to compute it. In the process, they make a novel language soundness claim: the result of a program should always be the same, regardless of how it is scheduled.

Typically, the algorithm language describes the desired computation at a high level, without low-level details like memory allocation or complex arithmetic in loop bounds and indexing expressions. The scheduling language includes a series of directives that guide the compilation of this algorithm to a target language, such as C, a lower-level IR, or the algorithm language again. These directives include program transformations that adjust performance, for example, by tiling iteration spaces for better cache locality or mapping program fragments to specific accelerators.

We present the first formalization and metatheory of language soundness for a user-schedulable language, the widely used array processing language Halide. Its soundness guarantee is tricky to provide in the presence of schedules that introduce redundant recomputation and computation on uninitialized data, rather than simply reordering statements. In addition, Halide ensures memory safety through a compile-time *bounds inference* engine that determines safe sizes for every buffer and loop in the generated code, presenting a novel challenge: formalizing and analyzing a language specification that depends on the results of unreliable program synthesis algorithms. Our formalization has revealed flaws and led to improvements in the practical Halide system, and we believe it provides a foundation for the design of new languages and tools that apply programmer-controlled scheduling to other domains.

High-performance kernel libraries are critical to exploiting accelerators and specialized instructions in many applications. Because compilers are difficult to extend to support diverse and rapidly-evolving hardware targets, and automatic optimization is often insufficient to guarantee state-of-the-art performance, these libraries are commonly still coded and optimized by hand, at great expense, in low-level C and assembly. To better support development of high-performance libraries for specialized hardware, we propose a new programming language, Exo, based on the principle of *exocompilation*: externalizing target-specific code generation support and optimization policies to user-level code. Exo allows custom hardware instructions, specialized memories, and accelerator configuration state to be defined in user libraries. It builds on the idea of user scheduling to externalize hardware mapping and optimization decisions. Schedules are defined as composable rewrites within the language, and we develop a set of effect analyses which guarantee program equivalence and memory safety through these transformations. We show that Exo enables rapid development of state-of-the-art matrix-matrix multiply and convolutional neural network kernels, for both an embedded neural accelerator and x86 with AVX-512 extensions, in a few dozen lines of code each.

Finally, we introduce Perceus, an algorithm for precise reference counting with reuse and specialization. Starting from a functional core language with explicit control-flow, Perceus emits precise reference counting instructions such that (cycle-free) programs are *garbage free*, where only live references are retained. This enables further optimizations, like reuse analysis that allows for guaranteed in-place updates at runtime. This in turn enables a novel programming paradigm that we call *functional but in-place* (FBIP). Much like tail-call optimization enables writing loops with regular function calls, reuse analysis enables writing in-place mutating algorithms in a purely functional way. We give a novel formalization of reference counting in a linear resource calculus, and prove that Perceus is sound and garbage free. We show evidence that Perceus, as implemented in Koka, has good performance and is competitive with other state-of-the-art memory collectors.

Chapter 2

Formal Semantics for the Halide Language

This chapter is based on the work in Reinking, Bernstein, and Ragan-Kelley [133], which is under submission, but available on arXiv.

2.1 Introduction

Halide is a domain-specific language used widely in industry to build high-performance image and array processing pipelines for everything from YouTube, to every Android phone, to Adobe Photoshop [71, 128, 130, 131]. As part of a new generation of *user-schedulable* languages and compilers [6, 22, 42, 62, 77, 86, 114, 144, 172], its design separates the specification of what is to be computed, known as the *algorithm*, from the specification of when and where those computations should be carried out and placed in memory, known as the *schedule*, while allowing both to be supplied by the programmer.

The key value of user scheduling is that programmers are relieved from troubleshooting large classes of bugs that arise when optimizing programs for memory locality, parallelism, and vectorization because these transformations are available in the scheduling language, rather than being directly expressed in the algorithm. This enables a schedule-centric workflow where the majority of effort is spent exploring different optimizations, not ensuring correctness after each attempt. This allows performance engineers to optimize programs competitively with the best hand-tuned C, assembly, and CUDA implementations, but with dramatically less code and development time [130].

Halide specifies algorithms in a purely functional dataflow language of infinite arrays that combines lazy and eager semantics. The schedule then guides compilation to generate some particular eager, imperative implementation. Halide schedules include classic loop *re-ordering* transformations, but their most unique constructs (**compute-at**, **store-at**) induce non-local transformations that intentionally exploit redundant recomputation and computation on uninitialized data—transformations well outside that classic model.

So changing the schedule of a Halide program dramatically changes the computation, but when is this safe and sound?

The safety and soundness of languages with user-controlled scheduling has never been formally defined and analyzed, particularly in the presence of non-reordering transformations (see chapter 6). This paper presents the first formal definition and analysis of the core of Halide, and a general approach to the metatheory of similar languages. We focus on proving a new safety and correctness guarantee unique to user-scheduled languages: regardless of what schedule is supplied, a given algorithm should always produce the same result and be memory-safe.

Formalizing the correctness of Halide is difficult for at least two reasons. First, traditional formalisms for reasoning about the correctness of compiler transformations (especially loop transformations) tend to reduce to dependence analysis of imperative code. This strategy is only applicable to re-ordering transformations, and so we must build our proofs and reasoning on a different structural basis (§2.3). Second, Halide's design is built around a *bounds inference* engine that assumes responsibility for synthesizing all loop and memory-buffer bounds. Because the synthesis of finite bounds is undecidable in the general case with data-dependent accesses and non-affine expressions, compliant bounds inference engines must be allowed to fail. This opens the door to compliant but useless engines (which always fail). We propose a solution to this conundrum, by specifying a reference algorithm to define minimum compliant quality (§2.3.3).

Halide's success in industry has, for better or worse, locked it in to its early design decisions and has influenced the design of its peers. At the same time, these systems have not historically been a subject of interest in formal programming languages. An important consequence of our work has been to crisply define Halide's semantics and metatheory and to correct mistakes without dramatically overhauling the language (§2.9). We further expect this effort can help the next wave of user-schedulable languages to create even more elegant and useful systems without making the same compromises.

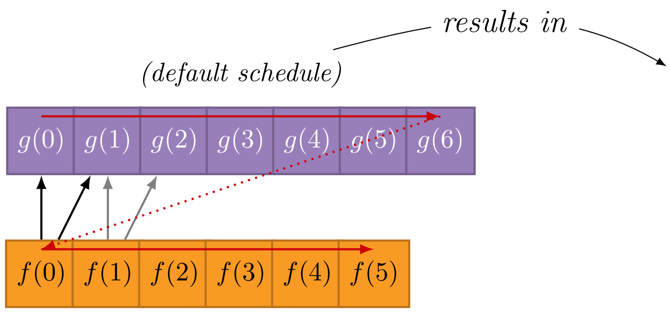
This chapter makes the following contributions:

- We give the first complete semantics and metatheory defining *sound* user-specified scheduling of a high-performance array processing language: that programs are unconditionally memory safe and that their output is not changed by scheduling decisions.

- We give the first precise description of the core of the practical Halide system: the algorithm language, scheduling operators, and bounds inference problem.

- We provide the first definition of Halide's bounds inference feature as a program synthesis problem, which was not previously understood as such.

- We apply our formalism to the practical Halide system, finding & fixing several bugs and making design improvements in the process.



(a) Points of g are shown in purple, and points of f are shown in orange. Dark arrows indicate dependencies; for clarity, only those for $f(0)$ and $f(1)$ are shown. Red arrows trace computation; all points of g are computed first, and only then is f computed.

$$
\begin{array}{l}
\textbf{pipeline}\, f(x_{\text{min}}, x_{\text{len}}) : \\
\quad \textbf{allocate}\, g(\boxed{?^{\text{mem}} g_x}); \\
\quad \textbf{label}\, g : \textbf{label}\, s_0 : \\
\quad\quad \textbf{for}\, x\, \textbf{in}\ \boxed{?^{\text{cpu}} g_x}\ \textbf{do} \\
\quad\quad\quad g[x] \leftarrow \ldots; \\
\quad \textbf{allocate}\, f(\boxed{?^{\text{mem}} f_x}); \\
\quad \textbf{label}\, f : \textbf{label}\, s_0 : \\
\quad\quad \textbf{for}\, x\, \textbf{in}\ \boxed{(x_{\text{min}}, x_{\text{len}})}\ \textbf{do} \\
\quad\quad\quad f[x] \leftarrow g[x] + g[x+1];
\end{array}
$$

(b) Loop nest IR with default schedule. Note the *holes* that will be filled in by bounds inference.

Figure 2.1: Example program with default execution order.

2.2 An Example Halide Program

To introduce key concepts and build intuition for the formalism to follow, we'll consider a minimal example that showcases the challenges present in analyzing the scheduling language. Our example algorithm consists of two *"funcs"*, defined like so:

$$
\begin{array}{l}
\textbf{pipeline}\, f() : \\
\quad \textbf{fun}\, g(x) = \ldots; \\
\quad \textbf{fun}\, f(x) = g(x) + g(x+1);
\end{array}
$$

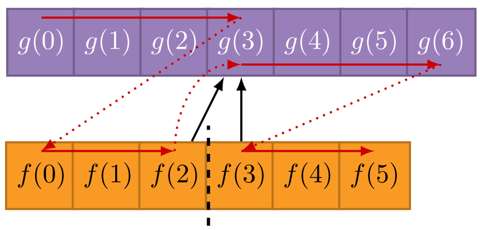
A *func* is the basic unit of computation. Halide funcs are defined on unbounded, n-dimensional, integer lattices, and are *not* bounded, multidimensional arrays. The body of g is deliberately left undefined since it is irrelevant to the upcoming discussion. The formula describing a func must be *total* and *non-recursive*, so that any window of the func has defined values. To run the pipeline, the user supplies as *input* [1] a desired window over which to compute the last func in the pipeline. The program then returns an array (formalized as partial functions) containing the computed values and which might be larger than requested. The computational model is therefore *demand-driven*, unlike most contemporary array languages. We will illustrate the evaluation of f on the window $[0, 6)$, which will in turn necessitate computing g on at least the window $[0, 7)$.

In order to recover an imperative implementation, we *lower* the pipeline into a second, imperative, target language with C-like semantics. While there exist sensible choices for loop

[1]For simplicity of formalization, we omit special treatment of *input* funcs, modeling these instead as procedural funcs with no dependencies. However, the practical system does support input arrays (whose bounds must be checked for consistency when the program starts running).

split$(\langle f, x\rangle, xo, xi, 3)$;
compute-at$(g, \langle f, xo\rangle)$;
store-at$(g, \langle f, xo\rangle)$;

results in



(a) This time the computation of f is split by 3, and g is *interleaved* per-tile. Bounds inference causes $g(3)$ to be *redundantly recomputed*. This allows each tile of f to be trivially parallelized.

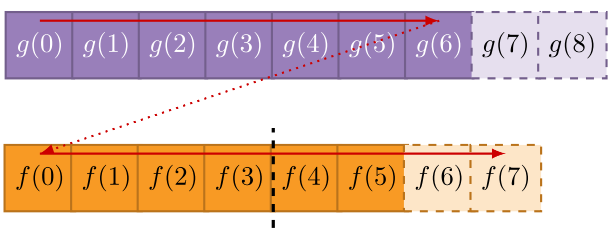
pipeline $f(x_{\min}, x_{\text{len}})$:
  allocate $f($ $?^{\text{mem}} f_x$ $)$;
  label f : **label** s_0 :
    for xo **in** $(x_{\min}, (x_{\text{len}} + 2)/3)$ **do**
      compute g as needed
      for xi **in** $[0, 3)$ **do**
        let $x = x_{\min} + 3xo + xi$ **in**
        if $x < x_{\min} + x_{\text{len}}$ **then**
          $f[x] \leftarrow g[x] + g[x+1]$;

(b) Loop nest IR with recomputing schedule. The loops for g have been abbreviated for space, but are identical to those in fig. 2.1b.

Figure 2.2: Example program after tiling f by 3. This schedule illustrates Halide's ability to introduce *redundant recomputation* to achieve better producer-consumer locality.

split$(\langle f, x\rangle, xo, xi, 4, \varphi_{\text{Round}})$

results in



(a) *Overcompute* occurs since bounds inference takes the (symbolic) requested window into account. Here, only $[0, 6)$ of f is requested, so $f(6)$ and $f(7)$ may contain any value. Thus g may be over-allocated with $g(7)$ and $g(8)$ uninitialized.

pipeline $f(x_{\min}, x_{\text{len}})$:
  compute g as needed
  allocate $f($ $?^{\text{mem}} f_x$ $)$;
  label f : **label** s_0 :
    for xo **in** $(x_{\min}, (x_{\text{len}} + 3)/4)$ **do**
      for xi **in** $[0, 4)$ **do**
        let $x = x_{\min} + 3xo + xi$ **in**
        $f[x] \leftarrow g[x] + g[x+1]$;

(b) Loop nest IR with overcomputing schedule. The loops for g have again been abbreviated. Notice that the **if** statement has disappeared.

Figure 2.3: Example program after vectorizing f and rounding up. This schedule illustrates Halide's ability to introduce *overcompute* on *uninitialized values* to trade-off between compute and storage efficiency.

iteration bounds and buffer sizes, our algorithm never specified these. Therefore, this initial lowered program leaves symbolic *holes* in the code (prefixed with '?').

To fill these holes, Halide performs *bounds inference*, which we formalize as a program synthesis problem. We assume a bounds inference oracle that returns expressions to fill every hole, satisfying derived memory safety and correctness conditions. However, since this oracle is only required to meet safety and correctness conditions, there is no guarantee on the (parameterized) minimality of memory allocations or loop bounds. We will discuss this complication in more detail shortly.

Now we will look at three different ways to schedule our pipeline.

First, the default schedule (fig. 2.1) computes all values required from each func before progressing to the next func, in order of their definition. Note that x_{min} and x_{len} are variables specifying the output window $[x_{\text{min}}, x_{\text{min}} + x_{\text{len}})$. Bounds inference could efficiently fill the hole $?^{\text{mem}}f_x$ with $[x_{\text{min}}, x_{\text{min}} + x_{\text{len}})$ and the holes $?^{\text{cpu}}g_x$ and $?^{\text{mem}}g_x$ with $[x_{\text{min}}, x_{\text{min}} + x_{\text{len}} + 1)$. An inefficient solution could fill $?^{\text{mem}}g_x$ with $[x_{\text{min}}, x_{\text{min}} + 2 \cdot x_{\text{len}})$, but $[x_{\text{min}}, x_{\text{min}} + x_{\text{len}} - 1)$ is not allowed because the last access would write out of bounds.

For our second schedule of f (fig. 2.2), we will tile it so that we can parallelize it, computing f in independent 3-element-wide tiles. This first scheduling directive says to **split**$(\langle f, x\rangle, xo, xi, 3)$ the computation of f along dimension x by a factor of 3 into an outer iteration dimension xo and inner iteration dimension xi. Then, the second directive tells us **compute-at**$(g, \langle f, xo\rangle)$, meaning to re-compute the necessary portion of g at f, within iteration level xo, and then to **store-at**$(g, \langle f, xo\rangle)$, similarly. In terms of imperative code, this is simply a relocation of the loop nest computing g. Bounds inference will now be able to infer much *tighter* bounds on g, since it only needs to be computed on a per-tile basis. For a given value of xo, only 4 values of g need to be computed and stored for use by the xi loop.

Notice that the windows of g required by adjacent tiles of f overlap by one element. In fig. 2.2a we can see that $g(3)$ is required by both tiles of f because both $f(2)$ and $f(3)$ depend on it. This ability to reduce synchronization and improve locality at the expense of redundant recomputation is at the heart of why Halide is able to generate high-performance code for modern micro-processors. It is also one reason why only using re-ordering loop transformations is insufficient.

For our third and final schedule (fig. 2.3), we will tile the computation of f in order to take advantage of fixed-width SIMD instructions (present on most CPUs today). To do so, we call **split** again, but now provide an alternate *tail-strategy*: φ_{Round}. Rather than introducing an if-guard, this strategy will cause f to be *unconditionally* evaluated in 4-wide tiles. If the requested window is not a multiple of 4, it will be rounded up and extra points will be computed.

Perhaps counter-intuitively, this *over-compute* strategy requires *fewer* instructions in a vectorized implementation, since the entire loop tail can be computed with a single instruction, rather than a variable number of scalar operations (e.g. in a loop epilogue). However, whereas the window of allocated, computed, and valid values all coincided before, those 3 windows now all uncouple. For the requested window $[0, 6)$, f is allocated and computed on the window $[0, 8)$, whereas g is allocated on the window $[0, 9)$ and computed on the window $[0, 7)$. Since neither $g(7)$ nor $g(8)$ are initialized, the values in $f(6)$ and $f(7)$ are themselves uninitialized.

The IR programs in figs. 2.1b, 2.2b and 2.3b show an important benefit of user-specified scheduling: in a traditional high performance language like C, a programmer would need to write loops and derive compute bounds by hand. By instead factoring these rewrites into a small scheduling language, Halide programmers can efficiently explore the space of

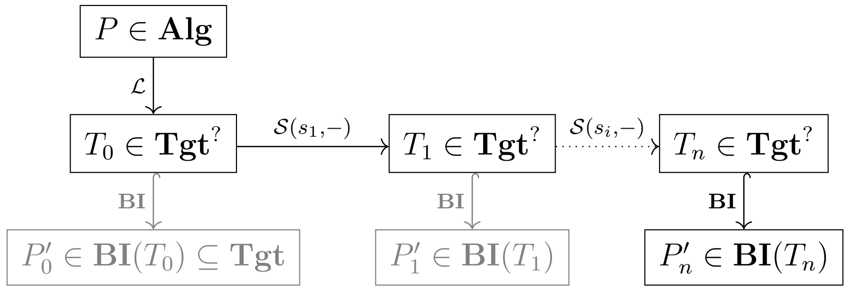


Figure 2.4: Compiling a program P with schedule $S = s_1; \ldots; s_n$ entails first lowering P to an IR program T_0 and then applying each directive s_i in turn. At the end, the bounds oracle **BI** fills the holes in T_n to produce a final program P'_n. Shown in gray are intermediate bounds oracle queries that are useful in the formal analysis, but are not part of compilation.

safe, equivalent programs. In this paper, we explain how—formally—this promise to Halide programmers is justified.

2.3 Overview & Proof Structure

Given an initial program P_0, and a schedule $S = s_1; \ldots; s_n$, let P_i be the result of applying the scheduling primitive s_i to P_{i-1}. Intuitively, if we can show that our soundness property is *invariant* under each possible primitive, then it must hold.

In loop-nest optimization this invariant was traditionally specified via dependence graphs: first over lexical statements, and then over whole iteration spaces of *statement instances*. For instance, Kennedy and Allen [85] formulate this invariant as the *Fundamental Theorem of Dependence*, which states “any reordering transformation that preserves every dependence in a program preserves the meaning of that program.” Unfortunately, this approach is strictly limited to verifying the soundness of *reordering transformations*, i.e. those transformations that permute the order in which statement instances occur, but never change those statements, duplicate them, or introduce new ones.

We resolve this problem in our proof structure by including the original *provenance* of the programs P_i in our soundness invariant. In Halide, this original reference program is the *algorithm*, which is expressed in a functional, rather than imperative, language. Since the functional algorithm does not specify order of execution, nor where and how values are stored in memory buffers, this soundness principle accommodates a greater range of transformations.

Finally, loop transformations on array code inevitably require complicated reasoning about various sets of bounds (e.g., for memory allocation) as transformations are performed. In Halide, these complexities are managed by deferring bounds analyses until *after* scheduling is performed, and by offloading those decisions to a *bounds inference* engine, which we treat here as an oracle. More generally, we expect that advances in program synthesis will only make the transformation of incomplete programs more common; this general approach should work in a variety of new language designs.

In order to handle the transformation of programs with holes, our soundness invariant must be stated on sets of programs (completions) rather than individual programs. Thus, rather than state that each transformed program is consistent with the algorithm reference, we require that all completions are consistent—where defined.

2.3.1 Basic Definitions

Halide programs are specified via an algorithm program denoted $P \in \mathbf{Alg}$, written in a functional language with big-step semantics, defined in §2.4. We immediately *lower* this algorithm into an imperative target language program *with holes* denoted $T \in \mathbf{Tgt}^?$. This language is defined in §2.5. Lowering is specified via a function $T = \mathcal{L}(P)$, defined in §2.6.

The lowered program is incomplete because it is missing various bounds. Halide's bounds inference completes a program with holes $T \in \mathbf{Tgt}^?$ into a program without holes $P' \in \mathbf{Tgt}$. We model the bounds-inference procedure as a non-deterministic oracle $\mathbf{BI}$, which defines a set of completions $\mathbf{BI}(T)$ via a syntactically derived synthesis problem, and returns one as specified in §2.7. Thus, $P' \in \mathbf{BI}(T) \subseteq \mathbf{Tgt}$.

The schedule for a Halide program is specified as a sequence of primitive scheduling directives $S = s_1; \ldots; s_n$, defined in §2.8. Scheduling proceeds by sequentially transforming a target program with holes $T_0 = \mathcal{L}(P)$ by each subsequent scheduling directive such that $T_{i+1} = \mathcal{S}(s_i, T_i)$. Consequently, a set of completions $\mathbf{BI}(T_i)$ is defined at each point in the scheduling process. These intermediate completions are not used when simply compiling a program, but are essential to analyzing the behavior of a scheduling directive by relating the sets before and after the transformation. The fully scheduled program is given as $P'_n \in \mathbf{BI}(\mathcal{S}(S, \mathcal{L}(P)))$. This structure is depicted in fig. 2.4.

2.3.2 Equivalence and Soundness of Programs

Halide makes two fundamental promises to programmers: memory safety and equivalence under scheduling transformations. Here is how we formulate those promises.

Programs are defined as functions of *input parameters* and an *output window*. Thus, the same program can be run multiple times to compute different windows into a conceptually unbounded output array.

Definition 2.1 (Input and Output)**.** Let $P \in \mathbf{Alg}$ or $P \in \mathbf{Tgt}$ be a program with m parameters and an n dimensional output func. An *input* z to P is an assignment to those m parameters and an assignment of n constant intervals defining an output window $R(z)$. The *output* of running P on z is a partial function $f = P(z)$ where $f(x)$ is defined on at least all $x \in R(z)$.

For a variety of reasons, a program may produce *more* than the requested output window $R(z)$. A program may even allocate padding space and fill it with garbage values in order to align storage and/or computation. For these reasons, we only define equivalence up to agreement on the specified output window:

Definition 2.2 (Output equivalence)**.** Let each of P and P' be either an algorithm or target language program, with common input z. We say they have equivalent outputs $P \simeq_z P'$, if for every point $x \in R(z)$, $P(z)(x) = P'(z)(x)$.

This definition of equivalence is sufficient to compare two complete programs. However, because our soundness invariant must be stated on incomplete programs $T \in \mathbf{Tgt}^?$, we will define the *confluence* of an algorithm with all completions of T. We will also have to account for certain exceptional cases in which the output may actually not be equivalent. Namely, if the original algorithm contains errors, then all bets are off, and if the completion of the program fails to satisfy bounds-constraints, then equivalence cannot be guaranteed. This latter case should be concerning; we will address it shortly.

Definition 2.3 (Algorithm confluence)**.** Let $P \in \mathbf{Alg}$ be a Halide algorithm and let $T \in \mathbf{Tgt}^?$ be a target language program with holes. We say that T is *confluent* with P if for all $P' \in \mathbf{BI}(T)$ and all inputs z, either $P(z)$ contains an *error value* in $R(z)$, $P'(z)$ *fails an assertion check*, or $P \simeq_z P'$. Error values and assertion failures are detailed in §2.4.1 and §2.5.2, respectively.

We are now able to state the two fundamental theorems about Halide. In stating these theorems, we assume that algorithm language programs are *valid* (§2.4.2), as are schedules (§2.8).

Theorem 2.1 (Memory safety)**.** *Let $P \in \mathbf{Alg}$ be a valid program, z an input, and S a valid schedule. Then, for all target language programs $P' \in \mathbf{BI}(\mathcal{S}(S, \mathcal{L}(P)))$, the computation $P'(z)$ will not access any out of bounds memory (§2.5.2).*

Memory safety will be guaranteed by the bounds inference oracle. The problem posed to this oracle is defined in §2.7 such that safety is provided by construction.

Theorem 2.2 (Scheduling equivalence)**.** *Let $P \in \mathbf{Alg}$ be a valid program and S any valid schedule. Then all target language programs $P' \in \mathbf{BI}(\mathcal{S}(S, \mathcal{L}(P)))$ are confluent with P.*

The preceding property constitutes our soundness invariant. Proving the theorem therefore reduces to showing that this invariant is preserved first by lowering, and then by each subsequent possible primitive scheduling transformation.

2.3.3 Bounds Inference and Language Specification

The definition of algorithm confluence permits the bounds inference oracle to insert assertion checks that might fail into completed programs. This design presents a unique challenge: a completion that *always* fails an assertion check is technically confluent with its original algorithm, but is not useful. Less vacuously, as the bounds inference engine improves, the set of scheduled programs for which we find good—or even just satisfactory—bounds changes. Hopefully, the result is a strict improvement but regressions are possible and even likely in the compiler.

There is a strong case that Halide's design is wrong because there are no guarantees that any given program will continue to work without assertion failures when run on different versions of the standard compiler, much less on alternative implementations. At a minimum, such a fact runs counter to the spirit of specifying programming language behavior. So, we might be tempted to try to re-design Halide.

Instead, we choose to tackle specifying Halide's existing design for two main reasons. First, Halide has been in industrial use for nearly a decade, shipped in many products, and would benefit more from specification of its actual design than of an idealization. Second, flexibility around bounds inference in Halide is essential to array processing and the ability to reason about redundant re-computation and over-computation. An alternative to bounds inference would be intriguing, but also constitute a novel advance in language design on its own.

Our strategy is to supply a baseline, or a lower bound, for the quality of the bounds inference any compliant implementation may have. We supply such a baseline in §2.7.2, specified via a *reference bounds engine*. Bounds-inference implementations must be "at least as good" as the baseline in the following sense:

Definition 2.4 (Bounds engine)**.** Let P be a valid algorithm, S a valid schedule, and $T = \mathcal{S}(S, \mathcal{L}(P))$. A map $\beta : \mathbf{Tgt}^? \to \mathbf{Tgt}$ is a *bounds engine* if $\beta(T) \in \mathbf{BI}(T)$ for any such T.

Definition 2.5 (Bounds quality constraint)**.** Let β_0 be the reference bounds engine given in §2.7.2. Let P be a valid algorithm, S a valid schedule, and z an input. A bounds engine β is compliant with the *bounds quality constraint* if whenever (1) $P(z)$ does not contain an error value, (2) $\beta_0(T)(z)$ does not fail an assertion check, and (3) $\beta_0(T) \simeq_z P$, then $\beta(T) \simeq_z P$.

Thus, any compliant implementation must produce a result on at least the set of programs and schedules accepted by the reference bounds algorithm. In this way, programmers can be assured some degree of portability between different compliant implementations (or across versions of a single implementation). For the existing Halide compiler, this reference method can be used to generate regression tests.

2.4 Algorithm Language

Here we describe the Halide algorithm language, whose purpose is to define the values that the final, scheduled, program must compute. It is a somewhat unusual dataflow language, consisting of *funcs* whose values are computed on-demand by their dependents, and which might have one or more *update stages*, which eagerly and in-place update the func being computed. This scheme preserves referential transparency of funcs, but the resulting mix of eager and lazy semantics complicates any attempt to assign a simple denotation; this is why we use a big-step semantics. Finally, the language is carefully designed with the scheduling language in mind: it underspecifies issues pertaining to bounds and evaluation order, while restricting the dependencies between funcs for the sake of analysis.

R	$::=$	$\mathbf{rdom}(r_1 = I_1, \ldots, r_k = I_k)$	rdom
U_0	$::=$	e	pure stage
U	$::=$	$R\, \mathbf{in}\ (e_1, \ldots, e_n) \leftarrow e\, \mathbf{if}\, e_P$	update stage
B	$::=$	$U_0; U*$	func body
F	$::=$	$\mathbf{fun}\, f(x_1, \ldots, x_n) = \{B\}$	func
D	$::=$	$F \mid D; F$	definitions
P	$::=$	$\mathbf{pipeline}\, f(p_1, \ldots, p_m) = D$	pipeline
Z	$::=$	$P; \mathbf{realize}(z)$	realization
z	$::=$	$\langle (I_1, \ldots, I_n), (c_1, \ldots, c_m) \rangle$	input

(a) Algorithm language

c	$::=$	$i \in \mathbb{V}$	constants
a	$::=$	$f[e_1, \ldots, e_n]$	func access
e	$::=$	$c \mid a \mid v \mid \mathbf{op}(e_1, \ldots, e_k)$	expression
I	$::=$	$(e_{\text{min}}, e_{\text{len}})$	interval
v	$::=$	x	pure variable
	$\mid$	r	reduction variable
	$\mid$	p	parameter variable
$\mathbf{op}$	$::=$	$+ \mid - \mid \times \mid \text{div} \mid \text{mod}$	arithmetic
	$\mid$	$\vee \mid \wedge \mid \neg \mid < \mid > \mid =$	logical
	$\mid$	$\text{select} \mid \text{min} \mid \text{max}$	conditional

(b) Expression language

Figure 2.5: High-level Halide syntax descriptions. These capture the core components of the user-facing algorithm and expression languages in the production Halide language.

2.4.1 Algorithm Terms and Expressions

Our formalization of Halide (syntax in fig. 2.5a) focuses on the fundamental issues at play: pure definitions, separable updates, and imperative updates. Along with pointwise evaluation, these are the primary constructs that govern the structure of computation.

Programmers write *pipelines* P, which are a sequence of *func definitions* F. Each func has some dimension n, associated loop variables $x_1, \ldots, x_n$, and a *body* B. A func body is made up of one or more *stages*. Each stage is made up of a *reduction domain* (or "rdom") R, a predicate e_P, and a rule $(e_1, \ldots, e_n) \leftarrow e$. The first stage $U_0 = e$ is known as the *pure stage* and is equivalent to $\mathbf{rdom}()\, \mathbf{in}\ (x_1, \ldots, x_n) \leftarrow e\, \mathbf{if}\, 1$.

A *reduction domain* repeats the stage rule for a fixed list of variables $r_1, \ldots, r_n$ (not necessarily of the same dimension as the func in which it appears) which range over provided intervals. These model a limited form of imperative updates on a func which happen before *any* other func observes any of its values. The variable r_1 is innermost (changes fastest), while r_n is outermost. As we will see in §2.4.2, there are many restrictions on the form of reduction domains and update stages.

Halide algorithms distinguish variables by their definition sites. The variables that are bound by func definitions are lettered x and are called *pure variables*. The variables bound by rdoms are lettered r and are called *reduction variables*. Finally, variables bound by the top-level pipeline definitions are lettered p are are called *parameter variables*.

These parameters are optional and are always passed *constants*, never other pipelines or funcs. A *realization* Z of a pipeline is a setting of the m parameters, plus n constant intervals over which to evaluate the output func, the last one in the pipeline, which is also named in its signature.

Figure 2.5b shows the syntax of the expression language. The set of values $\mathbb{V} = \mathbb{Z} \cup \{\varepsilon_{\text{rdom}}, \varepsilon_{\text{mem}}\}$ in the formal language extends[2] $\mathbb{Z}$ with special *error values*, which behave as follows:

Definition 2.6 (Error value)**.** The special expression values $\varepsilon_{\text{rdom}} < \varepsilon_{\text{mem}}$ encode a hierarchy of errors. Any operation in the expression language involving one or more of these values evaluates to the *greatest* among them.

Note the omission of arithmetic errors in this definition. These cannot arise because all operations in the expression language are *total*. In particular, division and modulo by zero are both defined to be zero. The reason for this is discussed further in §2.9. The other errors, $\varepsilon_{\text{rdom}}$ and ε_{mem}, respectively capture errors preventing ordinary execution of rdoms (§2.4.3) and *memory* errors, which do not occur in the algorithm semantics. Memory errors are possible in the target language (§2.5.2), but are prohibited by theorem 2.1.

There is no Boolean type in the expression language, so the logical operators interpret their arguments according to the usual convention of using zero to represent “false” and non-zero values to represent “true”. When a logical operator evaluates to “true”, it returns 1, specifically.

Finally, note that the expression language has no short-circuiting semantics. Thus, logical-or and logical-and may not be used to conditionally evaluate points in another func, and the “select” function (the common ternary-if operator) always fully evaluates all three of its arguments.

2.4.2 Algorithm Validity Rules

Halide algorithms must adhere to several non-standard restrictions. This first rule constrains the use of pure variables to facilitate flexible scheduling decisions.

Definition 2.7 (Syntactic separation restriction)**.** Let func f be given by $\textbf{fun}\, f(x_1, \ldots, x_n) = \{U_0; \cdots ; U_m\}$. The *syntactic separation restriction* states that for all pure variables x_i and

[2]The practical system also supports floating-point and fixed-width integers, and faces standard semantic issues with those.

all stages U_j, if x_i occurs anywhere in U_j then all accesses in U_j of the form $f[e_1, \ldots, e_n]$ must have $e_i \equiv x_i$. The update rule $U_j = R\,\textbf{in}\;(\ldots, e_i, \ldots) \leftarrow e\,\textbf{if}\,e_P$ must also have $e_i \equiv x_i$.

This rule is critical to the correctness of many scheduling directives and metatheory claims, but it is quite subtle, so we show a few examples. First, it might be tempting to write an in-place shift using the following func definition:

$$\textbf{fun}\, f(x) = \{g[x]; (x) \leftarrow f[x+1]\}$$

but such an update diverges on f's unbounded domain since $f(0)$ would need to first compute $f(1)$, which would need to compute $f(2)$ and so on. Such updates are disallowed by definition 2.7. It is also disallowed to use the variable in some places, but not others, as in:

$$\textbf{fun}\, f(x) = \{g[x]; \textbf{rdom}(r = (0,3))\,\textbf{in}\;(x) \leftarrow f[x] + f[r]\}$$

The reason here is that, viewed as an in-place *update* to the values of f, the update cannot be applied uniformly across the entire dimension x. On the other hand, a definition like

$$\textbf{fun}\, f(x) = \{0; \textbf{rdom}(r = (0,3))\,\textbf{in}\;(x) \leftarrow f[x] + g[x] + g[r]\}$$

is legal since the restriction only applies to the func whose update stage is being defined. At this point in the algorithm, all of g's values are known, so there is no hazard. Intuitively, updates that reference pure variables should augment the previous stage while remaining *well-founded*.

The syntactic separation restriction extends the notion of purity from variables to stage *dimensions*, which need not reference all of the func's pure variables.

Definition 2.8 (Pure/reduction dimensions)**.** For any pure variable x_i and stage U_j, it is said that i is a *pure dimension* in stage j if x_i appears in U_j. Dimensions which are not pure are called *reduction dimensions*.

Certain expressions in Halide may not refer to pure or reduction variables in order to keep scheduling flexible and sound. Such expressions are called *startup expressions* to reflect the fact that they are constant through the whole execution.

Definition 2.9 (Startup expression)**.** In a pipeline P with parameters $p_1, \ldots, p_n$, an expression e is a *startup expression* iff e contains no func references and any variable v occurring in e is identically one of p_i for some i.

With this definition, we are finally ready to define validity for a program in the algorithm language.

Definition 2.10 (Valid program)**.** A program $P \in \textbf{Alg}$ is *valid* if the bounds of all rdoms are startup expressions, the names of all funcs are unique, the names of pure variables within each func are unique, and the names of reduction variables within a single stage are unique.

All stages must obey the syntactic separation restriction (definition 2.7). The output func f in **pipeline** $f(\dots)$ must exist and be the last func defined. All funcs must be defined before they are referenced by another func. The first stage of every func may not include a self-reference (i.e., must be pure in all dimensions). Lastly, common type checking rules for expressions (eg. func arity) must be respected.

2.4.3 Algorithm Semantics

The purpose of a Halide algorithm is to *define* the value of every point in every func (fig. 2.6). Evaluation proceeds pointwise with no need to track bounds. Funcs are evaluated by substitution [FUNC-EVAL] as is standard for function calls. Compared to the target language, which precomputes values of funcs as if they were arrays, these semantics are *lazy*.

While this laziness avoids reasoning about bounds, it complicates the semantics of the comparatively eager rdom construct. How do we update a func in-place, when it is intuitively meant to be pure? To resolve this tension we simply unroll rdoms [RDOM-EVAL] into sequences of point updates when and as they are encountered.

These simple point updates [UPDATE-EVAL] can then be thought of as shadowing the previous func definition, similar to the functional definition of stores used by most operational semantics for imperative languages. If the lookup point and update point coincide, then the update rule is substituted, otherwise the existing value is used.

Lastly, we note that all valid algorithms terminate. This follows the intuition that Halide pipelines are defining mathematical objects by supplying formulas to compute the values.

Lemma 2.3 (Algorithms terminate)**.** *Given any algorithm $P \in$ **Alg** and input z, the output of $P(z)$ can be determined in a finite amount of time.*

Proof. Since rdom bounds are *startup expressions* (definition 2.9) and no infinity value exists in $\mathbb{V}$, there is no way to loop infinitely. The program validity checks (definition 2.10) prevent self-recursion in the function definitions. Functions must be declared before they are used, so recursion is impossible. Thus, Halide algorithms always terminate. In fact, this also shows Halide is not Turing-complete. □

2.5 Target Language

In this section, we describe the target language (IR) to which the algorithm language compiles. Unlike the algorithm language, it is similar to classic imperative languages, and programs in this language have a defined execution order (which is modified by the schedule). It uses the same expression language from §2.4.1 and has the same semantics for all expressions, save func accesses, which become references to memory.

$$\frac{\forall y = (i_1, \ldots, i_n) \in I_1 \times \cdots \times I_n : \left[\overline{c/p}\right](D; f[i_1, \ldots, i_n]) \Downarrow c_y}{\textbf{pipeline}\, f(\overline{p}) = D; \textbf{realize}\left(\overline{I}, \overline{c}\right) \Downarrow g(y) := c_y}\ [\textsc{realize}]$$

$$\frac{}{D; c \Downarrow c}\ [\textsc{const-eval}] \qquad \frac{\left(\overline{D; e \Downarrow c}\right) \qquad D; f[\overline{c}] \Downarrow c'}{D; f[\overline{e}] \Downarrow c'}\ [\textsc{func-arg-eval}]$$

$$\frac{\left(\overline{D; e \Downarrow c}\right) \qquad D; \textbf{op}(\overline{c}) \Downarrow c'}{D; \textbf{op}(\overline{e}) \Downarrow c'}\ [\textsc{op-eval}] \qquad \frac{f \neq g \qquad D; f[\overline{c}] \Downarrow c'}{D; \textbf{fun}\, g[\overline{x}] = \{B\}; f[\overline{c}] \Downarrow c'}\ [\textsc{func-skip}]$$

$$\frac{D; \left[\overline{c/x}\right] e \Downarrow c'}{D; \textbf{fun}\, f[\overline{x}] = \{e\}; f[\overline{c}] \Downarrow c'}\ [\textsc{func-eval}]$$

$$\frac{D; \textbf{fun}\, f[\overline{x}] = \{\overline{U}\}; \left[\overline{c/x}\right](\textbf{select}(\overline{e = x} \wedge e_p, e_b, f[\overline{x}])) \Downarrow c'}{D; \textbf{fun}\, f[\overline{x}] = \{\overline{U}; \textbf{rdom}()\ \textbf{in}\ \ (\overline{e}) \leftarrow e_b\ \textbf{if}\ e_p\}; f[\overline{c}] \Downarrow c'}\ [\textsc{update-eval}]$$

$$\frac{\left(\overline{I = \langle e^{\mathbf{min}}, e^{\mathbf{len}}\rangle}\right) \qquad \left(\overline{e^{\mathbf{min}} \Downarrow c^{\mathbf{min}}}\right) \qquad \left(\overline{e^{\mathbf{len}} \Downarrow c^{\mathbf{len}}}\right) \qquad \exists j. c_j^{\mathbf{len}} < 0}{D; \textbf{fun}\, f[\overline{x}] = \{\cdots; \textbf{rdom}(\overline{r = I})\ \textbf{in} \cdots\}; f[\overline{c}] \Downarrow \varepsilon_{\text{rdom}}}\ [\textsc{rdom-err}]$$

$$\frac{\begin{array}{c}\left(\overline{I = \langle e^{\mathbf{min}}, e^{\mathbf{len}}\rangle}\right) \qquad \left(\overline{e^{\mathbf{min}} \Downarrow c^{\mathbf{min}}}\right) \qquad \left(\overline{e^{\mathbf{len}} \Downarrow c^{\mathbf{len}}}\right) \\ D; \textbf{fun}\, f[\overline{x}] = \{\overline{U}; \mathit{unroll}\}; f[\overline{c}] \Downarrow c'\end{array}}{D; \textbf{fun}\, f[\overline{x}] = \{\overline{U}; \textbf{rdom}(\overline{r = I})\ \textbf{in}\ \ (\overline{e}) \leftarrow e_b\ \textbf{if}\ e_p\}; f[\overline{c}] \Downarrow c'}\ [\textsc{rdom-eval}]$$

$$\textit{where}$$

$$\mathit{unroll} = \left(\begin{array}{c}\left[c_k^{\mathbf{min}}/r_k\right](\textbf{rdom}(r_1 = I_1, \ldots, r_{k-1} = I_{k-1})\ \textbf{in}\ \ (\overline{e}) \leftarrow e_b\ \textbf{if}\ e_p)\,; \\ \vdots \\ \left[\left(c_k^{\mathbf{min}} + c_k^{\mathbf{len}} - 1\right)/r_k\right](\cdots)\end{array}\right)$$

Figure 2.6: Algorithm language natural semantics. Note that we use a metasyntactic notation, $\overline{\ \cdot\ }$, which indicates that the covered expression is repeated for each numerical subscript, e.g. $(\overline{e = x} \wedge e_p) \equiv (e_1 = x_1 \wedge \cdots \wedge e_n = x_n \wedge e_p)$. Parentheses distinguish such terms from axioms. The [REALIZE] rule defines the points of a partial function $g : (I_1 \times \cdots \times I_n) \to \mathbb{V}$.

τ	::=	serial | parallel	traversal order
s	::=	**nop**	no operation
	|	**assert** e	assertion
	|	s_1 ; s_2	sequencing
	|	**allocate** $f(I_1, \ldots, I_n)$	allocate buffer
	|	$f^\ell[e_1, \ldots, e_m] \leftarrow e_0$	update buffer
	|	**if** e_1 **then** s_1 **else** s_2	branching
	|	**for**$^\tau$ x **in** I **do** s	bounded loops
	|	**let** $x = e$ **in** s	let binding
	|	**label** $\ell : s$	statement label
P	::=	**pipeline** $f(p_1, \ldots, p_m)$: s	pipeline
e	::=	$\ldots \mid ?\ell \mid f^\ell[\ldots]$	$\mathbf{Tgt}^?$ expr

Figure 2.7: Halide IR syntax. Expressions and realizations are the same as in fig. 2.5b, but are augmented with labeled holes for $\mathbf{Tgt}^?$. Labels ℓ are arbitrary and left uninterpreted. Statement labels have no special semantics.

2.5.1 Syntax

Figure 2.7 presents the abstract syntax for the Halide IR. This language comes in two variants: *with* holes ($\mathbf{Tgt}^?$) and *without* holes ($\mathbf{Tgt}$). The lowering algorithm given in §2.6 translates an algorithm to a program in $\mathbf{Tgt}^?$ whose holes will be filled by *bounds inference* (§2.7). The main difference between $\mathbf{Tgt}$ and similar imperative languages is that loops are restricted to range-based for loops which can be marked for parallel traversal. Furthermore, these ranges are given as *minimum* and *length* pairs, rather than minimum and *maximum*. Some syntax may be annotated with *labels*, written ℓ. Labels are ignored by the semantics because they are simply used as handles by the *scheduling* (§2.8) and bounds inference systems.

2.5.2 Semantics

In fig. 2.8 we give small-step semantics for the IR. Note that Σ is an *environment* for loop variables and let bindings and σ is the *store* or *heap* in which memory is allocated.

These semantics are mostly standard, though there are a few instances where the semantics can get *stuck*. We enumerate and define all the failure modes here:

Definition 2.11 (Assertion failure)**.** If the execution of a program $P \in \mathbf{Tgt}$ gets stuck when an assertion fails (i.e. the condition evaluates to 0), then we say P has failed an assertion check.

Definition 2.12 (RDom failure)**.** If the execution of a program $P \in \mathbf{Tgt}$ gets stuck because a for loop has a negative extent, then we say P has encountered an *rdom failure*. This

$$
\begin{array}{rcl}
E & ::= & \Box \\
& | & E; s \\
& | & \textbf{allocate}\, f((c_1^{\min}, c_1^{\text{len}}), \ldots, E, \ldots, I_n) \\
& | & a \leftarrow E \\
& | & E \leftarrow c \\
& | & \textbf{if}\, E\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \\
& | & \textbf{for}^{\ell}\, x\, \textbf{in}\, E\, \textbf{do}\, s \\
& | & \textbf{let}\, x = E\, \textbf{in}\, s \\
& | & f[c_1, \ldots, E, \ldots, e_n] \\
& | & \textbf{op}(c_1, \ldots, E, \ldots, e_k) \\
& | & (E, e^{\text{len}}) \\
& | & (c^{\min}, E) \\
& | & \textbf{assert}\, E \\
& | & \textbf{pipeline}\, f(\overline{p}) : E
\end{array}
$$

$$\frac{\langle s \mid \Sigma \mid \sigma\rangle \to \langle s' \mid \Sigma' \mid \sigma'\rangle}{\langle E[s] \mid \Sigma \mid \sigma\rangle \to \langle E[s'] \mid \Sigma' \mid \sigma'\rangle}\ [\textsc{Reduce}]$$

$$\frac{v = \Sigma(x)}{\langle x \mid \Sigma \mid \sigma\rangle \to \langle v \mid \Sigma \mid \sigma\rangle}\ [\textsc{Var}]$$

$$\frac{}{\langle \textbf{nop}; s \mid \Sigma \mid \sigma\rangle \to \langle s \mid \Sigma \mid \sigma\rangle}\ [\textsc{Nop}]$$

$$\frac{c \neq 0}{\langle \textbf{assert}\, c \mid \Sigma \mid \sigma\rangle \to \langle \textbf{nop} \mid \Sigma \mid \sigma\rangle}\ [\textsc{Assert-True}]$$

$$\frac{}{\langle \textbf{let}\, x = c\, \textbf{in}\, s \mid \Sigma \mid \sigma\rangle \to \langle s \mid \Sigma[x = c] \mid \sigma\rangle}\ [\textsc{Let}]$$

$$\frac{}{\langle \textbf{pipeline}\, f(\overline{p}) : s; \textbf{realize}(\overline{c}) \mid \Sigma \mid \sigma\rangle \to \langle \textbf{pipeline}\, f(\overline{p}) : s \mid \Sigma\, [\overline{p = c}] \mid \sigma\rangle}\ [\textsc{Realize}]$$

$$\frac{\sigma(f) = \left\langle \hat{f}, \ldots \right\rangle}{\langle \textbf{pipeline}\, f(\overline{p}) : \textbf{nop} \mid \Sigma \mid \sigma\rangle \to \left\langle \hat{f} \,\middle|\, \Sigma \,\middle|\, \sigma \right\rangle}\ [\textsc{End}]$$

$$\frac{c^{\text{len}} > 0}{\left\langle \textbf{for}\, x\, \textbf{in}\ (c^{\min}, c^{\text{len}})\, \textbf{do}\, s \,\middle|\, \Sigma \,\middle|\, \sigma \right\rangle \to \left\langle s; \textbf{for}\, x\, \textbf{in}\ (c^{\min}+1, c^{\text{len}}-1)\, \textbf{do}\, s \,\middle|\, \Sigma[x = c^{\min}] \,\middle|\, \sigma \right\rangle}\ [\textsc{For-Iter}]$$

$$\frac{}{\left\langle \textbf{for}\, x\, \textbf{in}\ (c^{\min}, 0)\, \textbf{do}\, s \,\middle|\, \Sigma \,\middle|\, \sigma \right\rangle \to \langle \textbf{nop} \mid \Sigma \setminus \{x\} \mid \sigma\rangle}\ [\textsc{For-Stop}]$$

$$\frac{c \neq 0}{\langle \textbf{if}\, c\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \mid \Sigma \mid \sigma\rangle \to \langle s_1 \mid \Sigma \mid \sigma\rangle}\ [\textsc{If-T}] \qquad \frac{}{\langle \textbf{if}\, 0\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \mid \Sigma \mid \sigma\rangle \to \langle s_2 \mid \Sigma \mid \sigma\rangle}\ [\textsc{If-F}]$$

$$\frac{\text{InBounds}(f[\overline{c}], \sigma) \qquad \sigma' = \sigma[f(\overline{c}) = c']}{\langle f[\overline{c}] \leftarrow c' \mid \Sigma \mid \sigma\rangle \to \langle \textbf{nop} \mid \Sigma \mid \sigma'\rangle}\ [\textsc{Assn}] \qquad \frac{\sigma' = \sigma\left[f \to \left\langle \lambda\overline{x}.\ \varepsilon_{\text{mem}},\ \overline{I}\right\rangle\right]}{\left\langle \textbf{allocate}\, f\, (\overline{I}) \,\middle|\, \Sigma \,\middle|\, \sigma \right\rangle \to \langle \textbf{nop} \mid \Sigma \mid \sigma'\rangle}\ [\textsc{Alloc}]$$

$$\frac{\text{InBounds}(f[\overline{c}], \sigma)}{\langle f[\overline{c}] \mid \Sigma \mid \sigma\rangle \to \langle \sigma(f[\overline{c}]) \mid \Sigma \mid \sigma\rangle}\ [\textsc{Read}] \qquad \frac{\textbf{op}(\overline{c}) = c'}{\langle \textbf{op}(\overline{c}) \mid \Sigma \mid \sigma\rangle \to \langle c' \mid \Sigma \mid \sigma\rangle}\ [\textsc{Eval}]$$

Figure 2.8: Structural semantics for **Tgt** (*without* holes). Notice that there are four states that can get *stuck*: (1) when **assert false** is encountered, (2) when a **for** loop extent is negative, (3) when a read occurs out of bounds, and (4) when an assignment occurs out of bounds. The latter two memory errors cannot happen in programs derived from the scheduling and bounds inference processes by theorem 2.1.

corresponds to the failure mode in the algorithm semantics (§2.4.3) where an invalid rdom causes the program to return $\varepsilon_{\text{rdom}}$ everywhere.

Definition 2.13 (Memory error)**.** Recall that the [Read] and [Assn] rules assume their accesses are in bounds. If the execution of a program $P \in \mathbf{Tgt}$ gets stuck when accessing memory, we say P has attempted an *out of bounds* access or has encountered a *memory error*.

Recall that theorem 2.1 states that memory errors cannot occur in the execution of a program which was derived from an algorithm via lowering, scheduling, and bounds inference.

The [ALLOC] rule updates the store σ with a mapping from the *symbolic name* of the func to a pair of (1) a partial function $\hat{f}$ (initially ε_{mem} everywhere) that records the values and (2) the bounds that were stated at allocation time. The predicate $\text{InBounds}(f[\overline{c}], \sigma)$ uses this data to check the fully evaluated point $\overline{c} \equiv (c_1, \ldots, c_n)$ against the bounds stored in $\sigma(f)$.

[ASSN] defines assigning to a point in a func in the store and [READ] defines reading from a func in the store. Assignment is modeled by *shadowing* the old value, ie. by redefining the mapping of f in σ to a new partial function $\hat{f}'$ which agrees with $\hat{f}$ everywhere except at the point being updated. We use the terse syntax $\sigma' = \sigma[f(\overline{c}) = c']$ to denote this operation. Reading a value from a func is then a matter of simply evaluating the stored function.

2.6 Lowering

Halide algorithms are compiled to IR programs with holes by the *lowering* function $\mathcal{L}$, defined in fig. 2.9. The lowering function creates a sequence of top-level loop nests for every func in the program. Inside these loops are assignments implementing the formulas for each stage in the algorithm. Pure dimensions which do not appear in a stage are not lowered, and reduction domains appear as innermost loops.

The lowering function also annotates certain fragments with *labels* to facilitate scheduling and bounds inference. These labels appear in three places: first, they appear in **label** statements which act as handles for the scheduling directives; second, they are attached to the cpu and mem *bounds holes*; finally, they are attached to func references. The following lemma captures the structural invariant provided by the first set of these labels.

Lemma 2.4 (Loop naming)**.** *Given a valid algorithm $P \in$ **Alg** and a valid schedule $S \in$ **Sched**, any for loop in $\mathcal{S}(S, \mathcal{L}(P))$ is uniquely identified by (1) the func, (2) the specialization (or lack thereof, see §2.8.1), and (3) the stage to which it belongs, as well as (4) the name of its induction variable.*

Specializations do not exist in initially lowered programs, but are a scheduling feature (see §2.8.1) that enables replicating code behind one or more *branches*, each guarded by a predicate. Each branch can be scheduled independently, and its predicate is used to simplify the body. A common use case is to specialize a pipeline to common input sizes and reduce bounds computations. If a func is not specialized, that data can be regarded as 0. In any case, lemma 2.4 lets us relate syntax fragments in the IR to their *provenance* in the original

$$
\begin{aligned}
\mathcal{L}(\mathbf{pipeline}\, f(\overline{p}) : \overline{F}; \mathbf{fun}\, f(\overline{x}) = B) &= \begin{cases} \mathbf{pipeline}\, f\left(\overline{p}, \overline{x^{\min}, x^{\mathrm{len}}}\right) : \\ \quad \overline{\mathcal{L}(F)}; \mathcal{L}(\mathbf{fun}\, f(\overline{x}) = B) \end{cases} \\
\mathcal{L}(\mathbf{fun}\, f(\overline{x}) = B) &= \begin{cases} \mathbf{allocate}\, f\left(\overline{?^{\mathrm{mem}} f_x}\right) \\ \mathbf{label}\, f : \mathcal{L}_B(f, \overline{x}, B) \end{cases} \\
\mathcal{L}_B(f, \overline{x}, U_0; \cdots; U_m) &= \begin{cases} \mathbf{label}\, s_0 : \mathcal{L}_U(f, \overline{x}, 0, U_0) \\ \quad \vdots \\ \mathbf{label}\, s_m : \mathcal{L}_U(f, \overline{x}, m, U_m) \end{cases} \\
\mathcal{L}_U(f, \overline{x}, i, R\, \mathbf{in}\, \overline{e} = e_B\, \mathbf{if}\, e_P) &= \mathcal{L}_P(f, \overline{x}, \overline{e}, i, \mathcal{L}_R(R, \mathbf{if}\, e_P\, \mathbf{then}\, f^i[\overline{e}] \leftarrow e_B)) \\
\mathcal{L}_R(\mathbf{rdom}(), s) &= s \\
\mathcal{L}_R(\mathbf{rdom}(r_1 = I_1, \overline{r = I}), s) &= \mathcal{L}_R(\mathbf{rdom}(\overline{r = I}), \mathbf{for}\, r_1\, \mathbf{in}\, I_1\, \mathbf{do}\, s) \\
\mathcal{L}_P(f, (), (), i, s) &= [f^{i-1}/f]s \\
\mathcal{L}_P(f, (x, \overline{y}), (e, \overline{e'}), i, s) &= \begin{cases} \mathcal{L}_P\left(f, \overline{y}, \overline{e'}, i, \mathbf{for}\, x\, \mathbf{in}\ ?^{\mathrm{cpu}} f_x^i\, \mathbf{do}\, s\right) & \text{if } x \equiv e \\ \mathcal{L}_P(f, \overline{y}, \overline{e'}, i, s) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.9: Lowering algorithm with default eager schedule.

algorithm. The following lemma uses this to state that funcs are computed and allocated in a valid order in the IR.

Lemma 2.5 (Dominance)**.** *Let $P \in \mathbf{Alg}$ and $S \in \mathbf{Sched}$ be a valid algorithm and schedule, and let $P' = \mathcal{S}(S, \mathcal{L}(P))$. If a func f appears in the definition of a func g in P, then the loops for f* dominate *the assignment statement for g in P'. Furthermore, the* **allocate** *statement for any func f dominates the loops for f in P'.*

The previous two lemmas hold just after lowering by construction. Each scheduling directive needs to show that it maintains these invariants. Lowering also introduces a set of labeled *bounds holes*, which will be filled by the *bounds inference* oracle (§2.7), and which carry the following data.

Definition 2.14 (Bounds hole)**.** A bounds hole is an entity in the expression language of $\mathbf{Tgt}^?$ that stands in for a hole-free expression. A bounds hole is labeled by (1) whether it is an allocation hole (mem) or a compute hole (cpu), (2) whether it represents the minimum (min) of an interval, or its length (len), (3) the associated func and dimension, and (4) if it is a compute hole, the associated stage and specialization.

Across specializations, the last stage of a given func always uses a common bounds hole. We omit the stage number when referring to the *last* stage of a func and we omit the specialization number when the func is not specialized. Finally, we write $?^{\mathrm{mem}} f_x = \left[(?^{\mathrm{mem}} f_x)^{\min}, (?^{\mathrm{mem}} f_x)^{\mathrm{len}}\right]$

$$
\begin{aligned}
\mathcal{B}(\textbf{pipeline}\, f(\overline{p}, \overline{x^{\min}, x^{\mathrm{len}}}) : s) &= \ldots \\
\overline{\forall p \in [-\infty,\infty]} : \mathcal{B}(s) &\wedge \overline{[x^{\min}, x^{\mathrm{len}}] \in ?^{\mathrm{cpu}} f} \\
\mathcal{B}(\textbf{assert}\, e) &= e \\
\mathcal{B}(\textbf{nop}) &= \textbf{true} \\
\mathcal{B}(\textbf{allocate}\, f(\ldots); s) &= \exists ?^{\mathrm{mem}} f : \mathcal{B}(s) \\
\mathcal{B}(s_1; s_2) &= \mathcal{B}(s_1) \wedge \mathcal{B}(s_2) \\
\mathcal{B}(\textbf{label}\, f : s) &= \exists ?^{\mathrm{cpu}} f : \mathcal{B}(s) \\
\mathcal{B}(\textbf{let}\, v = e \,\textbf{in}\, s) &= \mathcal{B}_{\mathrm{cpu}}(e) \wedge \textbf{let}\, v = e \,\textbf{in}\, \mathcal{B}(s) \\
\mathcal{B}(\textbf{if}\, e \,\textbf{then}\, s_1 \,\textbf{else}\, s_2) &= \ldots \\
&\mathcal{B}_{\mathrm{cpu}}(e) \wedge e \Rightarrow \mathcal{B}(s_1) \wedge \neg e \Rightarrow \mathcal{B}(s_2) \\
\mathcal{B}(\textbf{for}\, v \,\textbf{in} [e^{\min}, e^{\mathrm{len}}] \,\textbf{do}\, s) &= \ldots \\
&e^{\mathrm{len}} \geq 0 \wedge \forall v \in [e^{\min}, e^{\mathrm{len}}] : \mathcal{B}(s) \\
\mathcal{B}\left(f^{i,j}[\overline{e}] \leftarrow e_0\right) &= \overline{e} \in ?^{\mathrm{mem}} f \wedge \overline{\mathcal{B}_{\mathrm{cpu}}(e)} \wedge \mathcal{B}_{\mathrm{mem}}(e_0) \\
&\wedge \overline{e} \in ?^{\mathrm{cpu}} f^{i,j} \Rightarrow \mathcal{B}_{\mathrm{cpu}}(e_0)
\end{aligned}
$$

where

$$\mathcal{B}_{\mathrm{cpu}}(f^{i,j}[\overline{e}]) = \overline{\mathcal{B}_{\mathrm{cpu}}(e)} \wedge \overline{e} \in ?^{\mathrm{mem}} f \wedge \overline{e} \in ?^{\mathrm{cpu}} f^{i,j}$$

$$\mathcal{B}_{\mathrm{mem}}(f[\overline{e}]) = \overline{\mathcal{B}_{\mathrm{cpu}}(e)} \wedge \overline{e} \in ?^{\mathrm{mem}} f$$

Remaining cases for $\mathcal{B}_{\mathrm{cpu}}, \mathcal{B}_{\mathrm{mem}}$ *fold with union.*

(a) Query extraction function $p = \mathcal{B}(T)$ produces predicate from program $T \in \mathbf{Tgt}^{?}$.

$$
\begin{aligned}
\beta_0(\forall v \in I : b, \Gamma) &= \beta_0(b, \Gamma[v \mapsto I]) \\
\beta_0(b_1 \wedge b_2, \Gamma) &= \beta_0(b_1, \beta_0(b_2, \Gamma)) \\
\beta_0(e \Rightarrow b, \Gamma) &= \beta_0(b, \Gamma) \\
\beta_0\left(\exists\, ?^{\ell} : b, \Gamma\right) &= \beta_0(b, \Gamma) \\
\beta_0(\textbf{let}\, v = e \,\textbf{in}\, b, \Gamma) &= \beta_0(b, \Gamma[v \mapsto \mathcal{I}(e, \Gamma)]) \\
\beta_0\left(\overline{e} \in\ ?^{\ell}, \Gamma\right) &= \Gamma\left[?^{\ell} \mapsto \Gamma(?^{\ell}) \cup \overline{\mathcal{I}(e,\Gamma)}\right]
\end{aligned}
$$

$\boxed{\text{Note: } [\omega_i^{\min}, \omega_i^{\mathrm{len}}] = \mathcal{I}(e_i) \text{ below.}}$

$$\mathcal{I}(e_1 + e_2, \Gamma) = [\omega_1^{\min} + \omega_2^{\min}, \omega_1^{\mathrm{len}} + \omega_2^{\mathrm{len}} - 1]$$

(standard interval arithmetic rules elided)

$$\mathcal{I}(e_1 \,\mathrm{div}\, e_2, \Gamma) = [-M, 2M - 1]$$

$$\textit{where } M = \max(-\omega_1^{\min}, \omega_1^{\min} + \omega_1^{\mathrm{len}} - 1)$$

$$
\begin{aligned}
\mathcal{I}\left(f^{\ell}[\overline{e}], \Gamma\right) &= [-\infty, \infty] \\
\mathcal{I}(v, \Gamma) &= \Gamma(v) \\
\mathcal{I}(c, \Gamma) &= [c, 1]
\end{aligned}
$$

(b) Baseline bounds engine $\beta_0(p)$ applies naive interval arithmetic rules to queries produced by $\mathcal{B}$.

Figure 2.10: Overview of the bounds inference system, showing query extraction and the baseline bounds engine β_0.

for the allocation bounds interval for func f, dimension x. By analogy, $?^{\mathrm{cpu}} f_x^{i,j}$ denotes the compute bounds interval for func f, dimension x, stage i, and specialization j.

Finally, the labels attached to func references ($f^{\ell}[\ldots]$) record the *previous* stage and current specialization. This helps the bounds extraction procedure (§2.7) construct the necessary predicates to ensure safety and correctness.

2.7 Bounds Inference

Previous work on Halide discusses bounds inference in terms of a particular algorithm used to fill the bounds holes. Improvements to the compiler regularly change the results of this algorithm, resulting in an unstable definition in practice.

In order to abstract over the ever-changing bounds inference algorithm, we pose bounds inference as a *program synthesis* problem via an oracle query. While the resulting satisfiability problem is undecidable in general, this definition provides previously underformulated soundness conditions for any bounds inference algorithm. Queries to this oracle are defined as follows:

Definition 2.15 (Bounds oracle query)**.** Let $P \in \mathbf{Alg}$ be an algorithm and let $S \in \mathbf{Sched}$ be a schedule for it so $T = \mathcal{S}(S, \mathcal{L}(P))$. Then a query to the *bounds oracle* $\mathcal{O}$ is the *predicate* $p = \mathcal{B}(T)$. The oracle responds with some set of hole substitutions $\Gamma \in \mathcal{O}(p)$ that is compatible with T. Hence, the set $\mathbf{BI}(T) = \{[\Gamma]T \mid \Gamma \in \mathcal{O}(\mathcal{B}(T))\}$.

Recall from definition 2.14 that there are two kinds of bounds. The *compute bounds* define regions over which the points in the buffers must have non-error values that agree with those defined by the original algorithm. The *allocation bounds* enclose the compute bounds, and further includes at least all points read from or written to. As we saw in the example (§2.2), this gap can be exploited by overcompute strategies during scheduling (§2.8.2).

2.7.1 Bounds constraint extraction

The algorithm for extracting the bounds constraints for a program $T \in \mathbf{Tgt}^?$ is shown in fig. 2.10a. The extraction traverses the AST of the program and translates every statement into a logical condition with existentially quantified holes.

This extraction encodes a few important correctness conditions. First, if a point being computed lies in the compute bounds, then all of the accesses on the right hand side of the assignment must be in the compute bounds of their funcs. (What happens outside the compute bounds stays outside the compute bounds.) Second, accesses occurring anywhere inside an expression that is used for *indexing* or *branching* must be in the compute bounds as well. Finally, every point that is read anywhere in the program must at least be in the allocation bounds, in order to preserve memory safety.

This second point is particularly important: splitting loops in data-dependent update stages (such as when computing a histogram) will introduce **if** statements whose values must not be errors resulting from reading uninitialized memory. The rule for **let** is similarly motivated; **let** expressions are only introduced by scheduling directives to hold expressions used for indexing (§2.8), so accesses there must be in the compute bounds.

2.7.2 Reference algorithm

Figure 2.10b gives the baseline bounds inference algorithm β_0. It works by scanning the extracted constraint and performing interval arithmetic (via $\mathcal{I}$) on the terms, naively trying to symbolically satisfy the consequent of each implication without using its predicate (ie. unconditionally). β_0 merges these intervals to determine safe coverings for each hole. Because the constraint is extracted from the fully scheduled target program, it can rely on the association order of $\wedge$ to reflect the sequencing order in the original program and ensure that we make inferences about holes backwards through the dependencies. Since β_0 only produces a list of substitutions, it does not meet the bounds engine definition (definition 2.4) on its own. However, it is easily lifted to a bounds engine by applying the substitutions whenever every hole is determined and no $\pm\infty$ appears in the substitutions. When this is not the case, it simply fails by replacing the body with **assert false**.

Beyond the naïvety of the algorithm, interval arithmetic has an inherent *dependency problem*. The classic example is $x^2 + x$, where $x \in [-1, 1]$ and so $x^2 \in [0, 1]$. Adding these bounds gives $[-1, 2]$, which is slightly wider than the true bounds: $[-\frac{1}{4}, 2]$. This is because interval arithmetic treats $x^2 + x$ as $x^2 + y$, where y varies *independently* over the same interval as x. These errors can accumulate rapidly as expressions grow larger.

The algorithm β_0 is only meant to be a *baseline*; and, although it is quite naïve, it still identifies tight bounds for the example in §2.2. The practical system contains many improvements over this, including analyses of function value ranges, of correlated differences and sums, and of conditionally-correct simplifications backed by path-sensitive analysis.

2.7.3 Metatheory

Finally, we state the main lemmas concerning the structure of solutions to the bounds inference problem.

Lemma 2.6 (Memory safety)**.** *All programs resulting from bounds inference $P' \in \mathbf{BI}(T)$, are memory safe.*

Lemma 2.7 (Compute bounds confluent)**.** *Let $P \in \mathbf{Alg}$, $P' \in \mathbf{BI}(\mathcal{L}(P))$, and let f be a func in P. If all of the points in compute bounds of funcs preceding f are confluent with P, then the loop nest for f computes values confluent with P.*

The proofs of these lemmas are deferred to the appendices. Together, they form the base case of the inductive proof that the scheduling directives are sound.

2.8 Scheduling Language

We formalize scheduling by directly mutating programs in $\mathbf{Tgt}^?$. Because some directives — like split — must be applied after certain other directives, we require that schedules be ordered

S	$::=$	$s_1; \ldots ; s_n$	schedule program
ℓ	$::=$	$\langle f, i, j, v \rangle$	loop names (lemma 2.4)
τ	$::=$	**serial** | **parallel**	traversal orders
φ	$::=$	$\varphi_{\text{Guard}} \mid \varphi_{\text{Shift}} \mid \varphi_{\text{Round}}$	split strategies
s	$::=$	**specialize**$(f, e_1, \ldots, e_n)$	Specialization (§2.8.1)
	|	**split**$(\ell, x_o, x_i, e, \varphi)$	Loops (§2.8.2)
	|	**fuse**(ℓ, x)	
	|	**swap**(ℓ)	
	|	**traverse**(ℓ, τ)	
	|	**compute-at**(f, ℓ_g)	Compute (§2.8.3)
	|	**store-at**(f, ℓ_g)	Storage (§2.8.4)
	|	**bound**$(f, x, e^{\text{min}}, e^{\text{len}})$	Bounds (§2.8.5)
	|	**bound-extent**(f, x, e^{len})	
	|	**align-bounds**(f, x, e^m, e^r)	

Figure 2.11: The Halide scheduling language. The s definition is grouped by phase of scheduling (presented in order).

into phases[3] as indicated in fig. 2.11. Scheduling directives use loop names to determine their targets.

In fig. 2.12 we show the IR transformations for each scheduling directive. In each subsequent section, we describe each phase and enumerate its restrictions, but defer safety proofs to the appendices. For each phase, we require an inductive lemma like the following:

Lemma 2.8 (Scheduling phase is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a scheduling directive in this phase, then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

2.8.1 Specialization Phase

Certain scheduling decisions may be more or less efficient, depending on program parameters. For instance, simpler schedules tend to work better for small output sizes.

Specialization duplicates an existing func's code for each of n conditions, and introduces labels that allow later scheduling directives to operate differently on each instance. These conditions, like all expressions in the scheduling language, are required to be *start-up expressions*. In our formal system, schedules may give at most one specialization directive per func. The following lemma captures an essential property of specializations, namely that only one specialization is "active" during any given run.

[3]The practical system sorts directives into phases automatically.

Lemma 2.9 (Unique active specialization)**.** *Given algorithm $P \in \mathbf{Alg}$ and a schedule $S \in \mathbf{Sched}$, let $P' \in \mathbf{BI}(\mathcal{S}(S, \mathcal{L}(P)))$. Then for any input z, $P'(z)$ will evaluate exactly one specialization for any given func f.*

It is also important to note that the transformation attaches the specialization instance to func references inside the copied (and original) statements. As per definition 2.14, the rule in fig. 2.12 should be interpreted to exclude the final stage when attaching this information.

2.8.2 Loops Phase

Halide provides several standard loop transformations to change the order of computations. A loop can be *split* into two nested loops, two nested loops can be *fused* into a single loop, a loop may be *swapped* with the immediately nested loop, and loops may be traversed in *parallel*. Swapping and parallelization apply only to *pure loops*, a manifestation of pure dimensions in the target IR. We define these here:

Definition 2.16 (Pure loop)**.** Let v be the loop variable for some loop in a program $P \in \mathbf{Tgt}^?$. We say v and its associated loop are *pure* if v is one of the pure dimensions of its associated func, if it is the result of splitting a pure loop, or if it is the result of fusing two pure loops together. We letter the iteration variable of pure loops x. All other loops are *reduction loops*, lettered r.

We may **split** a loop ℓ by a *split factor* of e into an outer loop iterated by x_o and inner loop iterated by x_i. This division may produce a remainder, which is handled by choice of a *tail strategy* (denoted φ): (1) *guard*ing the body with an **if**; (3) *shift*ing the last loop iteration inwards, causing recomputation; or (3) *round*ing the loop bounds upward, causing overcomputation of the func and affecting upstream bounds. Shifting and rounding are only allowed on pure stages.

Two nested loops can be *fused* together into a single loop whose extent is the product of the original extents, provided both loops are pure or both are reduction loops. This is approximately an inverse to the split directive, and is useful for controlling the granularity of parallelism. Immediately nested loops can be *swapped* as long as the swap does not reorder two reduction loops. Finally, each pure loop can also be traversed in either *serial* or *parallel* order. All variable names introduced by these directives must be new, unique, and non-conflicting.

2.8.3 Compute Phase

To narrow the scope of computation, the **label**ed statement for computing a func f may be moved from the top level to just inside any loop as long as the **label**ed statement continues to dominate all external accesses to f.

The closer a producer is computed to its consumer, the less of the producer needs to be computed per iteration of the consumer. The expectation is that bounds inference will use

$$\begin{array}{l} \qquad\quad \textbf{label}\ f : s \\ \qquad \xrightarrow{\textbf{specialize}(f,e_1,\ldots,e_n)} \\ \textbf{label}\ f : \\ \quad \textbf{if}\ e_1\ \textbf{then}\ \textbf{label}\ z_1 : [f^{i,1}/f^i]s \\ \quad \vdots \\ \quad \textbf{else if}\ e_n\ \textbf{then}\ \textbf{label}\ z_n : [f^{i,n}/f^i]s \\ \quad \textbf{else}\ \textbf{label}\ z_0 : [f^{i,0}/f^i]s \end{array}$$

$$\begin{array}{l} \textbf{for}\ x\ \textbf{in}\ b\ \textbf{do} \\ \quad \textbf{for}\ x'\ \textbf{in}\ b'\ \textbf{do}\ s \\ \quad \xrightarrow{\textbf{swap}(\ell)} \\ \textbf{for}\ x'\ \textbf{in}\ b'\ \textbf{do} \\ \quad \textbf{for}\ x\ \textbf{in}\ b\ \textbf{do}\ s \end{array} \qquad \begin{array}{c} \textbf{for}\ x\ \textbf{in}\ b\ \textbf{do}\ s \\ \xrightarrow{\textbf{traverse}(\ell,\tau)} \\ \textbf{for}^{\tau}\ x\ \textbf{in}\ b\ \textbf{do}\ s \end{array}$$

$$\begin{array}{l} \textbf{label}\ f : s_1\ \ldots \\ \textbf{label}\ g :\ \ldots \\ \quad \textbf{label}\ s_0 :\ \ldots \\ \qquad \textbf{for}\ x \cdots : s_2 \\ \xrightarrow{\textbf{compute-at}(f,\langle g,x\rangle)} \\ \textbf{label}\ g :\ \ldots \\ \quad \textbf{label}\ s_0 :\ \ldots \\ \qquad \textbf{for}\ x \cdots : (\textbf{label}\ f : s_1); s_2 \end{array} \qquad \begin{array}{l} \textbf{allocate}\ f(\ldots)\ \ldots \\ \textbf{label}\ g :\ \ldots \\ \quad \textbf{label}\ s_0 :\ \ldots \\ \qquad \textbf{for}\ x \cdots : s_2 \\ \xrightarrow{\textbf{store-at}(f,\langle g,x\rangle)} \\ \textbf{label}\ g :\ \ldots \\ \quad \textbf{label}\ s_0 :\ \ldots \\ \qquad \textbf{for}\ x \cdots : (\textbf{allocate}\ f(\ldots); s_2) \end{array}$$

$$\begin{array}{l} \qquad \textbf{label}\ f : s \\ \quad \xrightarrow{\textbf{bound}(f,x,e^{\min},e^{\text{len}})} \\ \textbf{label}\ f : \\ \quad \textbf{assert}\ \ ?^{\mathbf{cpu}}f_x = [e^{\min}, e^{\text{len}}]\ ; s \end{array} \qquad \begin{array}{l} \qquad \textbf{label}\ f : s \\ \quad \xrightarrow{\textbf{bound-extent}(f,x,e^{\text{len}})} \\ \textbf{label}\ f : \\ \quad \textbf{assert}\ \ (?^{\mathbf{cpu}}f_x)^{\text{len}} = e^{\text{len}}\ ; s \end{array} \qquad \begin{array}{l} \qquad \textbf{label}\ f : s \\ \quad \xrightarrow{\textbf{align-bounds}(f,x,e^m,e^r)} \\ \textbf{label}\ f : \\ \quad \textbf{assert}\ \ (?^{\mathbf{cpu}}f_x)^{\min}\ \%\ e^m = e^r\ ; \\ \quad \textbf{assert}\ \ (?^{\mathbf{cpu}}f_x)^{\text{len}}\ \%\ e^m = 0\ ; s \end{array}$$

$$\begin{array}{c} \textbf{for}\ x\ \textbf{in}\ \ (e^{\min}, e^{\text{len}})\ \textbf{do}\ s \\ \xrightarrow{\textbf{split}(\ell,x_o,x_i,e^{\text{fac}},\varphi)} \\ \textbf{for}\ x_o\ \textbf{in}\ \ (0, \lceil e^{\text{len}}/e^{\text{fac}} \rceil)\ \textbf{do} \\ \quad \textbf{for}\ x_i\ \textbf{in}\ \ (0, e^{\text{fac}})\ \textbf{do} \\ \qquad \textbf{push-down}(\varphi, s) \end{array}$$

$$\textit{where } \varphi \textit{ is} \quad \varphi_{\text{Guard}}(s) = \begin{cases} \textbf{let}\ x = e^{\min} + x_i + e^{\text{fac}} \cdot x_o\ \textbf{in} \\ \textbf{if}\ x < e^{\min} + e^{\text{len}}\ \textbf{do}\ s \end{cases}$$

$$\textit{or} \quad \varphi_{\text{Shift}}(s) = \begin{cases} \textbf{let}\ x = e^{\min} + x_i \\ \qquad + \min(e^{\text{fac}} \cdot x_o, \\ \qquad\quad \max(0, e^{\text{len}} - e^{\text{fac}})) \\ \textbf{in}\ s \end{cases}$$

$$\textit{or} \quad \varphi_{\text{Round}}(s) = \textbf{let}\ x = e^{\min} + x_i + e^{\text{fac}} \cdot x_o\ \textbf{in}\ s$$

$$\begin{array}{l} \textbf{for}\ x_1\ \textbf{in}\ \ (e_1^{\min}, e_1^{\text{len}})\ \textbf{do} \\ \quad \textbf{for}\ x_2\ \textbf{in}\ \ (e_2^{\min}, e_2^{\text{len}})\ \textbf{do}\ s \\ \qquad \xrightarrow{\textbf{fuse}(\ell,y)} \\ \quad \textbf{for}\ y\ \textbf{in}\ \ (0, e_1^{\text{len}} e_2^{\text{len}})\ \textbf{do} \\ \qquad \textbf{push-down}(\varphi, s) \end{array} \qquad \textit{where } \varphi(s) = \begin{cases} \textbf{let}\ x_1 = e_1^{\min} + \lfloor y/e_2^{\text{len}} \rfloor\ \textbf{in} \\ \textbf{let}\ x_2 = e_2^{\min} + y \% e_2^{\text{len}}\ \textbf{in}\ s \end{cases}$$

$$\boxed{\begin{array}{r} \textbf{push-down}(\varphi, \textbf{for}\ x\ \textbf{in}\ I\ \textbf{do}\ s) = \textbf{for}\ x\ \textbf{in}\ I\ \textbf{do}\ \textbf{push-down}(\varphi, s) \\ \textbf{push-down}(\varphi, \textbf{let}\ x = e\ \textbf{in}\ s) = \textbf{let}\ x = e\ \textbf{in}\ \textbf{push-down}(\varphi, s) \\ \textbf{push-down}(\varphi, s) = \varphi(s) \end{array}}$$

Figure 2.12: Scheduling directives over the IR

the additional flexibility granted by the additional loop iteration information to derive tighter bounds. This directive therefore controls *how much* of a func to compute before computing part of its consumers.

2.8.4 Storage Phase

Each func is tied to a particular piece of memory when it is computed. Halide offers some control over how much memory a func occupies during the run of a pipeline. The *store-at* directive (analogous to compute-at above) moves the allocation statement to just inside any loop such that the allocation still dominates all accesses of the func it allocates.

Bounds inference is then free to choose a more precise size for the allocation based on the code that follows, and the particular values of the variables of the loops that enclose it.

2.8.5 Bounds Phase

Additional domain knowledge might allow a user to derive superior bounds functions than those inferred. Halide provides directives to give hints to the bounds engine just before querying it.

The first two directives, *bound* and *bound-extent*, assert equality of bounds holes to provided startup expressions. The third directive, *align-bounds*, adds assertions that constrain the divisibility and position of the window. The minimum is constrained to have a particular remainder modulo a factor which is declared to divide the extent. These assertions affect the bounds inference query such that the inferred computation window will expand to meet these requirements. Recall that these assertions are allowed to fail without violating *confluence* (definition 2.3).

2.8.6 Practical directives

Halide provides many more scheduling directives that are out of scope for this work. It has directives for assigning loops to coprocessors like GPUs and DSPs, and directives for prefetching and memoization. It has two additional traversal orders that apply only to constant-extent loops after bounds inference has completed and are semantically uninteresting: **vectorize**, which asks Halide to vectorize the loop, and **unroll**, which simply unrolls the loop. Some of the most esoteric directives may require more substantial adjustments to these semantics.

2.9 Practical impact

These formalization efforts have influenced Halide's design, and we have found and fixed bugs where actual and expected behavior differed in significant ways.

Negative rdom extents. While formalizing the behavior of reduction domains (§2.4.3), we discovered that the practical system had not defined the behavior of loops with *negative* extents [69]. Test cases designed to probe the behavior suggested that Halide treated such loops as no-ops; however, there could be instances wherein a negative extent is treated as unsigned, which would silently wrap to a very large positive integer. While unsigned underflow is a well-known problem, Halide has the additional obligation of making sure that no scheduling transforms accidentally introduce this behavior even if it's absent in the original code. We worked with the developers to determine that this situation should be treated as an error that can be checked at program startup (recall definition 2.9), as formalized here.

Impure identity functions. The practical system has several APIs for computing the results of a pipeline. One such API intended to match the interface formalized here (§2.4, §2.5): the user supplies the desired *compute bounds* (see §2.7) and receives a buffer containing at least the requested values.

For efficiency, another API allows a user to supply their own output buffer, rather than delegating the allocation to Halide. In this case, the pipeline checks at startup that the supplied buffer is at least as large as the buffer it *would have* allocated. However, when the simple API was implemented in terms of this advanced API, it incorrectly assumed that the *compute bounds* and *allocation bounds* would be equal. This led to vexing errors on pipelines whose outputs were scheduled to overcompute [65].

This confusion had a surprising consequence: adding an unscheduled *identity* func to the end of the pipeline would compile to a *copy* of the former output, and which would have equal compute and allocation bounds. So, from the perspective of the user, identity functions were *impure* since they had side effects due to bounds inference. After reaching clarity on these issues through our formalism, we worked with the Halide authors to fix this behavior. The latest release correctly returns the full, possibly overallocated, buffer.

Arithmetic error semantics. As discussed in §2.7.1, values used in control flow or indexing must be *well-defined* regardless of the schedule. Data-dependent accesses in rdom conditions and update locations might necessitate computing points not required by the default schedule, especially when over-computing strategies are employed. Similarly, computation outside the compute bounds must be side-effect free, even when processing uninitialized values. One consequence of this is that integer division and modulo must be made into *total* functions, similar to IEEE 754 arithmetic.

We constructed test cases for the practical system that crashed due to integer division by zero happening outside of the compute bounds [67]. We worked with the Halide authors to *define* these operations to return zero and implemented the new behavior with runtime checks. The compiler leverages its existing bounds analyses to eliminate these checks when it can, for instance when dividing by a non-zero constant.

One might wonder why the convention $x \bmod 0 \equiv 0$ was chosen in favor of the more typical $x \bmod 0 \equiv x$. Both conventions were tried and the former produced tighter bounds equations in practice; in short, it is better to bound (x mod y) by y than by x, which is typically much wider.

This change also impacted concurrent work on verifying Halide's term-rewriting expression simplifier. As Newcomb et al. [116] report, these new semantics invalidated dozens of existing rewrite rules and required many new proofs of correctness for valid rules.

Compute bounds for indexing accesses. Another consequence of the rules in §2.7.1 is that accesses that occur inside indexing expressions must have well-defined values, which means that the points must be in the compute bounds. However, the practical system did not implement this rule; it instead relied on an unsound analysis of the bounds of a func's value to compute the bounds in the indexing expression and did not widen the compute bounds to fit the accessed point. We were able to construct a real crash based on this insight in [63] and provided a patch to the compiler.

Race conditions in rdom predicates It is unsafe to parallelize a loop that contains an RDom whose predicate depends on values written by that loop. Race conditions on the values read by the predicate can lead to non-deterministic behavior. We discovered that the compiler was missing these checks. We constructed a real instance of non-determinism based on this insight [68] and provided a patch to the compiler[64].

Compute-with directive. *Compute-with* was a scheduling directive intended to interleave the computation of two or more independent funcs by fusing their outermost loops together. This could benefit performance by reducing memory traffic if the two funcs shared many reads from a common producer. However, the prototype implementation did not consider dependencies between the stages of a single func (§2.4.3), nor did it consider *specializations* (§2.8.1). We discovered cases where compute-with could move the pure stage of a func *after* one of its update stages, resulting in crashes and mangled outputs [66].

We worked with the Halide authors to define the feature, but due to little widespread use (perhaps owing to these bugs, in part) and the highly complex implementation, the feature was deprecated instead. We look forward to designing a sound replacement in future work.

Future work We believe this work provides a foundation to study the new class of languages with user-controlled scheduling. One major question is how they could incorporate abstraction and module systems. Another is whether alternative bounds inference algorithms, based on our program synthesis formulation, could be useful in practice and in other settings.

2.10 Proofs of Theorems and Lemmas

Lemma 2.10 (Loop phase naming)**.** *The loop phase preserves loop names as described in lemma 2.4.*

Proof. All loop phase directives replace loops of a given func, specialization, and stage with similarly localized loops, albeit with different loop variables. Since conflicts between loop variable names are prohibited, loops remain uniquely named. □

Definition 2.17 (Narrowing program)**.** Let $P_1, P_2 \in \mathbf{BI}(T)$ where T is a program scheduled from some algorithm. We say that $P_1 \leq P_2$ iff for all inputs z, every execution of any allocation or loop in $P_1(z)$ has a corresponding execution in $P_2(z)$ and the bounds I_1 for $P_1(z)$ are contained in the bounds I_2 for $P_2(z)$. (Here "corresponding" means that the two statements are at the same lexical site, executing with identical environments Σ.)

Lemma 2.11 (Narrowing executions match)**.** *Let $P_1 \leq P_2$ as above. Then after the execution of corresponding loops for some func f on intervals I_1 and I_2, the contents of the buffer for f agree on I_1.*

Proof of lemma 2.11. Let $\mathbf{SI}(P, z)$ denote the set of assignment executions in $P(z)$. By definition, $\mathbf{SI}(P_1, z) \subseteq \mathbf{SI}(P_2, z)$. Let $\mathbf{SI}(P_2, z)|_{I_1}$ be the set of assignment statement executions in $P_2(z)$ which write which write to a point of f in I_1. Then $\mathbf{SI}(P_1, z) \supseteq \mathbf{SI}(P_2, z)|_{I_1}$. Thus there are no writes to values in I_1 in P_2 that do not also occur in P_1. Furthermore, all of these computed values depend only on the values computed in both programs. This follows from the assignment rule of $\mathcal{B}$. □

For the next three proofs, note that if an expression appearing in an access is unbounded then there is no possible bounds query result. So without loss of generality, we may assume that some satisfying bounds *actually do exist*, since confluence is vacuous otherwise.

Proof of lemma 2.6. First recall that by lemma 2.5, every access to a func f is dominated by the **allocate** statement for f. The rule for accesses in $\mathcal{B}$ (see fig. 2.10a) explicitly requires that every point read or written is contained in the allocation bounds (**Mem**) of the associated func. Thus P' will always satisfy the InBounds condition. □

Proof of lemma 2.7. This proof proceeds by induction over prefixes of the syntactic structure of the algorithm $P \in \mathbf{Alg}$ and corresponding prefixes of the initially lowered program $P' \in \mathbf{BI}(\mathcal{L}(P))$.

Recall from the definition of the bounds extraction function $\mathcal{B}$ (§2.7.1) that each of f's stages' loop bounds cover the compute bounds. lemma 2.5 ensures that the func is realized before it is read.

Thus, as a base case, if f consists of only a pure stage, we are done because the static lack of self-reference and reduction dimensions makes each assignment statement in that stage completely independent of every other (this is ensured by definition 2.7). The assignment exactly matches the algorithm's expressions and only reads from compute bounds by assumption.

Inductively, if f has n stages the claim holds, we argue that adding another stage s to f preserves the claim. Bounds extraction ensures that every access in s is included in the allocation bounds and assignments writing to a point in the compute bounds read only within the compute bounds of other funcs and f. By the induction hypothesis on stages, the values in the compute bounds of the buffer for f (and all other funcs) are correct just before s runs.

Recall the structure of the loop nest for s. The outermost for loops correspond to pure dimensions and range over bounds holes, which are constrained to cover the compute bounds of f. The innermost for loops implement any reduction domain in the stage and have bounds supplied by the algorithm. If the stage is guarded by a condition, it is included within the innermost for loop. Finally, a single assignment statement corresponding to the update rule for the stage is the innermost statement.

We need to show that the code produced by lowering for s will compute confluent values for f. Consider a point p in the compute bounds of f after the stage runs. The stage s separates pure dimensions from reduction dimensions of p. Let x_p be the assignment to pure dimensions induced by p. Consider the definition of s in the algorithm. It consists of a series of simple updates after unrolling the rdom all of which share values x_p in common. Now, observe the iteration of the pure loops of s in the target language which coincide with x_p. This iteration is guaranteed to occur by the covering of the compute bounds by the loop bounds. The content of this iteration is a sequence of assignments corresponding to the unrolled rdom (as in the big step semantics in fig. 2.6). Lastly, any other $x'_p \neq x_p$ touches no memory in common with this iteration because of syntactic separation (definition 2.7). □

Lemma 2.12 (Lowering is sound)**.** *$P' \in \mathbf{BI}(\mathcal{L}(P))$ is confluent with P for all $P \in \mathbf{Alg}$.*

Proof of lemma 2.12. Let z be any input. If $P(z)$ contains an error, then we are done. Since lowering does not create any assertion statements, failing one is not possible. All lowered programs trivially respect dominance. Finally the argument in lemma 2.7 applies inductively over the lowered code. □

Proof of lemma 2.9. If f has no specializations, then its only compute statement is considered the default specialization. Valid schedules have at most one specialization directive per func, so there is one block of if-then statements for each func, each containing a compute statement for f. Since the conditions of those branches are startup expressions, the input z determines which one will be taken every time the block is encountered. Proofs that each subsequent phase of scheduling preserves this invariant appear in §2.10. □

Lemma 2.13 (Specialization is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a scheduling directive in this phase, then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

Proof of lemma 2.13. Let z be a valid input for P, $P'_i \in \mathbf{BI}(T_i)$, and $P'_{i+1} \in \mathbf{BI}(T_{i+1})$. If $P(z)$ contains an error, we are done; and no assertions are present until the bounds phase; so we may assume $P \simeq_z P'_i$. By assumption, s is a specialization of some func f. Now by lemma 2.9, exactly one branch of f is taken in any execution $P'_{i+1}(z)$. This preserves producer domination as required by lemma 2.5. □

Lemma 2.14 (Loop phase is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a scheduling directive in this phase, then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

Proof of lemma 2.14. Lemma 2.13 ensures that T_i is confluent as long as we have not yet reached this phase. So we may inductively assume confluence before issuing any such s. As before, s does not introduce assertions, so we need only assess output equivalence. Each directive s operates locally on loops in a single stage, so therefore we need only show that the transformation of this stage is observationally equivalent; i.e. the state of the buffers before and after the stage is the same within the compute bounds.

Suppose $s = \mathbf{split}(\langle \dots, v \rangle, v_o, v_i, e^{\text{fac}})$. We may check that the values v ranges over are unchanged; this means that $\mathcal{B}(T_i)$ is logically equivalent to $\mathcal{B}(T_{i+1})$. Since the scopes of holes have also not changed ($K(T_i) = K(T_{i+1})$), the sets of bounds query results are identical. Thus there is a bijection between programs $P'_i \in \mathbf{BI}(T_i)$ and $P'_{i+1} \in \mathbf{BI}(T_{i+1})$, s.t. corresponding holes have been filled with identical expressions. These two programs execute the same statement instances in the same order; therefore the effect on buffers is identical.

The argument for the **fuse** directive proceeds analogously, as do the arguments for the overcomputation and inward-shifting split strategies that apply only to single-stage (pure) funcs, which are adjusted for the expanding compute bounds and identically recomputed points, respectively.

For **swap** and **traverse**, consider the loop nest for the stage of f. Analogously to the proof of lemma 2.7, syntactic separation ensures that distinct iterations of pure loops access f at disjoint sets of points. Since **split** and **fuse** preserve the sets and order of accesses as they introduce new loops, two writes agree on their pure dimensions iff the pure loop variable values agree.

But this means that two assignments touch the same memory of f only if the values of the pure loops are the same. Since **traverse** parallelizes over a single pure loop, no two tasks touch the same memory. And **swap** interchanges two pure loops, so the writes which do touch memory in common are not interchanged with respect to one another. $\square$

Lemma 2.15 (Compute phase is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a compute-at directive moving the func f to some location ℓ_g. Then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

Proof of lemma 2.15. Let $P'_i \in \mathbf{BI}(S_i)$ and let $\{P''_i\}$ be the set of programs resulting from applying s to P'_i. Let $P'_{i+1} \in \mathbf{BI}(T_{i+1})$. First observe that any P''_i computes the same values as P'_i because the **compute** statement for f computes *all* of the points ever needed in P''_i and dominates all of its consumer funcs. This also implies that $P''_i \in \mathbf{BI}(T_{i+1})$.

Now suppose P'_{i+1} is not one of the P''_i. Then for each z, if $P(z)$ does not contain an error, then there is some $P''_i \geq P'_{i+1}$ (see definition 2.17). By lemma 2.11, P'_{i+1} computes the exact same values of f on the narrower range, which bounds inference guarantees is sufficient for confluence. $\square$

Lemma 2.16 (Storage phase is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a store-at directive, then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

Proof of lemma 2.16. Lemma 2.6 required only dominance of the **allocate** statement over all accesses to the associated func for correctness. This is preserved by definition. □

Lemma 2.17 (Bounds phase is sound)**.** *Let $P \in \mathbf{Alg}$ be a valid algorithm and let $T_i \in \mathbf{Tgt}^?$ be the result of lowering P and applying scheduling directives up through this phase. Let s be a scheduling directive in this phase, then $T_{i+1} = \mathcal{S}(s, T_i)$ is confluent with P.*

Proof of lemma 2.17. The directives in this phase only mutate programs by adding assertions to them, the only side effect of which is to transition to an error state. Thus in any non-erroring execution of any $P_i \in \mathbf{BI}(S_i)$, there is some $P_i \in \mathbf{BI}(T_i)$ whose behavior exactly matches P_{i+1}. □

Chapter 3

Exocompilation for Productive Programming of Hardware Accelerators

This chapter is based on the work in Ikarashi, Bernstein, Reinking, Genc, and Ragan-Kelley [80], which was published at PLDI '22.

3.1 Introduction

Modern computers are increasingly comprised of accelerators. From neural and cryptography engines, to image signal processors, to GPUs, a state-of-the-art system-on-chip (SoC) today includes dozens of different specialized accelerators. Even within their main CPUs, performance improvement increasingly comes via new instructions performed by specialized functional units. This specialized hardware is orders of magnitude more efficient than software running on general-purpose hardware, but most applications are only able to realize this performance and efficiency insofar as key low-level libraries of high-performance kernels (e.g., BLAS, cuDNN, MKL, etc.) are optimized to exploit the hardware.

While the role played by high-performance kernel libraries is increasingly critical, there is little programming language support for the performance engineers who write them. Progress continues to be made after decades of effort on fully-automatic compiler optimization, but state-of-the-art kernels—from linear algebra, to deep learning, to signal processing and cryptography—are still predominantly written by hand, directly in low-level C and hardware-specific intrinsics or assembly, or with lightweight metaprogramming (e.g., macros or C++ templates) of such low-level code. As a result, developing and tuning these libraries is enormously labor intensive, limiting the range of accelerated routines and creating barriers to deploying new or improved accelerators.

Developing accelerated high-performance libraries is a unique software engineering task, with several unusual characteristics. First, in contrast to conventional programs on general-purpose processors, the hardware-software interfaces to accelerators are both complex—including specialized memories, exposed configuration state, and complex operations—and highly diverse, with *different* complexities unique to each accelerator. Second, the rates of change at different levels in the stack—from applications to hardware ISA—are inverted: accelerator architectures change *more* rapidly than the essential functions which run on them (e.g., mobile phone SoCs are rebuilt every year, with major revisions to nearly every accelerator block, while the BLAS standard changes much more slowly), and the implementation of these functions to most efficiently use the hardware is iterated more quickly, still. This is especially acute during accelerator development, where target application workloads are often fixed, while both the hardware architecture and kernels mapping to it are iteratively co-designed to maximize performance and efficiency.

In this paper, we propose *exocompilation* as a new approach to programming language and compiler support for developing hardware-accelerated high-performance libraries. The principle of exocompilation is to externalize as much accelerator-specific code-generation logic and optimization policy from the compiler as possible, instead exposing them at the user level to high-performance library writers. Specifically, we externalize accelerator specification to user-level libraries, and we build on the idea of user scheduling, popularized by languages like Halide and TVM [22, 130], to externalize hardware mapping and optimization decisions.

We develop a new language and compiler called Exo based on this principle of exocompilation. Exo allows custom hardware instructions to be user-defined and abstracted as procedures. It also allows specialized memories and accelerator configuration state to be defined in user code, without modifying the core compiler. User scheduling enables a rich space of optimization and hardware mapping choices to be directly explored by the performance engineer, rather than requiring an automated optimizer to always make perfect decisions.

In contrast to optimization by manually rewriting low-level code, scheduling transformations are concise and safe. They elide many details like array and loop re-indexing (which can be automatically inferred), while guaranteeing both functional equivalence and memory safety. Different schedules best optimize the same library function for different hardware, or even for different parameter values, and specialized versions for each case can be generated from a single source algorithm. Arbitrary program fragments can be replaced during scheduling with equivalent user-defined accelerator instructions, or specialized subroutines, using a *unification* procedure that automates the transformation of essential arguments and array indexing.

Finally, in contrast to languages like Halide and TVM, Exo implements user scheduling via composable rewrite rules. This allows the scheduling language itself to be easily extended, since each operator defines an independent rewrite, rather than interacting with all others in a monolithic lowering process.

We explore what is required of safety analyses for such a language, and define a set of *effect analyses* which support guarantees of program equivalence and memory safety after scheduling (§3.5). We make the simplifying assumption of affine loops and array indexing, which has been shown to be sufficient for many kernels of interest in high-performance libraries [43]. Nonetheless, accelerator configuration introduces global mutable state which breaks the classic "static control program" assumption, and requires introducing approximation into the analyses. Our analyses are then defined in a ternary logic, which distinguishes effects which *definitely* occur (necessary for, e.g., eliminating redundant setting of configuration state) from those which *maybe* occur (relevant for reasoning about the statement reorderings which emerge from many loop transformations).

Finally, we perform a series of case studies applying Exo to optimizing high-performance kernels for specialized hardware. We develop user-level backends for the Berkeley Gemmini neural network accelerator [51] (a software-controlled systolic array similar to many TPU-like architectures) and x86-64 with AVX-512. For each target, we focus on optimizing matrix multiply and convolutional neural network layers—among the most highly-optimized kernels in common libraries. Using Exo, we were able to easily develop implementations competitive with state-of-the-art libraries in a few days and a few dozen lines of code.

3.2 Example

Today's large machine learning models (and scientific computing) rely on highly tuned matrix-matrix multiplication kernels (aka. GEMM). In order to introduce Exo, we will show how to write and optimize such GEMM kernels, targeting one to an accelerator ISA designed to resemble machine learning accelerators. These accelerators all focus on the efficient execution of small (e.g. 16×16), dense matrix-matrix multiplication instructions.

Optimizing these kernels is primarily an exercise in orchestrating data movement, and only secondarily a matter of selecting compute instructions, such as the actual matrix multiplication primitive. Therefore, we need to explicitly schedule loads and stores from custom, explicitly managed accelerator memories. Lastly, much of the behavior of hardware accelerators is controlled by infrequently changing *configuration state*. Instructions to configure such state usually flush the accelerator pipeline.

To model a particular hardware accelerator, users must define custom memories, instructions and configuration state. This work is done once per accelerator, written as a *hardware library*. Throughout the example, we will indicate whether each piece of code lives in the application (GEMM) or can be abstracted out into a reusable description of the hardware.

3.2.1 Exo Procedures, Compilation, and Scheduling

Consider matrix-matrix multiplication, written in Exo:

in `app.py`

```
@proc
def gemm(A: R[128, 128] @ DRAM, B: ..., C: ...):
  for i in seq(0, 128):
    for j in seq(0, 128):
      for k in seq(0, 128):
```

```
        C[i, j] += A[i, k] * B[k, j]
```

Exo is embedded in Python, and the function decorator `@proc` indicates the beginning of an Exo function. Function arguments are given by the syntax

$$\langle name \rangle \texttt{:} \langle type \rangle \texttt{[} \langle size \rangle \texttt{]} \ \texttt{@} \ \langle memory \rangle$$

`R` is an abstract type for all numeric data types, which can be specialized to specific precision types such as `f32` and `i8` via scheduling operations. For simplicity, the $\langle size \rangle$ in this example is constant, but usually refers dependently to other function arguments. The `@` symbol is a *memory specification*; `@DRAM` means that the buffer is expected to be in DRAM. Finally, `for i in seq(0, 128)` is a *sequential* for loop that ranges from `0` to `127` (inclusive).

Exo compiles to `C` source code in the expected way:

`app.c` (generated)

```
void gemm(float *A, float *B, float *C) {
  for (int i=0; i<128; i++) {
    for (int j=0; i<128; i++) {
      for (int k=0; i<128; i++) {
        C[128*i + j] += A[128*i + k] * B[128*k + j];
} } } }
```

In order to target our accelerator, we need to expose a 16×16 matrix-multiplication as the inner loop nest. We do this by using *scheduling operations* to rewrite the procedure. In particular, we `split(i,16,io,ii)` (sim. for `j`, `k`) and then `reorder()` the loops (see §3.3.3) to produce the following tiled matrix multiplication:

in `app.py`

```
def gemm(A: R[128, 128] @ DRAM, B: ..., C: ...):
  for io in seq(0, 8):
    for jo in seq(0, 8):
      for ko in seq(0, 8):
        for ii in seq(0, 16):
          for ji in seq(0, 16):
            for ki in seq(0, 16):
              C[16*io+ii, 16*jo+ji] += A[..] * B[..]
```

3.2.2 Memories

Many accelerators—including ours in this example—have explicitly-managed memories. Performance critically depends on how data movement to and from these memories is interleaved with other computation. Therefore Exo puts scheduling of data movement in the hands of the programmer. The first step in doing this, is to define *custom memories* on a per-accelerator basis. For example, in hw_lib.py

```
class ACCUMULATOR(Memory):
    def alloc(...):
        return f"{prim_type} {name} = hw_malloc({sz});"
    def free(...):
        return f"hw_free({name});"
    def read(...):  # also write, reduce
        raise MemGenError('memory is not addressable')
```

If a buffer is annotated with ACCUMULATOR instead of DRAM, then these alloc and free *macros* will determine the C code that is generated when that buffer is allocated or freed. (see §3.3) Furthermore, note that the ACCUMULATOR memory explicitly disables code generation for reading, writing and accumulating into individual locations, preventing direct access from C. Instead, we will only allow custom instructions (see below) to access this custom memory.

Supposing we have written custom ACCUMULATOR and SCRATCHPAD memories, we use stage_mem scheduling operations to stage C, A, and B into these memories:

in app.py

```
def gemm(...):
    res: R[...] @ ACCUMULATOR
    a  : R[...] @ SCRATCHPAD
    b  : R[...] @ SCRATCHPAD
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Load C to res
            for ko in seq(0, 8):
                # Load A to a
                for ii in seq(0, 16):
                    for ki in seq(0, 16):
                        a[...] = A[...]
                ... # Load B to b
                # Matmul of a and b
                for ii in seq(0, 16):
                    for ji in seq(0, 16):
                        for ki in seq(0, 16):
                            res[..]+=a[..]*b[..]
            ... # Store res to C
```

3.2.3 Instructions

We can clearly see opportunities in the above code to map loops to semantically equivalent accelerator instructions. However, to do this safely and soundly, the compiler needs definitions of our accelerator instructions in terms of Exo's semantics. The key idea of exocompilation is to provide users with a framework for defining these instructions in libraries, without modifying the compiler itself. Below, we show an example of such a definition for the scratchpad load.

in hw_lib.py

```
@instr("config_ld({src}.strides[0]);\n"
       "mvin({src}.data, {dst}.data, {m}, {n});")
def ld_data(n: size, m: size,
            src: [R][n, m] @ DRAM,
            dst: [R][n, 16] @ SCRATCHPAD):
    assert m <= 16
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

Notice that this function has been annotated with `@instr` rather than `@proc`. This indicates that the declaration *asserts* equivalence between the Exo code in the body and the C code template (i.e. macro) in the annotation. The resulting `ld_data` function may be scheduled and called like any other function, but Exo's C code generator will instead emit the C code "`config_ld({src}.strides...)`", with argument placeholders `{src}` and `{dst}` substituted appropriately.

Exo provides a `replace()` scheduling directive (§3.3.4) for matching code in one procedure with the body of another procedure (including an `@instr` like `ld_data`), then *replacing* the matched code with an appropriate procedure call.

3.2.4 Configuration State

We could issue this directive now to schedule the accelerator instructions, however, the C code has fused the expensive `config_ld` instruction to the `mvin` instruction we are really interested in scheduling. Since the stride does not actually change during the kernel, this will cause the accelerator pipeline to repeatedly flush and stall. We must somehow schedule the configuration instruction independently of the actual load.

Therefore, we need a way to define hardware state. The following code models the stride configuration state in Exo.

in hw_lib.py

```
@config
class ConfigLoad:
    src_stride : stride

@instr("config_ld({s});")
def config_ld_def(s : stride):
    ConfigLoad.src_stride = s
```

Here, `ConfigLoad` defines a global struct of configuration variables, here containing a single `src_stride` field that models the state of the stride hardware parameter. We also write an instruction definition, `config_ld_def`, that updates the `src_stride` field. Now we can write a new instruction for the 16×16 load without the `config_ld` setup:

in `hw_lib.py`

```
@instr("mvin({src}.data, {dst}.data, {m}, {n});")
def real_ld_data(...):
    assert ConfigLoad.src_stride ==
           stride(src, 0)
    # same as ld_data
```

Using scheduling instructions, we will rewrite the body of `ld_data` into a call to `config_ld_def()`, followed by a call to `real_ld_data()`. First, we use the `configwrite_at()` scheduling operation to rewrite `ld_data` into the following:

in `hw_lib.py`

```
def ld_data(...):
    assert m <= 16
    ConfigLoad.src_stride = stride(src, 0)
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

Unlike previous scheduling operations, `configwrite_at()` only partially preserves procedure equivalence—the new `ld_data()` is only equivalent *up to* the configuration state `ConfigLoad.src_stride`. In general, Exo needs to reason about this kind of program equivalence modulo configuration state (see definition 3.1 and §3.6.2).

Since the statement `ConfigLoad.src_stride` = ... is equivalent to the body of `config_ld_def`, and the statement `for i in seq(...):` ... is equivalent to the body of `real_ld_data`, we can now `replace()` the body of `ld_data` with the two calls, as desired:

in `hw_lib.py`

```
def ld_data(...):
    assert m <= 16
    config_ld_def(stride(src, 0))
    real_ld_data(n, m, src, dst)
```

By following this same procedure, we can create instruction abstractions for our 16x16 matmul and store instructions. At last, we can replace the code in `gemm` with calls to `ld_data` and inline its definition.

in `app.py`

```
def gemm(...):
    res: R[...] ...
    for io in seq(0, 8):
        for jo in seq(0, 8):
```

```
            ... # Loading C to res
            for ko in seq(0, 8):
                config_ld_def(stride(A, 0))
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```

We will hoist the call to `config_ld_def` using scheduling operations `reorder_stmts()`, `fission_after()`, as well as `remove_loop()`. Doing so will require Exo's program analysis to both reason about when different statements *commute* (can be reordered) as well as when they are *idempotent* (allowing the loop to be removed). To further complicate matters, the presence of global, mutable *configuration* state means that fully precise analyses are undecidable, and thus impossible in Exo. By using a ternary logic (§3.5), Exo can distinguish between memory locations that are *definitely* written to (a necessary condition for idempotency) and locations that are *maybe* written to (the relevant condition for commutativity).

in `app.py`

```
def gemm(...):
    config_ld(stride(A, 0))
    res: R[...] ...
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Loading C to res
            for ko in seq(0, 8):
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```

All of the above code transformations are achievable using the scheduling primitives discussed in §3.3. Full definitions of the memory, configuration, and load instructions for the Gemmini accelerator can be found in §3.15.

3.3 The Exo Language and System

The Exo system consists of an imperative programming language (§3.3.1), means of defining hardware targets via libraries (§3.3.2), and a rewrite-based scheduling system (§3.3.3 and 3.3.4). Figure 3.1 shows the Exo system from the standpoint of a particular program being compiled. In this section, we explain each part of this process.

3.3.1 The Exo Language

Exo is a familiar imperative language in the mold of the static control program model [43]. It supports for-loops, if-conditions, arrays and procedures, but not while-loops or recursion. A BNF grammar for its formal core is defined later (fig. 3.2). In addition to that grammar, the full language supports `stride` values and expressions, as well as memory annotations, both of which were shown in the example (§3.2).

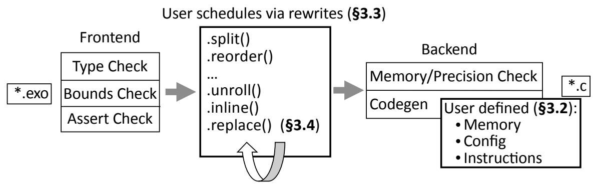


Figure 3.1: Exo system overview

Six relatively standard (but not universally adopted) features of Exo are worth discussing further: (1) control/data separation, (2) mutable global control state, (3) dependently typed arrays [165], (4) array windowing/slicing, (5) explicit += reduction primitives, and (6) static assertions.

1. Exo is built around a distinction between control and data values. Control values (types `int`, `bool`, `size`, etc.) are constrained so that they may be analyzed more precisely. Arithmetic on integer control values must be quasi-affine, meaning that values can only be multiplied, divided, or modulo-ed by an integer literal. Expressions inside loop bounds and branches must be control values. Meanwhile, data values (types `R`, `f32`, `i8`, etc.) are floating-point or fixed-point numbers stored in scalars or arrays. There are no restrictions on allowed computations between data values.

2. Configuration state (§3.2) is introduced via structs of variables using `@config` and modeled formally as global variables (§3.4). Unlike the other sources of control values, configuration state is mutable. Consistent with the idea of static control programs, Exo currently prohibits any dependence of control values on data-values, regardless of whether those control variables are local or global.

3. Dependently typed arrays allow sizes to be specified by control value expressions of strictly positive value. Exo then performs static bounds checks, guaranteeing memory safety without incurring any of the costs of dynamic bounds checks. This is made possible by the control/data separation idea.

4. Arrays in Exo are further extended with support for windowing (aka. slicing) via the `x[lo:hi]` syntax. Creating a window does not copy data; instead, reading from and writing to locations in a window accesses the underlying buffer (e.g. if `y = x[3:8]` then `y[2] == x[3+2]`). In particular, note that windows may be lower-dimensional than their underlying buffers by slicing some indices, while point-accessing others. For instance, `x[0:n,j]` creates a 1-dimensional window on column `j` of matrix `x`.

5. In addition to primitive reading and writing, reduction via the += syntax is supported as a special commutative and associative operation from the point of view of program analysis.

6. Finally, we allow static assertions about control values to be made at the beginning of procedures. These assertions act as *pre-conditions* and not as dynamic tests. Program

analysis within a procedure may assume its asserted pre-conditions, whereas a calling procedure is only valid if it ensures that the callee's pre-conditions are true.

Backend Checks: Precision and Memory

Type-checking, bounds-checking, and assertion checking are all front-end checks on Exo code. By contrast, consistency of data-variable precision types as well as consistency of memory annotations are performed as back-end checks immediately prior to code generation. Exo requires all data-expressions to have consistent precision, (e.g. multiplying an `f32` and `i8` is forbidden) but inserts type-casts as necessary just before writing or reducing data values.

Code Generation

Exo is designed to generate human-readable C-code that is more or less a syntactic translation of the corresponding Exo code. This enables the programmer to more easily integrate Exo with existing tools and workflows. There are a few non-obvious details with this translation that merit discussion. First, all data values (including scalars, buffers, and windows) are passed by pointer rather than by value. This is necessary even in the case of scalars to allow "returning" modified scalar values to a caller. Second, windows are compiled to structs containing both the data pointer and stride values, since the static size of a window is insufficient to compute a linear address into the underlying buffer. Lastly, we translate static assertions into compiler-specific optimization hints to help improve downstream analyses and optimizations.

3.3.2 Hardware Targets as Libraries

To add support for a new hardware accelerator to Exo, programmers write a library, rather than a compiler backend. These libraries use three key features of the Exo language: (1) memories, (2) instructions, and (3) configuration state. Using these features, an Exo programmer can hand-write code to target a given accelerator, or use scheduling to rewrite a simple program into one targeting a given accelerator (§3.3.3).

Defining hardware in libraries has two advantages over defining hardware in compiler backends (as Halide, TVM, LLVM and most compilers do). First, hardware vendors do not need to maintain compiler forks in order to protect proprietary details of their hardware. Second, the cost of adding support for new hardware is significantly reduced. Our experience adding support for new hardware to both Exo and Halide suggests that the library approach requires at least an order of magnitude less development time.

Memories

By default, all Exo buffers are assumed to reside in system DRAM and are managed using standard `malloc` and `free`. However, hardware accelerators often require modeling buffers that are resident in special accelerator memories, are pinned to special address ranges in the global address space, or otherwise exhibit strange behavior. To support these scenarios, Exo allows users to tag buffer and window types with a memory annotation. For example, `x : f32[n] @ MEM` says that the vector `x` lives in a custom memory `MEM`. These custom memories are defined by sub-classing a `Memory` base class (§3.2) and overloading methods.

Exo allows custom memories to change code generation for buffer `alloc`, `free`, and `windowing` via string interpolation. The author of a custom memory chooses whether to allow standard reading and writing the buffer (e.g., if the memory simply changes the memory management policy) or disable all usual accessing of the memory. The latter option is ideal for modeling hardware scratchpads, which should only be accessed using custom instructions. Such improper accesses are prevented by "backend checks." In general, memory annotations are ignored during scheduling.

Instructions

Instructions in Exo are procedures that are annotated with a macro/string-template. For example, given a vector load procedure with the signature `load(n : size, dst : f32[n], src : f32[n])`, we can make it into an instruction by annotating it with `@instr("hw_ld({src},{dst},{n})")` instead of `@proc`. When code generating calls to instructions, this annotation string is used instead of a sub-procedure call. Arguments are interpolated into the template as strings. This works as well for scheduling fine-grained intrinsics as it does for coarse-grained calls to existing microkernels or library calls.

As a result, the annotated Exo procedure has no effect on code generation, but instead serves as a semantic specification of the instruction for the purposes of checking correctness and program equivalence (for scheduling). This approach to an instruction mechanism has the following benefits and tradeoffs. First, programmers need not learn any additional specification language beyond Exo. Second, Exo entrusts programmers with the responsibility of verifying the link between the Exo procedure and annotation. Third and finally, programmers can use instructions in clever ways, including as an escape hatch. For example, a prefetch instruction can be modeled using a no-op procedure and thereby be inserted anywhere.

Configuration State

As we saw in §3.2, Exo models hardware configuration state via global structs of control variables annotated by `@config`. When defining configurations, programmers have the choice of realizing them as DRAM-resident data or disabling direct access to the configuration state (similar to disabling direct reading and writing of a memory). In the latter case, no global struct is generated.

3.3.3 Scheduling via Rewrites

Rather than directly writing code that uses a hardware library, Exo users transform a simple program into an equivalent, but more complex and high-performance version, targeted to the

Table 3.1: Some primitive Exo scheduling operators. Each operator rewrites $s_0 \rightsquigarrow s_1$ within a procedure `p`. This sort of rewrite-based scheduling makes it easier to expand the list of primitive operators, since the correctness of each operator is independent of the correctness of each other operator.

Command	Transform

`p.reorder(i,j)`	`for i:`
	`  for j:`
	$\rightsquigarrow$
	`for j:`
	`  for i:`

`p.split(i,c,io,ii)`	`for i<I:`
	$\rightsquigarrow$
	`for io<I/c:`
	`  for ii<c:`

`p.unroll(i)`	`for i:`
	$\rightsquigarrow$
	`for 0:`
	`...`

`p.inline(foo)`	inline a callsite of foo in p
`p.set_memory(a,MEM’)`	`a @ MEM` $\rightsquigarrow$ `a @ MEM’`
`p.set_precision(a,typ’)`	`a : typ` $\rightsquigarrow$ `a : typ’`
`p.call_eqv(foo,foo’)`	call foo’ at a callsite of foo

`p.bind_expr(a,a’)`	`s` $\rightsquigarrow$
	`a’ : R`
	`a’ = a`
	`s[a ↦ a’]`

`p.stage_mem(a,a’,s)`	`s` $\rightsquigarrow$
	`a’ : R[]`
	`for i:`
	`  a’ = a`
	`s[a |↦ a’]`
	`for i:`
	`  a = a’`

`p.bind_config(config,a)`	`s` $\rightsquigarrow$
	`config = a`
	`s[a ↦ config]`

`p.lift_alloc(a:R)`	`for i:`
	`  a : R`
	`  s`
	$\rightsquigarrow$
	`a : R`
	`for i:`
	`  s`

`p.fission_after(s1)`	`for i:`
	`  s1`
	`  s2`
	$\rightsquigarrow$
	`for i:`
	`  s1`
	`for i:`
	`  s2`

(continued on next page)

Table 3.1, continued

Command	Transform
p.reorder_stmts(s1,s2)	s1 s2 ⇝ s2 s1
p.configwrite_at(s,config,e)	s ⇝ s config = e
p.replace(s,foo)	s ⇝ foo(<<inferred>>)
p.add_guard(s,e)	s ⇝ if e: s else: s

p.fuse_loop(i)	for i:
	  s1
	for i:
	  s2
	⇝
	for i:
	  s1
	  s2

p.lift_if(if c: s)	for i:
	  if c: s
	⇝
	if c:
	  for i: s

p.partition_loop(i,c)	for i in lo,hi:
	⇝
	for i in lo,c:
	for i in c,hi:

p.remove_loop(i)	for i: s ⇝ s

specific hardware accelerator. This transformation is accomplished via successive rewriting of the application—a process called scheduling.

Because Exo is an embedded DSL, schedules are written as meta-programs in the host language (Python). Each primitive scheduling operator (table 3.1) takes a procedure `p` plus some other arguments as input, and returns an equivalent, rewritten procedure as output. Most of these operators require *pointing* at a location within the procedure. In our prototype, this is accomplished via simple syntactic pattern matching strings. For instance, `src : _` points at the first allocation of a buffer named `src`, and `for i in _: _ #2` points at the third loop in `p` with an iteration variable named `i`. This API is currently being re-designed, but was sufficient to demonstrate the benefits of rewrite-based scheduling.

Exo advances the idea of user-scheduling in two important ways. First, like Lift and Elevate [61, 141] but unlike Halide and TVM, scheduling operators are rewrites of programs, rather than arguments to a monolithic lowering process. As a result, the implementation and correctness of a scheduling primitive is independent of each other primitive. This makes the Exo implementation much simpler and easier to maintain. Importantly, Exo rewrites imperative rather than functional programs (Lift and Elevate). This makes checking the correctness of primitive rewrites more complex (§3.5 and 3.14).

Second, Exo supports scheduling of programs decomposed into procedures. This happens via the `inline()`, `call_eqv()`, and `replace()` primitives. `inline()` simply inlines a procedure's body at some call site, and `replace()` can be thought of as the inverse of `inline()` (see next section). `call_eqv()` on the other hand replaces a call to some sub-procedure `f` with a call to an equivalent sub-procedure `f'`. This equivalence is tracked by provenance, since the Exo system records the sequence of transformations by which `f` was transformed into `f'`. This concept of an *equivalent* sub-procedure is complicated by those scheduling primitives which pollute configuration state (e.g. `bind_config()`). To handle these, Exo tracks a lattice of different equivalence relations, modulo different parts of the configuration state (§3.6).

This provenance tracking system also enables an important optimization: when constructing SMT queries we may use the *simplest* equivalent (including configuration) definition of a procedure when constructing SMT queries. This is necessary to keep the cost of calling the solver low as scheduling complicates a procedure.

3.3.4 Code Replacement & Instruction Selection

The `replace()` scheduling primitive takes a designated statement block `s` and *replaces* it with a call to a designated sub-procedure `foo`. In particular, when `foo` is an `@instr`, this rewriting performs instruction selection. In other cases, it allows Exo programmers to manage code size trade-offs, as well as more neatly abstract and organize their code.

Our implementation of `replace()` is based on a form of unification modulo linear equalities. First, we attempt to unify (i.e. pattern match) the body of the sub-procedure `foo` with the designated statement block `s`. When doing this, the arguments of `foo` are designated as unknowns, the free variables of `s` as known symbols and any symbols introduced/bound in the body of `foo` or within `s` are unified. The ASTs are required to match exactly with respect to statements, and with respect to all expressions which are not simply integer typed control. Equivalences between integer typed control expressions are recorded as a system of linear equations to be solved in a second step.

If Exo did not support windowing, then we could determine expressions for the unknown argument variables by symbolically solving the resulting linear system of equations. However, the possibility of windowing expressions as arguments forces us to make categorical choices between different possible windowing expressions, resulting in disjunctions as well as conjunctions of linear equalities. For example, if replace is asked to infer a 1-dimensional window onto a 2-dimensional buffer `x`, it could infer an expression of the form `x[i,jlo:jhi]` or of the form `x[ilo:ihi,j]`. To handle this complication, we observe that all inferred integer expressions must be affine combinations of the known, free variables. Therefore, we can transform our symbolic linear system problem into a linear system in the unknown coefficients of these affine expressions. Once encoded in this way, we can discharge the problem to an SMT solver.

3.4 Formal Core Language

In order to define our program analysis, we provide a formal definition of the core of Exo, including a denotational semantics. The core idea is that statements denote store-transforming functions of type $\Sigma \to \Sigma$. Using these semantics we can define equivalence of Exo programs as functional equivalence of their denotations. A scheduling transformation can then be said to be safe when it transforms between equivalent Exo programs.

3.4.1 Mathematical Model of Exo Programs

The main concept in our mathematical model of Exo programs is the *store*, which represents the program state at any given point during its execution. The simplest model of a store $\sigma \in \Sigma$ would be a partial function from variable names to values. However, we must complicate this naive model in a few ways. Rather than present the full definitions (available in §3.9) we will focus on a high level gloss of the ideas here.

Control values are modeled as Boolean or integer values (in $\mathbb{B}$ and $\mathbb{Z}$) while data values are modeled as real numbers (in $\mathbb{R}$). Names of variables are drawn from a set of identifiers Name. Additionally, we rely on exceptional values to capture errors ϵ and unknown or uninitialized data $\perp$. For simplicity, we assume that all built in functions on data (basic arithmetic and the math library) are total, so that e.g. $0/0$ is not an error.

The first complication is that we need to model buffers and windows. Buffers can be thought of as maps from coordinate tuples to data $\mathbb{Z}^m \to (\mathbb{R} \uplus \{\perp, \epsilon\})$, where $\perp$ designates uninitialized but allocated memory, whereas ϵ designates out-of-bounds memory. These buffers are placed in the store Σ at special addresses $\ell \in \mathsf{Name}$ that are disjoint from names used in the program. Then windows can be modeled as a pair of a buffer address ℓ and affine-indexing function $\phi \in \mathbb{Z}^n \to \mathbb{Z}^m$. For instance, reading a window at coordinates i would translate to the lookup $\sigma(\ell)(\phi(i))$.

Having modeled buffers and windows, we can define stores $\sigma \in \Sigma$ as partial functions from Name to buffers, windows, or control values. In order to further capture the concept of program crashes (which should never happen for well-typed, well-bounded and assertion-satisfying programs) we expand the domain of stores to include the special value ϵ. We may assume that all functions are strict with respect to ϵ, meaning that once a program crashes it remains crashed.

3.4.2 Syntax, Semantics, and Well-Typed Programs

The syntax for the formal core of Exo is straightforward (fig. 3.2). The denotation of a statement or procedure s is written $\mathrm{S}[\![s]\!]$ and is a function $\Sigma \to \Sigma$. The full definition of denotations for expressions, statements and procedures are deferred to §3.9. Note again that this core language makes no reference to user-defined instructions or memories. This is because the core program analysis is blind to those features—which only affect code generation. This separation is what allows us to make the program analysis extensible to new hardware backends.

Our focus in this paper is not on basic type-checking (which is standard) or even bounds-checking and assertion-checking (which are straightforward based on prior work and repurposing our later analysis machinery). However, it is worth re-iterating what guarantees all of these front-end checks provide for Exo programs. First, all integer-valued control expressions are constrained to be quasi-affine. Second, all windowing and accessing of buffers and windows is statically guaranteed to be in-bounds. Lastly, any procedure call is guaranteed to satisfy the asserted pre-conditions of the called procedure. Mutation of non-global control values is also prohibited. The quasi-affine restriction in particular is what allows us to translate arbitrary control expressions into SAT queries modulo the Linear Integer Arithmetic (LIA) theory, and thus discharge problems to an SMT solver.

3.4.3 Program Equivalence

Definition 3.1 (program equivalence)**.** Let s_1, s_2 both be Stmt or Proc. These two programs are equivalent, written $s_1 \cong s_2$ when the store-transforming functions they denote are equivalent $\mathrm{S}\,[\![s_1]\!] = \mathrm{S}\,[\![s_2]\!]$ on valid input stores—i.e. stores which are not in an error state and satisfy any precondition assertions of s_1 and s_2, which are equivalent.

As discussed in §3.2, we often want to reason about programs that are equivalent "up-to/excluding a set of globals $\mathcal{L}$" because many transformations end up polluting configuration state. We define a lattice of weaker equivalence relations:

Definition 3.2 (program equivalence modulo globals)**.** Let s_1, s_2 both be Stmt or Proc, and let $\mathcal{L} \subseteq \mathsf{Name}_{global}$ be a set of globals to ignore. The two programs are equivalent "modulo $\mathcal{L}$", written $s_1 \cong_{\mathcal{L}} s_2$ when $\forall \sigma, x \notin \mathcal{L}.\ \mathrm{S}\,[\![s_1]\!]\,\sigma\ x = \mathrm{S}\,[\![s_2]\!]\,\sigma\ x$, with the same caveats about valid input stores.

3.5 Effect Analysis & Transformation of Programs

Our analysis of Exo programs is based on an *effect* analysis. An effect a extracted from a statement s characterizes which functions $f : \Sigma \to \Sigma$ the statement s could possibly denote $\mathrm{S}\,[\![s]\!]$. This effect analysis allows us to determine when code transformations like $s_1\,;s_2 \rightsquigarrow s_2\,;s_1$ and $s_1\,;s_2 \rightsquigarrow s_2$ are valid.

This analysis will require us to define (1) effect-expressions and environments, (2) a global symbolic data-flow analysis, (3) location sets as a symbolic abstraction of store locations, and finally (4) effects as an abstraction of programs. We can then state safety conditions for various program rewrites using these building blocks.

$$
\begin{array}{rcll}
\tau_a : \textsf{ArgType} & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{R}[e^*] & \\
\tau_s : \textsf{SigType} & ::= & (x : \tau_a) \to \tau_s \mid \texttt{unit} & \\
\tau : \textsf{Type} & ::= & \tau_a \mid \texttt{R} & \\
\multicolumn{4}{l}{\textit{note: we use } \cdot^* \textit{ to mean 0 or more}} \\
\hline
e : \textsf{Expr} & ::= & x & \text{variables} \\
 & \mid & op(e^*) & \text{built-in operations} \\
 & \mid & e[e^*] & \text{array read} \\
 & \mid & \texttt{win}(e, w^*) & \text{window expression} \\
w : \textsf{WinCoord} & ::= & e & \text{point-access} \\
 & \mid & e\,..\,e & \text{interval-access} \\
op \in & \multicolumn{3}{l}{\left\{ \begin{array}{l} +, -, *, /, \texttt{mod}, \texttt{and}, \texttt{or}, \texttt{not}, \\ ==, <, <=, >, >= \end{array} \right\} \cup \textit{Literals}} \\
\hline
s : \textsf{Stmt} & ::= & s ; s & \text{sequencing} \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ s & \text{guards} \\
 & \mid & \texttt{for}\ x\ \texttt{in}\ e\,..\,e\ \texttt{do}\ s & \text{sequential loops} \\
 & \mid & \texttt{alloc}\ \ x(e^*) & \text{array allocation} \\
 & \mid & e[e^*] = e & \text{array write} \\
 & \mid & e[e^*] \mathrel{+}= e & \text{array reduce} \\
 & \mid & x = e & \text{global write} \\
 & \mid & p(e^*) & \text{sub-procedure call} \\
\hline
pdef : \textsf{Proc} & ::= & \begin{array}{l} \texttt{proc}\ \ p : \tau_s \\ \quad \texttt{assert}\ e \\ \quad \texttt{do}\ s \end{array} & \\
\hline
L : \textsf{Lib} & ::= & \begin{array}{l} \texttt{globals}\ (x : \tau)^* \\ pdef^* \end{array} &
\end{array}
$$

Figure 3.2: Abstract Syntax for Exo core language

3.5.1 Ternary Logic

When extended with $\perp$, $\mathbb{B}$ becomes a ternary logic with the values true (true or T), false (false or F), and unknown ($\perp$). Intuitively, this ternary logic will allow us to distinguish between statements that are *definitely* true, and statements that *may be* true. As detailed in §3.10, this logic can be encoded in classical logic for the purposes of targeting SMT solvers.

We define two additional operators for collapsing back down from ternary to classical logic. $D\ p$ (“definitely p”) is defined by $DT = T$, $D\perp = F$, and $DF = F$; $M\ p$ (“maybe p”) is defined by $MT = T$, $M\perp = T$, and $MF = F$.

3.5.2 Effect Expressions

Effect Expressions both give us a way of expressing symbolic values and of encoding sentences in a first-order logic, for discharging to an SMT solver.

Definition 3.3 (Effect Expressions)**.** We define the following grammar of effect-expressions

$$ee : \mathsf{EffExpr} ::= x \mid c \mid \bot \mid op(ee^*) \mid ee?\ ee\ \mathsf{else}\ ee \mid \forall x.ee$$

where every expression either has sort bool or sort int. The operators are the same as the bool and int operators from fig. 3.2. Recall that in the case of int operators, the pseudo-affine condition means that the quotient for / and mod must be a constant, and one side of $*$ must be a constant.

Definition 3.4 (Effect Environments)**.**

$$\gamma : \mathsf{EffEnv} = (\mathsf{Name}_{global} \uplus \mathsf{Name}_{local}) \to \mathsf{EffExpr}$$

are partial functions that default to mapping x to x, not $\bot$.

Effect environments abstract functions $\Sigma \to \Sigma$ with respect to control values, not stores Σ. This is why they may appear to be impredicative (mapping x to x by default). We define substitution $\gamma(ee)$ in the usual way. Using this, we can define composition of two effect environments $(\gamma \cdot \gamma')x = \gamma(\gamma'(x))$, which may also be resolved by substituting with γ inside the expressions bound by γ'. This definition of substitution extends naturally up to our later definitions of location sets LocSet, and effects a.

3.5.3 Global Dataflow

The major complication in our program analyses is handling mutable, global control state—which makes precise analysis of program control logic undecidable. Our dataflow analysis is symbolic (producing effect environments as a result) and control-sensitive (symbolic values reflect guards wrapped around statements). However we must make some kind of approximation to force convergence on loops. We use a very simple heuristic, expressed symbolically: If every loop iteration does not change the value of a global variable x, then the loop behaves as an identity function. Otherwise, the loop drives x to the uncertain value $\bot$. This usually suffices because configuration state that depends on the loop iteration is usually meaningless outside of the loop.

We define global dataflow analysis **ValG** : $\mathsf{Stmt} \to \mathsf{EffEnv}$ precisely in §3.11, along with lifting of expressions to effect expressions $\mathit{Lift} : \mathsf{Expr} \to \mathsf{EffExpr}$.

3.5.4 Location Sets

Definition 3.5 (Location Set)**.**

$$\begin{aligned} \mathcal{L} : \mathsf{LocSet} \quad ::= \quad & \emptyset \mid \{x, ee^*\} \mid \mathcal{L} \cup \mathcal{L} \mid \textstyle\bigcup_x \mathcal{L} \\ \mid \quad & \mathcal{L} \cap \mathcal{L} \mid \mathcal{L} - \mathcal{L} \mid \mathsf{filter}(ee, \mathcal{L}) \end{aligned}$$

Location sets symbolically abstract sets of global and heap locations in the store.

These sets support a set membership predicate $(\_ \in \_) : (\mathsf{Name} \times \mathsf{EffExpr}^n) \to \mathsf{LocSet} \to \mathsf{EffExpr}$ and an is-empty predicate $(\_ = \emptyset) : \mathsf{LocSet} \to \mathsf{EffExpr}$, both in the expected way (see §3.12 for details)

Note that because effect expressions are a ternary logic, these location sets express upper and lower bounds on a set of locations: points definitely not in the set, points definitely in the set, and a penumbra of points ambiguously in the set. We collapse these sets down to "classical sets" using the aforementioned operators: $D\mathcal{L}$ meaning points definitely in the set, and $M\mathcal{L}$ meaning points that might be in the set. Thus $x \in D\mathcal{L}$ means $D(x \in Ls)$ and $x \notin M\mathcal{L}$ means $D(x \notin \mathcal{L})$.

3.5.5 Effects

Definition 3.6 (Effects)**.**

$$\begin{array}{rcl} a : \mathsf{Effect} & ::= & a\,;a \mid \emptyset \mid \mathsf{Guard}(ee, a) \mid \mathsf{Loop}(x, a) \\ & \mid & \mathsf{GlobalRead}(x) \mid \mathsf{GlobalWrite}(x) \\ & \mid & \mathsf{Read}(x, ee^*) \mid \mathsf{Write}(x, ee^*) \\ & \mid & \mathsf{Reduce}(x, ee^*) \mid \mathsf{Alloc}(x) \end{array}$$

This definition allows us to define the obvious translation of expressions ($\mathit{Eff}_e : \mathsf{Expr} \to \mathsf{Effect}$) and statements ($\mathit{Eff} : \mathsf{Stmt} \to \mathsf{Effect}$) into effects (see §3.13). Effects then allow us to define read, write, and reduce location sets.

To start, we define the set of buffers allocated by and visible to subsequent statements/effects:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{A} : \mathsf{Effect} \to \mathsf{LocSet}} \\ \mathbf{A}\ \mathsf{Alloc}(x) & = \{x\} \\ \mathbf{A}\ (a_1\,;a_2) & = \mathbf{A}(a_1) \cup \mathbf{A}(a_2) \\ \mathbf{A}\ \_ & = \emptyset \end{array}$$

Definition 3.7 (Locations of an Effect)**.** Let $\mathbf{Rd_G}$, $\mathbf{Wr_G}$, $\mathbf{Rd_H}$, $\mathbf{Wr_H}$, and $\mathbf{R{+}_H}$, be functions $\mathsf{Effect} \to \mathsf{LocSet}$. To avoid redundancy, define common cases for all such functions $\mathcal{F}$:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{F} : \mathsf{Effect} \to \mathsf{LocSet}} \\ \mathcal{F}\ \mathsf{Guard}(ee, a) & = \mathsf{filter}(ee, \mathcal{F}\ a) \\ \mathcal{F}\ \mathsf{Loop}(x, a) & = \bigcup_x \mathcal{F}\ a' \end{array}$$

Sequencing is defined differently for read and write sets:

$$\begin{array}{ll} \mathbf{Rd_G}\ (a_1\,;a_2) & = \mathbf{Rd_G}(a_1) \cup (\mathbf{Rd_G}(a_2') - \mathbf{Wr_G}(a_1) - \mathbf{A}(a_1)) \\ \mathbf{Wr_G}\ (a_1\,;a_2) & = \mathbf{Wr_G}(a_1) \cup (\mathbf{Wr_G}(a_2') - \mathbf{A}(a_1)) \\ \mathbf{Rd_H}\ (a_1\,;a_2) & = \mathbf{Rd_H}(a_1) \cup (\mathbf{Rd_H}(a_2') - \mathbf{Wr_H}(a_1) - \mathbf{A}(a_1)) \\ \mathbf{Wr_H}\ (a_1\,;a_2) & = \mathbf{Wr_H}(a_1) \cup (\mathbf{Wr_H}(a_2') - \mathbf{A}(a_1)) \\ \mathbf{R{+}_H}\ (a_1\,;a_2) & = \mathbf{R{+}_H}(a_1) \cup (\mathbf{R{+}_H}(a_2') - \mathbf{A}(a_1)) \end{array}$$

Each function detects its corresponding leaf-node:

$$
\begin{aligned}
&\mathbf{Rd_G}\ \mathsf{GlobalRead}(x) &&= \{x\}\\
&\mathbf{Wr_G}\ \mathsf{GlobalWrite}(x) &&= \{x\}\\
&\mathbf{Rd_H}\ \mathsf{Read}(x, ee_1, \ldots, ee_n) &&= \{x, ee_1, \ldots, ee_n\}\\
&\mathbf{Wr_H}\ \mathsf{Write}(x, ee_1, \ldots, ee_n) &&= \{x, ee_1, \ldots, ee_n\}\\
&\mathbf{R{+}_H}\ \mathsf{Reduce}(x, ee_1, \ldots, ee_n) &&= \{x, ee_1, \ldots, ee_n\}\\
&\mathcal{F}\ \_ &&= \emptyset
\end{aligned}
$$

From these five primitive sets we can define six other useful sets:

$$
\begin{aligned}
\mathbf{Rd}\ a &= \mathbf{Rd_G}\ a \cup \mathbf{Rd_H}\ a\\
\mathbf{Wr}\ a &= \mathbf{Wr_G}\ a \cup \mathbf{Wr_H}\ a\\
\mathbf{R+}\ a &= \mathbf{R{+}_H}\ a - \mathbf{Wr_H}\ a\\
\mathbf{All}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a \cup \mathbf{R{+}_H}\ a\\
\mathbf{Mod}\ a &= \mathbf{Wr}\ a \cup \mathbf{R{+}_H}\ a\\
\mathbf{RW}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a
\end{aligned}
$$

3.5.6 Effects as Abstraction

The different objects we have talked about so far each abstract some part of the program. For instance, the dataflow analysis of a statement $\mathbf{ValG}\,[\![s]\!]$ is an abstraction of its denotation $\mathrm{S}\,[\![s]\!]$ with respect to global values. Similarly, the effect extracted from an expression $\mathit{Eff}_e\,[\![e]\!]$ abstracts its denotation $\mathrm{E}\,[\![e]\!]$, and the effect extracted from a statement $\mathit{Eff}\,[\![s]\!]$ abstracts its denotation $\mathrm{S}\,[\![s]\!]$. But what do we mean by this?

The effect abstraction a for a statement s with denotation f guarantees a few properties. First, it provides an analogue of the "frame axiom" from separation logic. If a location x lies outside of $M\mathbf{Mod}(a)$, then it is unmodified: $f\sigma x = \sigma x$. Second, if a location is in the write set $x \in D\mathbf{Wr}(a)$, then the post-hoc value at that location $f\sigma x$ is determined solely by the values at read locations $y \in M\mathbf{Rd}(a)$. Third, if a location is reduced to $x \in D\mathbf{R+}(a)$, then the difference between the initial and final value at that location $f\sigma x - \sigma x$ is determined solely by values at read locations $y \in M\mathbf{Rd}(a)$. Finally, so long as the values at read locations $y \in M\mathbf{Rd}(a)$ are determined, then one of the three previous cases applies to every store location, even if we can't be certain which set(s) the location is in.

Even more simply in the case of expression abstraction, the effect a of an expression e with denotation $f : \Sigma \to \mathsf{Val}$ guarantees one property: The value $f\sigma$ is solely determined by the values at read locations $y \in M\mathbf{Rd}(a)$.

3.5.7 Basic Program Rewrites

The preceding analysis objects allow us to turn program equivalence checks into SMT queries.

Reorder statements The rewrite $s_1\,;s_2 \rightsquigarrow s_2\,;s_1$ is safe when $\mathsf{Commutes}\ \mathit{Eff}\,[\![s_1]\!]\ \ \mathit{Eff}\,[\![s_2]\!]$ holds. Commutativity of statements is defined as non-interference of effects. A special exception must be made for locations that are reduced.

Definition 3.8 (Commutativity)**.**

$$\begin{array}{l} \mathsf{Commutes}\ a_1\ a_2 = \\ D\left(\begin{array}{l} \mathbf{Wr}(a_1) \cap \mathbf{All}(a_2) = \emptyset\ \wedge\ \mathbf{Wr}(a_2) \cap \mathbf{All}(a_1) = \emptyset \\ \mathbf{R+}(a_1) \cap \mathbf{Rd}(a_2) = \emptyset\ \wedge\ \mathbf{R+}(a_2) \cap \mathbf{Rd}(a_1) = \emptyset \end{array}\right) \end{array}$$

Shadow statement The rewrite $s_1\,;s_2 \rightsquigarrow s_2$ is safe when $\mathsf{Shadows}\ \mathit{Eff}\,[\![s_1]\!]\ \ \mathit{Eff}\,[\![s_2]\!]$ holds. Whereas commutativity requires reasoning about what definitely doesn't intersect (and hence what memory might be touched), shadowing requires reasoning positively about what definitely is overwritten—which is why a one-sided approximation sets is insufficient.

Definition 3.9 (Shadowing)**.**

$$\begin{array}{l} \mathsf{Shadows}\ a_1\ a_2 = \\ \forall x \in M\mathbf{Mod}(a_1) \Rightarrow (x \notin M\mathbf{Rd}(a_2)\ \wedge\ x \in D\mathbf{Wr}(a_2)) \end{array}$$

New config write The rewrite $s \rightsquigarrow s\,;x_g{=}e$ is always safe, but only results in code that is equivalent modulo $\{x_g\}$. As we will soon see (§3.6.2), performing this rewrite in a context requires satisfying additional conditions, but in isolation it is very simple.

3.5.8 Loop Rewrites

When working with rewrites of loops, it is convenient to abbreviate notation for an iteration variable being in bounds. If the variable x occurs in `for` x `in` $e_{lo} .. e_{hi}$ `do` , then let $\mathsf{Bd}(x) = \mathit{Lift}\,[\![e_{lo}]\!] \le x < \mathit{Lift}\,[\![e_{hi}]\!]$ in the following.

Loop reordering One of the most basic non-trivial loop transformations is loop-reordering. When can we rewrite `for` x `do` `for` y `do` s into `for` y `do` `for` x `do` s? This transformation is valid when the loop bounds commute with the body, and when any loop iterations that are moved past each other commute. To formulate these conditions, let a_x be the effect of the x-loop's bound-expressions and a_y similarly for the y-loop. Let x', y' be copies of these iteration variables s.t. $s' = [x \mapsto x'][y \mapsto y']s$. Let $a = \mathit{Eff}\,[\![s]\!]$ and $a' = \mathit{Eff}\,[\![s']\!]$. Then the reordering condition may be precisely stated as

$$\begin{array}{l} (\forall x, y.\ M\mathsf{Bd}(x,y) \Rightarrow \mathsf{Commutes}((a_x\,;a_y), a)) \\ \wedge\left(\begin{array}{r} \forall x, y, x', y'.\ M(\mathsf{Bd}(x,y,x',y') \wedge x < x' \wedge y' < y) \\ \Rightarrow \mathsf{Commutes}(a, a') \end{array}\right) \end{array}$$

Loop fusion & fission Another basic loop transformation is to fuse two loops together, or in reverse to fission one loop in two. When can we rewrite $\texttt{for}\ x\ \texttt{do}\ s_1\texttt{;}s_2$ into $(\texttt{for}\ x\ \texttt{do}\ s_1)\texttt{;}\texttt{for}\ x\ \texttt{do}\ s_2$? This is possible when the loop bound commutes with the first statement, and when the statements that get reordered commute with each other. Letting a_x be the effect of the loop bounds, $a_1 = \mathit{Eff}\,[\![s_1]\!]$, $s_2' = [x \mapsto x']s_2$ and $a_2' = \mathit{Eff}\,[\![s_2']\!]$ we can state fission conditions precisely as

$$\begin{aligned}&(\forall x.\ M\mathsf{Bd}(x) \Rightarrow \mathsf{Commutes}(a_x, a_1))\ \wedge\\ &(\forall x, x'.\ M(\mathsf{Bd}(x, x') \wedge x' < x) \Rightarrow \mathsf{Commutes}(a_1, a_2'))\end{aligned}$$

Loop removal In order for the rewrite $\texttt{for}\ x\ \texttt{do}\ s \rightsquigarrow s$ to be safe, the variable x must not be free in s, s must be idempotent, and the loop must run for at least one iteration. If $a = \mathit{Eff}\,[\![s]\!]$, then these conditions are precisely

$$(\exists x.\ D\mathsf{Bd}(x))\ \wedge\ \mathsf{Shadows}(a, a)$$

3.6 Contextual Analyses

In order to make our program rewriting primitives useful, we must be able to modify some fragment of a procedure in a context. In this section, we define one-holed statement contexts, define how to process them, and extend equivalences between statements to account for context.

3.6.1 Contexts & Derived Quantities

Definition 3.10 (Contexts)**.**

$$\begin{aligned} C : \mathsf{Ctxt}\ \ ::=&\ \ \bullet \mid C\texttt{;}s \mid s\texttt{;}C \mid \texttt{for}\ x\ \texttt{in}\ e\,..\,e\ \texttt{do}\ C\\ \mid&\ \ \texttt{if}\ e\ \texttt{then}\ C \end{aligned}$$

The expression $C[s]$ means a statement resulting from substituting the hole ($\bullet$) in context C with statement s. Similarly, we can have a Proc context: $\texttt{proc}\ \ p : \tau_s\ \texttt{assert}\ e\ \texttt{do}\ C$.

We define three derived quantities from a context/statement pair C/s: (1) $\mathsf{CtrlPred}\,[\![C]\!]\,s : \mathsf{EffExpr}$, a predicate expressing under what conditions the statement s will execute; (2) $\mathsf{PreValG}\,[\![C]\!]\,s\ :\ \mathsf{EffEnv}$, capturing the dataflow values right before executing s; and (3) $\mathsf{PostEff}\,[\![C]\!]\,s : \mathsf{Effect}$, telling us the effect of context code that executes *after* s. See §3.14 for details.

3.6.2 Context Extension

Using these tools we can get from an argument of the form $s_1 \cong s_2$ back up to an argument of the form $C[s_1] \cong C[s_2]$. Thus, we can reach into the body of some procedure and perform a local rewrite, while maintaining equivalence of the overall procedure.

Consider a context C with statements s_1 and s_2, as well as a set of global names $\mathcal{L}$ to consider equivalence "up to."

$$\begin{aligned} \text{Let } p &= \mathsf{CtrlPred}\ C\ s_1 \\ \gamma &= \mathsf{PreValG}\ C\ s_1 \\ a &= \mathsf{PostEff}\ C\ s_1 \\ \mathcal{L}' &= M(\mathcal{L} - \mathbf{Wr_G}\ a) \\ s_1', s_2' &= \gamma(s_1), \gamma(s_2) \end{aligned}$$

$$\begin{array}{ll} \text{If} & (Mp \Rightarrow s_1' \cong_{\mathcal{L}} s_2') \wedge D(\mathbf{Rd_G}\ a \cap \mathcal{L} = \emptyset) \\ \text{Then} & C[s_1] \cong_{\mathcal{L}'} C[s_2] \end{array}$$

3.7 Case Studies

3.7.1 Gemmini

Using Exo, we developed highly-optimized schedules for Gemmini [51], a DNN accelerator, which significantly outperformed DNN kernel implementations that had been handwritten by Gemmini's designers.

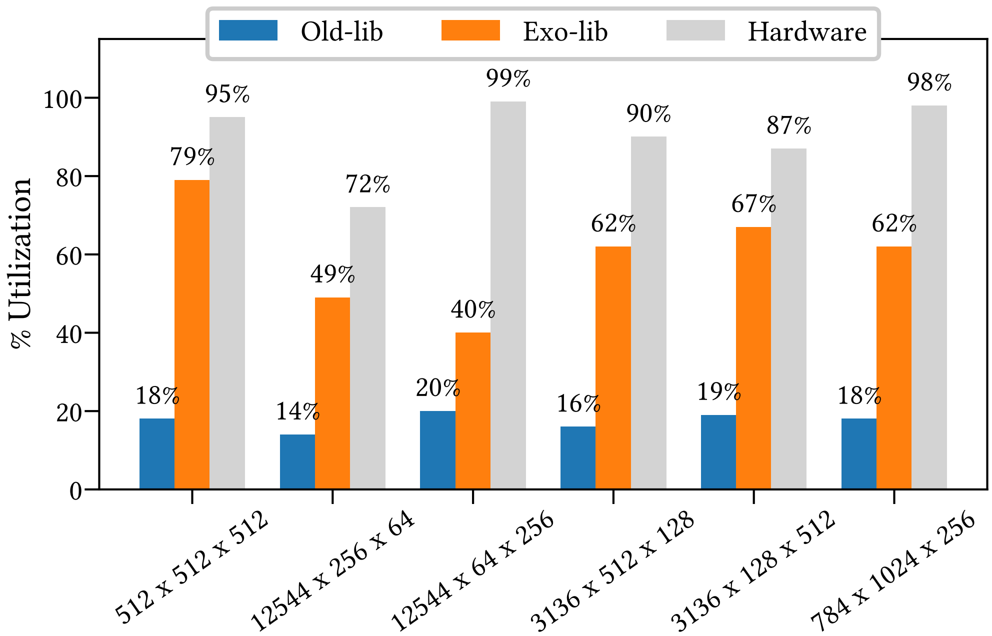
We targeted Gemmini's default architectural instantiation, which include a 16x16 systolic array that performs block matrix multiplications, a 256KB scratchpad for quantized inputs and weights, and a 64KB accumulator for partial sums. Gemmini's instruction set architecture (ISA) includes low-level instructions to move strided matrices to and from the scratchpad, as well as instructions to calculate dot products and perform non-linear activations on this data.

Gemmini also ships with a hand-written C library for common DNN kernels. This library wraps calls to Gemmini's low-level ISA in statically-scheduled, hand-tuned loops. However, Gemmini can also be built with hardware loop unrollers that dynamically schedule these kernels to maximize overlap between data loads, data stores, and matrix multiply operations. The hardware implementations typically run much faster than the software implementations at the cost of hardware complexity, area, power consumption, and reduced scheduling flexibility. The hardware kernels also have fixed loop orders and dataflows, while the software can adapt these to different tensor shapes.

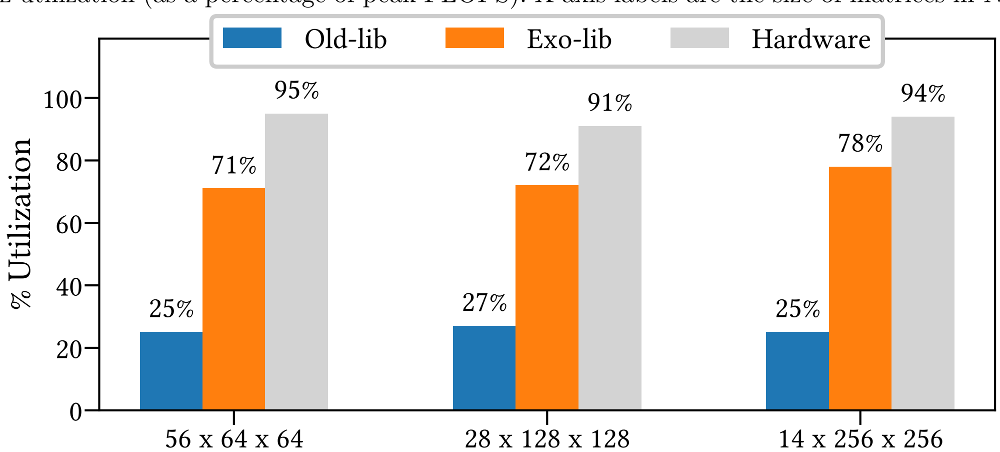
We implemented kernels for matrix multiply (MATMUL) and convolutional (CONV) layers in Exo and compared their performance against Gemmini's handwritten C library and hardware loop unrollers. The results are shown in figs. 3.3a and 3.3b, respectively. The tensor shapes in both are selected from those in a ResNet-50 DNN with a batch size of 4.

On average, Exo-generated code outperforms Gemmini's handwritten C library by 3.5× on the MATMUL sizes listed above, and achieves 67% of the performance of the hardware loop unrollers. For the convolutions listed, it runs 2.9× faster than the handwritten library, and is competitive with the hardware loop unroller, achieving 79% of its performance.

Note that the hardware loop unrollers use optional hardware resources (increasing area and power consumption) which are not available to Exo or the handwritten C library.



(a) MATMUL utilization (as a percentage of peak FLOPS). X axis labels are the size of matrices in $N \times M \times K$.



(b) CONV utilization (as a percentage of peak FLOPS). X axis labels are the shape of convolution in *output dimension* $\times$ *output channel* $\times$ *input channel*.

Figure 3.3: Performance of Exo-generated code on the Gemmini DNN accelerator. Exo-generated code achieves much higher performance than the DNN kernels hand-written by the designers of Gemmini (Old-lib). Gemmini's dynamically-scheduled hardware loop unrollers (Hardware) outperform Exo by using additional hardware resources, but therefore require additional chip area and power consumption.

However, we expect that changing Gemmini's ISA to support coarser-granularity instructions and better schedules may be able to close this performance gap in the future, providing software-programmable performance comparable to the inflexible hardware loop-unrollers.

Finally, Exo enabled faster co-design of Gemmini's hardware-software interface. When we started targeting Gemmini, its low-level hardware configuration instructions had many side effects which made optimizations difficult to reason about, limiting the performance we could achieve. We worked with the Gemmini hardware designers to disaggregate these configuration instructions into more orthogonal components; e.g. instructions which configured Gemmini's memory units would no longer have any side effects on the arithmetic units. 46 lines in Gemmini's handwritten C library had to be updated after this change, compared to only 5 in Exo's implementation. Exo made it easier for programmers to target fluid and changing hardware targets, which is common when developing new accelerators.

3.7.2 x86

As an acid test of the language design, we optimized matrix-matrix multiplication (SGEMM) for x86, where we can compare against state-of-the-art libraries that run near theoretical peak compute throughput. We chose to target single-core x86 with AVX512 extensions.[1]

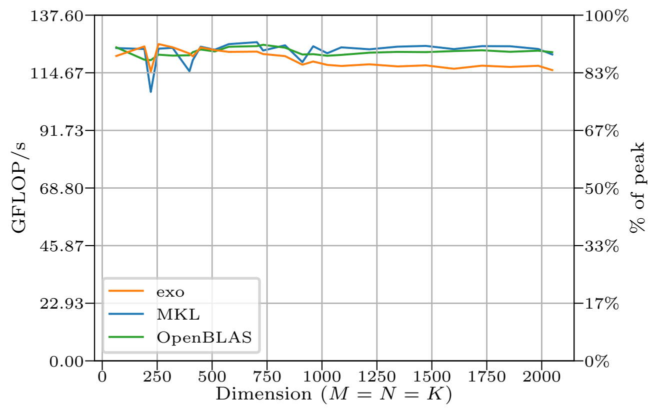
Recall that the computation is given by $C \mathrel{+}= A \cdot B$ where C is $M \times N$, A is $M \times K$, and B is $K \times N$. Our Exo implementation decomposes the problem as follows: at the deepest level of blocking, a register-blocked micro-kernel accumulates the inner dimension into a 6×64 panel of C, the output matrix. The level above the micro-kernel handles edge cases by dispatching to *specialized* versions of the micro-kernel for each edge case. Along the bottom, five distinct kernels are needed as they are always 64 elements wide and never 0 or 6 tall; similarly, four distinct kernels are needed along the right. The variable tail on the right edge is handled by masked loads. Finally, one level above this handles staging memory and blocking.

Every one of these routines was produced by scheduling and specializing a single, naive implementation of SGEMM consisting of three nested loops. Unification and equivalent-call replacement were crucial for avoiding any sort of error-prone, manual optimization.

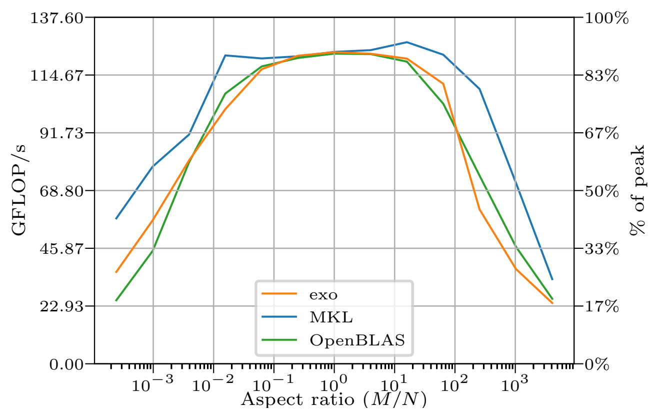
The performance results are shown in fig. 3.4. All benchmarks were run on an Intel i7-1185G7 at 4.3 GHz, a Tiger Lake CPU with AVX-512 instructions and peak single-precision floating-point performance of 137.60 GFLOPs. We tested our SGEMM against the hand-optimized implementations in Intel's MKL and the open-source OpenBLAS in two experiments. First (fig. 3.4a), we tested square matrices, so $M = N = K$. Each implementation performs quite closely (within measurement noise), between 80-95% of theoretical peak FLOPS across the parameter range.

Second (fig. 3.4b), we tested our SGEMM on a fixed workload, but with a variable aspect ratio for C. Specifically, we fix the inner dimension $K = 512$ and the product

[1]Although multi-core implementations are valuable, single-core workloads are representative of practice (ML inference in interactive web services is often run batch-parallel on single-core kernels), and the baselines are highly-optimized.



(a) SGEMM performance on square matrices. We approximately match other systems on square matrices.



(b) SGEMM performance with fixed workload and variable output aspect ratio. $K = 512$ and $M \times N = 512^2$, with the ratio of M to N varying. We match OpenBLAS performance across aspect ratios.

Figure 3.4: SGEMM performance compared to state-of-the-art libraries on x86. Benchmarks were run on one core of an Intel i7-1185G7 running at 4.3GHz.

$MN = 512^2$, then we sweep across the ratio M/N keeping the total FLOP count identical across experiments. Here, Exo matches OpenBLAS almost exactly, but MKL pulls ahead of both implementations when the aspect ratio is very far from square. MKL includes more specialized kernels for these extreme aspect ratios, which would be natural to do with further scheduling in Exo, as well.

For a final experiment, we tried to replicate the convolutional layer performance of a highly-tuned implementation provided by the Halide project. State of the art convolutions specialize or JIT-compile code templates to particular input, output, and kernel sizes. In Halide's case, it specialized to a batch size of 5, a kernel size of 3×3, an output size of 80×100, and 128 channels for both input and output. There is no padding and unit stride is used.

We configured Intel's oneDNN convolution to use these parameters and scheduled a basic

Impl.	N	W	H	IC	OC	% of peak
Exo	5	82	102	128	128	40.50%
Halide	5	82	102	128	128	40.59%
oneDNN	5	82	102	128	128	40.55%

Table 3.2: Summary of x86 CONV performance results. Single-threaded performance of various implementations with no padding and unit stride. A ReLU activation is applied. Benchmarks were run on an Intel i7-1185G7 running at 4.3GHz on a single core. The size was chosen to match the previously-published hand-scheduled Halide implementation. All three specialize or JIT to tune their code to specific sizes.

description of convolution in Exo to these parameters, too. The results are shown in table 3.2. Our CONV performs almost identically to the optimized baselines.

Overall, we believe these results show that Exo can be used to achieve performance competitive with state-of-the-art, highly hand-tuned libraries on x86.

3.7.3 Code Size

Table 3.3 summarizes some statistics regarding the size of Exo programs relative to handwritten C baselines.

On x86, our SGEMM schedule instantiates many specialized micro-kernels for handling loop tail cases at higher levels. Unlike Gemmini, it does not have SGEMM-specific hardware to utilize that might reduce the scheduling burden. Even so, the basic algorithm is expressed in 11 statements (the function signature, three loops, an accumulation statement, and a handful of size assertions) and 162 scheduling directives. The generated C code totals 831 source lines of code This already constitutes a nearly 5x code size reduction, but a comparison to OpenBLAS (an established open-source implementation) is even more favorable: at least 1690 source lines of code[2] make up that implementation. MKL is more complex, still.

Although the x86 conv implementation is "only" half the size of the equivalent generated C, it is much more flexible since other specialized versions can be quickly instantiated by meta-programming the schedule in Python. The size of the most comparable open-source implementation, Intel's oneDNN, is difficult to measure; just one file in the implementation measures well over 5000 source lines of code[3]. The size of the Halide code and schedule was nearly identical to ours: 64 relevant lines, compared to 62.

The story is similar for our Gemmini kernels. Both the matmul and conv Exo implementations are an order of magnitude smaller than the original, handwritten C implementations. The large generated code sizes reflect the high degree of loop unrolling in the generated schedules. A real application would likely either resort to the C preprocessor to manage this

[2] Summing the source line counts of the files mentioned in `kernel/x86_64/KERNEL.SKYLAKEX` for non-transposed SGEMM gives a very loose lower bound

[3] `src/cpu/x64/jit_avx512_common_conv_kernel.cpp`

App.	Platform	C (gen)	C (ref)	Alg.	Sched.
MATMUL	Gemmini	462	313	23	43
CONV	Gemmini	8317	450	26	44
SGEMM	x86	846	>1,690	11	162
CONV	x86	102	>5,400	23	39

Table 3.3: Source code sizes for matrix multiplication and convolutional layer on Gemmini and x86. Gemmini implements a fixed-point matrix multiply neural network layer (with fused ReLU activation), while x86 implements the BLAS SGEMM kernel. Both implement a standard 2D convolutional layer with ReLU activation. The Exo sources are counted in lines of code for the algorithm and number of directives for the schedule. This is compared to the size of both the Exo-generated C and state-of-the-art reference implementations (Gemmini standard library, OpenBLAS, and oneDNN, respectively) in source lines of code.

complexity, or not attempt the transformation at all (or as aggressively) beyond whatever the C compiler might choose to do automatically.

3.8 Limitations & Future Work

Multi-Core Semantics Although the instruction replacement directive (§3.3.4) enables users to access fine-grained intra-instruction or SIMD parallelism, Exo does not currently model multi-core parallelism. Naïvely, we could introduce a parallel for-loop with OpenMP-like semantics. Our effect analysis is powerful enough to conservatively check that different loop iterations touch strictly disjoint regions of memory. However, there is no single platform independent approach to threading—which clashes with our design goal of externalizing hardware backends. A more ambitious solution would find some way to externalize both the semantics and primitives associated with different kinds of threading. (e.g. pthreads, CUDA, MPI, etc.)

Alternatively, the `.replace()` directive applied to a no-op instruction can serve an escape hatch to, for example, inject OpenMP pragmas around a given loop. We tested this on our conv implementation and observed that our new implementation still matches Halide, while both pull ahead of oneDNN by 25% (flops) on 8 or more threads.

Automatic Scheduling We have not yet written any autoschedulers [1, 21, 115, 173] for Exo, but plan to. We expect Exo autoscheduling to differ from prior systems in two essential ways. First, because hardware targets are externalized, idiosyncratic, and frequently proprietary, we do not expect any one single autoscheduling strategy to work across all accelerators. Second, because Exo schedules are composable (as successive rewrites) rather than monolithic, Exo autoschedulers can also be developed compositionally. This opens up the possibility of developing libraries of re-usable mid-level scheduling operators built from semi-automated combinations of primitive scheduling operators. With time, whole suites of optimization passes could be written—entirely external to the Exo compiler.

3.9 Definitions for the core language

3.9.1 Mathematical Model of Exo programs

We present a simple denotational-style semantics. Our goal is to describe the set of stores/states, and store-transforming functions, which statements and procedures denote.

Definition 3.11 (names)**.** Our namespace Name is partitioned into three parts: *data*, *local*, and *global*.

We refer to booleans $\mathbb{B}$ and integers $\mathbb{Z}$ as *control values*, and "real numbers"[4] $\mathbb{R}$ as *data values*.

Definition 3.12 (exceptional values)**.** We use three kinds of exceptional values: unknown ($\perp$), memory error (ϵ_m), and type error (ϵ_τ), with unknowns for each basic type ($\perp_\mathbb{B}$, $\perp_\mathbb{Z}$, $\perp_\mathbb{R}$). We use an information order s.t. $\epsilon_\tau \sqsubseteq \epsilon_m \sqsubseteq \perp_\mathbb{R}$, $\epsilon_\tau \sqsubseteq \perp$, and values are otherwise un-ordered. A well-typed program ought not produce ϵ_τ, and a well-bounded program ought not produce ϵ_m.

Definition 3.13 (buffers)**.** The heap part of the store holds buffers, defined over all possible dimensionalities as partial functions $\mathsf{Buf} = \bigcup_{m=0}^{\infty} (\mathbb{Z}^m \to (\mathbb{R} \uplus \{\perp, \epsilon_m\}))$. These partial functions will default to ϵ_m as the least informative value. Thus, reading un-allocated memory results in a memory error.

Definition 3.14 (windows)**.** In Exo, memory is accessed through "windows," which are slices of multi-dimensional arrays. An n-dimensional window identifies a buffer in the heap and potential indexing transformation: $\mathsf{Win}_n = \mathsf{Name}_{data} \times \bigcup_{m=n}^{\infty} (\mathbb{Z}^n \to \mathbb{Z}^m)$. The functions $\phi \in \mathbb{Z}^n \to \mathbb{Z}^m$ must be defined as *injective translations* in the following sense: $\phi_i(x) = x_j + c$ or $\phi_i(x) = c$, and ϕ is injective (no two output coordinates depend on the same input coordinate).

Definition 3.15 (values)**.** The set of *control values* is $\mathsf{Val}_c = \mathbb{B} \uplus \mathbb{Z} \uplus \{\perp\}$. The set of *argument values* further includes windows $\mathsf{Val}_a = \mathsf{Val}_c \uplus \bigcup_{n=0}^{\infty} \mathsf{Win}_n$. Finally, the set of all values includes scalars and errors as well $\mathsf{Val} = \mathsf{Val}_a \uplus \mathbb{R} \uplus \{\epsilon_\tau, \epsilon_m\}$ The information ordering on exceptional values is extended s.t. all non-exceptional values x are pairwise unordered, and $\perp \sqsubseteq x$ with respect to each domain. This ordering forms a meet($\sqcap$) semi-lattice.

[4]As we discussed in the example (§3.2) our semantics is insensitive to questions of how the data values are approximated in finite precision—one may safely replace $\mathbb{R}$ in this paper with rationals $\mathbb{Q}$ without any loss of meaning.

Definition 3.16 (functions on values)**.** Given a function $f : D_1 \times D_2 \to \mathsf{Val}$ where $D_i \subseteq \mathsf{Val}$, the *extension* of f to all values $f' : \mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$ is $f'(x, y) = f(x, y)$ when $(x, y) \in D_1 \times D_2$, $f'(x, y) = x \sqcap y$ otherwise. Thus, exceptional values pre-empt each other, and applying a function to values of the wrong type produces a type error. In this way, functions are monotonic w.r.t. the information order. Arbitrary n-ary functions are extended similarly.

Definition 3.17 (stores)**.** A *store* (aka. *state*) is either an error value or a tuple of partial functions: $\Sigma = \{\epsilon_\tau, \epsilon_m\} \uplus (\Sigma_{data} \times \Sigma_{local} \times \Sigma_{global})$ where

$$\begin{aligned} \Sigma_{data} &= \mathsf{Name}_{data} \to \mathsf{Buf} \\ \Sigma_{local} &= \mathsf{Name}_{local} \to \mathsf{Val}_a \\ \Sigma_{global} &= \mathsf{Name}_{global} \to \mathsf{Val}_c \end{aligned}$$

and the default value in these partial functions is ϵ_τ. The information ordering is extended to stores as well. We write $\sigma(x)$ instead of $\sigma_{local}(x)$ when the meaning is clear from context.

Definition 3.18 (heap vs. stack)**.** When calling sub-procedures we need to restrict the store to only the non-local parts. For $\sigma \in \Sigma$, let $\mathsf{heap}(\sigma) = (\sigma_{data}, \emptyset, \sigma_{global})$ for non-error stores. When returning from a procedure call, we overwrite the local heap and globals with the results of running the sub-procedure: $\sigma[\mathsf{heap} \mapsto \sigma'] = (\sigma'_{data}, \sigma_{local}, \sigma'_{global})$.

Definition 3.19 (functions on stores)**.** Given a function $f : (\Sigma - \{\epsilon_\tau, \epsilon_m\}) \to \Sigma$, we lift it to a function $\Sigma \to \Sigma$ by propagating error values, as expected. Thus, store functions are monotonic.

$$\mathrm{E} : \mathsf{Expr} \to \Sigma \to \mathsf{Val}$$

$$\begin{aligned} \mathrm{E}\,[\![x]\!]\,\sigma &= \sigma(x) \\ \mathrm{E}\,[\![op\,(e_1, \ldots, e_k)]\!]\,\sigma &= \widehat{op}(\mathrm{E}\,[\![e_1]\!]\,\sigma, \ldots, \mathrm{E}\,[\![e_k]\!]\,\sigma) \\ \mathrm{E}\,[\![e_0\,[e_1, \ldots, e_n]]\!]\,\sigma &= \sigma(\ell, \varphi(e'_1, \ldots, e'_n)) \\ \mathrm{E}\,[\![\mathtt{win}(e_0, w_1, \ldots, w_m)]\!]\,\sigma &= (\ell, \varphi \circ \phi_{win}(w'_1, \ldots, w'_m)) \\ \mathrm{E}\,[\![e_{lo}\,..\,e_{hi}]\!]\,\sigma &= (\mathrm{E}\,[\![e_{lo}]\!]\,\sigma, \mathrm{E}\,[\![e_{hi}]\!]\,\sigma) \\ \text{where} &\quad \ell, \varphi = \mathrm{E}\,[\![e_0]\!]\,\sigma \\ e'_i = \mathrm{E}\,[\![e_i]\!]\,\sigma &\quad w'_i = \mathrm{E}\,[\![w_i]\!]\,\sigma \end{aligned}$$

$$\begin{aligned} \phi_{win}() &= () \\ \phi_{win}(i, u_2, \ldots) &= p(i) \circ \phi_{win}(u_2, \ldots) \\ \phi_{win}((i_{lo}, i_{hi}), u_2, \ldots) &= \lambda x.cons(x + i_{lo}) \circ \phi_{win}(u_2, \ldots) \\ cons(i, (y_1, \ldots, y_n)) &= (i, y_1, \ldots, y_n) \end{aligned}$$

Figure 3.5: Expression Denotations

3.9.2 Syntax and Semantics of Exo programs

The syntax and denotations for Exo programs are given in figs. 3.2 and 3.5 to 3.7.

$$
\begin{array}{l}
\mathrm{S} : \mathsf{Stmt} \to \Sigma \to \Sigma \\
\begin{aligned}
\mathrm{S}\,[\![s_1 \,;\, s_2]\!]\,\sigma \;&=\; (\mathrm{S}\,[\![s_2]\!] \circ \mathrm{S}\,[\![s_1]\!])\,\sigma \\
\mathrm{S}\,[\![\texttt{if}\ e\ \texttt{then}\ s]\!]\,\sigma \;&=\; \begin{cases} \mathrm{S}\,[\![s]\!]\,\sigma, & \text{if } \ \mathrm{E}\,[\![e]\!]\,\sigma = \mathsf{true} \\ \sigma, & \text{otherwise} \end{cases} \\
\mathrm{S}\,[\![\texttt{for}\ x\ \texttt{in}\ e_{lo} \,..\, e_{hi}\ \texttt{do}\ s]\!]\,\sigma \;&=\; \phi_{m-1} \circ \cdots \circ \phi_n \\
&\quad \text{where } n, m = \mathrm{E}\,[\![e_{lo}]\!]\,\sigma, \mathrm{E}\,[\![e_{hi}]\!]\,\sigma \\
&\quad\quad\quad\;\; \phi_k = \lambda\sigma'.\mathrm{S}\,[\![s]\!]\,(\sigma'[i \mapsto k]) \\
\mathrm{S}\,[\![\texttt{alloc}\ \ x(e_1, \ldots, e_k)]\!]\,\sigma \;&=\; \sigma \begin{bmatrix} \ell \mapsto \mathsf{buf}(\vec{e'}) \\ x \mapsto (\ell, id) \end{bmatrix} \\
&\quad \text{where } \ell \text{ is a fresh name} \\
&\quad\quad\quad\;\; e'_i = \mathrm{E}\,[\![e_i]\!]\,\sigma \\
\mathrm{S}\,[\![x \texttt{ = } e]\!]\,\sigma \;&=\; \sigma[x \mapsto \mathrm{E}\,[\![e]\!]\,\sigma] \\
\mathrm{S}\,[\![e_0\,\texttt{[}e_1, \ldots, e_k\texttt{]}\texttt{ = } e_{rhs}]\!]\,\sigma \;&=\; \sigma\,[(\ell, \hat{\imath}) \mapsto e'_{rhs}] \\
\mathrm{S}\,[\![e_0\,\texttt{[}e_1, \ldots, e_k\texttt{]}\texttt{ += } e_{rhs}]\!]\,\sigma \;&=\; \sigma\,[(\ell, \hat{\imath}) \oplus\!\!\!\to e'_{rhs}] \\
&\quad \text{where } \ell, \varphi = \mathrm{E}\,[\![e_0]\!]\,\sigma \\
&\quad\quad\quad\;\; \hat{\imath} = \varphi(e'_1, \ldots, e'_k) \\
\mathrm{S}\,[\![p(e_1, \ldots, e_n)]\!]\,\sigma \;&=\; \sigma[\mathsf{heap} \mapsto \sigma'] \\
&\text{where } \sigma' = \mathsf{call}(p, \vec{e'}, \sigma)
\end{aligned} \\
\hline
\text{for } \texttt{proc}\ p : (x_1 : \tau_1) \to \cdots (x_n : \tau_n) \to ()\ \texttt{assert}\ e\ \texttt{do}\ s, \\
\begin{aligned}
\mathsf{arg}_i(p) \;&=\; x_i \\
\mathsf{call}(p, \vec{e'}, \sigma) \;&=\; \mathrm{P}\,[\![p]\!]\,(\mathsf{heap}(\sigma)[\mathsf{arg}_i(p) \mapsto e'_i]) \\
\mathsf{buf}(\vec{e'}) \;&=\; [\vec{u} \mapsto \bot \mid 0 \le u_i < e'_i]
\end{aligned}
\end{array}
$$

Figure 3.6: Statement Denotations

$$
\begin{array}{l}
\mathrm{P} : \mathsf{Proc} \to \Sigma \to \Sigma \\
\mathrm{P} \left[\!\!\left[\begin{array}{l} \texttt{proc}\ \ p : \tau_s \\ \quad \texttt{assert}\ e \\ \quad \texttt{do}\ s \end{array} \right]\!\!\right] \sigma \;=\; \begin{cases} \epsilon_m, & \text{if } \ \mathrm{E}\,[\![e]\!]\,\sigma \neq \mathsf{true} \\ \mathrm{S}\,[\![s]\!]\,\sigma, & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.7: Procedure Denotations

3.10 Encoding Ternary Logic

Recall our language of effect-expressions

$$ee : \mathsf{EffExpr} ::= x \mid c \mid \bot \mid op(ee^*) \mid ee?\ ee\ \mathsf{else}\ ee \mid \forall x.ee$$

with $op \in \{+, -, *, /, \mathsf{mod}, \wedge, \vee, \neg, =, <, >, \leq, \geq\}$

We may encode this language for standard SMT solvers as follows. Represent a value in $\mathbb{Z} \cup \{\bot\}$ by a value in $\mathbb{Z} \times \mathbb{B}$ and a value in $\mathbb{B} \cup \{\bot\}$ by a value in $\mathbb{B} \times \mathbb{B}$, s.t. the meaning of the encoding is defined by $\mathsf{Val}(x, \mathsf{true}) = x$ and $\mathsf{Val}(x, \mathsf{false}) = \bot$. Accordingly, we can pull back the definitions of the various operators as follows.

For $op \in \{+, -, *, /, \mathsf{mod}, =, <, >, \leq, \geq\}$ with inputs of int sort, the rule is simple and illustrated by $+$:

$$(x_1, d_1) + (x_2, d_2) \mapsto (x_1 + x_2, d_1 \wedge d_2)$$

For operators on inputs of bool sort, we may take advantage of Boolean short-circuiting to produce known values even when some input is unknown. Such definitions are key to extracting useful information from the ternary logic. These operators are

$$\begin{array}{rcl}
(x_1, d_1) \wedge (x_2, d_2) & \mapsto & \left(x_1 \wedge x_2,\ \begin{array}{l} (d_1 \wedge d_2) \vee \\ (\neg x_1 \wedge d_1) \vee \\ (\neg x_2 \wedge d_2) \end{array} \right) \\
(x_1, d_1) \vee (x_2, d_2) & \mapsto & \left(x_1 \vee x_2,\ \begin{array}{l} (d_1 \wedge d_2) \vee \\ (x_1 \wedge d_1) \vee \\ (x_2 \wedge d_2) \end{array} \right) \\
\neg(x, d) & \mapsto & (\neg x, d) \\
\forall x_1.(x_2, d_2) & \mapsto & \left((\forall x_1.x_2),\ \begin{array}{l} (\forall x_1.d_2) \vee \\ (\exists x_1.(\neg x_2 \wedge d_2)) \end{array} \right) \\
\multicolumn{3}{l}{(x_1, d_1)?\ (x_2, d_2)\ \mathsf{else}\ (x_3, d_3) \mapsto} \\
\multicolumn{3}{r}{(x_1?\ x_2\ \mathsf{else}\ x_3, d_1 \wedge (x_1?\ d_2\ \mathsf{else}\ d_3))}
\end{array}$$

3.11 Global Dataflow Definitions

We specify a lifting from expressions to $\mathsf{EffExpr}$ in a reasonably obvious way

Definition 3.20 (lifting effect-expressions).

$$
\begin{array}{ll}
\mathit{Lift} : \mathsf{Expr} \to \mathsf{EffExpr} & \\
\mathit{Lift}\,[\![x]\!] & = \begin{cases} (x, id), & \text{if } x : \texttt{R[...]} \\ x, & \text{otherwise} \end{cases} \\
\mathit{Lift}\,[\![op(e_1, \ldots, e_n)]\!] & = \begin{array}{rl} \mathsf{let}\ ee_i = & \mathit{Lift}\,[\![e_i]\!] \\ \mathsf{in} & op(ee_1, \ldots, ee_n) \end{array} \\
\mathit{Lift}\,[\![e_0[e_1, \ldots, e_n]]\!] & = \bot \\
\mathit{Lift}\,[\![\texttt{win}(e_0, w_1, \ldots, w_n)]\!] & = \left[\begin{array}{rl} \mathsf{let}\ x, \varphi & = \mathit{Lift}\,[\![e_0]\!] \\ w_i' & = \mathit{Lift}\,[\![w_i]\!] \\ \varphi' & = \varphi \circ \phi_{win} \\ \mathsf{in} & (x, \varphi'(w_1', \ldots, w_n')) \end{array} \right.
\end{array}
$$

Global dataflow analysis is defined precisely as follows:

Definition 3.21 (Global Values).

$$
\begin{array}{ll}
\mathbf{ValG} : \mathsf{Stmt} \to \mathsf{EffEnv} & \\
\mathbf{ValG}\,[\![s_1\,;s_2]\!] & = (\mathbf{ValG}\ s_1) \cdot (\mathbf{ValG}\ s_2) \\
\mathbf{ValG}\,[\![\texttt{if}\ e\ \texttt{then}\ s]\!] & = [x \mapsto v_{\text{if}} \mid x \in G] \\
 & \quad \text{where } v_{\text{if}} = (\mathit{Lift}\,[\![e]\!]?\ G\ x\ \mathsf{else}\ x) \\
\mathbf{ValG}\,[\![\texttt{for}\ i\ \texttt{in}\ e_{lo}\,..\,e_{hi}\ \texttt{do}\ s]\!] & \\
 & = [x \mapsto (\mathsf{fix}\ x)?\ x\ \mathsf{else}\ \bot \mid x \in G] \\
 & \quad \text{where } G = \mathbf{ValG}\,[\![s]\!] \\
 & \qquad bd_i = \mathit{Lift}\,[\![e_{lo}]\!] \le i < \mathit{Lift}\,[\![e_{hi}]\!] \\
 & \qquad F_{i,x} = (bd_i \Rightarrow G\ x = x) \\
 & \qquad \mathsf{fix}\ x = \forall i : \mathsf{int}.\ F_{i,x} \\
\mathbf{ValG}\,[\![x = e]\!] & = [x \mapsto \mathit{Lift}\,[\![e]\!]] \\
\mathbf{ValG}\ \_ & = \emptyset
\end{array}
$$

3.12 Location Set Membership

$$
\begin{array}{rl}
\in : \mathsf{Name} \times \mathsf{EffExpr}^n \to \mathsf{LocSet} \to \mathsf{EffExpr} & \\
xs \in \emptyset = & \mathsf{false} \\
xs \in \{y, ee_1, \ldots, ee_n\} = & x = y \wedge \bigwedge_{i=0}^{n} ee_i' = ee_i \\
 & \text{where } xs = (x, ee_1', \ldots, ee_n') \\
xs \in \mathcal{L}_1 \cup \mathcal{L}_2 = & (xs \in \mathcal{L}_1) \vee (xs \in \mathcal{L}_2) \\
xs \in \bigcup_x \mathcal{L} = & \exists_x (xs \in \mathcal{L}) \\
xs \in \mathcal{L}_1 \cap \mathcal{L}_2 = & (xs \in \mathcal{L}_1) \wedge (xs \in \mathcal{L}_2) \\
xs \in \mathcal{L}_1 - \mathcal{L}_2 = & (xs \in \mathcal{L}_1) \wedge (xs \notin \mathcal{L}_2) \\
xs \in \mathsf{filter}(ee, \mathcal{L}) = & ee \wedge (xs \in \mathcal{L})
\end{array}
$$

$$
\begin{array}{l}
(\_ = \emptyset) : \mathsf{LocSet} \to \mathsf{EffExpr} \\
(\mathcal{L} = \emptyset) = \forall xs.(xs \notin \mathcal{L})
\end{array}
$$

3.13 Effect Extraction

Definition 3.22 (Effect of an Expression)**.** In the following, we sometimes combine effects using $\cup$ in place of ; to indicate that since we are strictly combining read-effects, the order of composition is irrelevant.

$$
\begin{aligned}
\mathit{Eff}_e[\![x]\!] &= \begin{cases} \mathsf{GlobalRead}(x), & \text{if } x \in \mathsf{Name}_{glob} \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathit{Eff}_e[\![op(e_1,\ldots,e_n)]\!] &= \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e[\![e_i]\!] \\
\mathit{Eff}_e[\![e_0[e_1,\ldots,e_n]]\!] &= \left[\begin{array}{rl} \mathsf{let}\ x,\varphi = & \mathit{Lift}[\![e_0]\!] \\ ee_i = & \mathit{Lift}[\![e_i]\!] \\ a' = & \bigcup_{i=0}^{n} \mathit{Eff}_e[\![e_i]\!] \\ \varphi_{ee} = & \varphi(ee_1,\ldots,ee_n) \\ \mathsf{in} & a' \cup \mathit{Read}(x,\varphi_{ee}) \end{array}\right. \\
\mathit{Eff}_e[\![\mathtt{win}(e_0,w_1,\ldots,w_n)]\!] &= \mathit{Eff}_e[\![e_0]\!] \cup \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e[\![w_i]\!]
\end{aligned}
$$

Definition 3.23 (Effect of a Statement)**.**

$$
\begin{aligned}
&\mathit{Eff}[\![s_1\,;s_2]\!] \\
&\quad = \mathit{Eff}[\![s_1]\!]\,;\mathbf{ValG}[\![s_1]\!](\mathit{Eff}[\![s_2]\!]) \\
&\mathit{Eff}[\![\mathtt{if}\ e\ \mathtt{then}\ s]\!] \\
&\quad = \mathit{Eff}_e[\![e]\!]\,;\mathsf{Guard}(\mathit{Lift}[\![e]\!],\mathit{Eff}[\![s]\!]) \\
&\mathit{Eff}[\![\mathtt{for}\ x\ \mathtt{in}\ e_{lo}\,..\,e_{hi} s\ \mathtt{do}\ ]\!] \\
&\quad = \left[\begin{array}{l} (\mathit{Eff}_e[\![e_{lo}]\!] \cup \mathit{Eff}_e[\![e_{hi}]\!]); \\ \mathsf{Loop}(x,\mathsf{Guard}(bds, G(\mathit{Eff}[\![s]\!]))) \end{array}\right. \\
&\quad\ \text{where } bds = \mathit{Lift}[\![e_{lo}]\!] \le x < \mathit{Lift}[\![e_{hi}]\!] \\
&\qquad\qquad G = \mathbf{ValG}[\![\mathtt{for}\ x\ \mathtt{in}\ e_{lo}\,..\,e_{hi} s\ \mathtt{do}\ ]\!] \\
&\mathit{Eff}[\![\mathtt{alloc}\ \ x(e_1,\ldots,e_n)]\!] \\
&\quad = \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e[\![e_i]\!] \\
&\mathit{Eff}[\![e_0\,[e_1,\ldots,e_n]\,\mathtt{=}\ e_r]\!] \\
&\quad = \left[\begin{array}{rl} \mathsf{let}\ x,\varphi = & \mathit{Lift}[\![e_0]\!] \\ ee_i = & \mathit{Lift}[\![e_i]\!] \\ \mathsf{in} & (\mathit{Eff}_e[\![e_r]\!] \cup \bigcup_{i=0}^{n} \mathit{Eff}_e[\![e_i]\!])\ ; \\ & \mathsf{Write}(x,\varphi(ee_1,\ldots,ee_n)) \end{array}\right. \\
&\mathit{Eff}[\![e_0\,[e_1,\ldots,e_n]\,\mathtt{+=}\ e_r]\!] \\
&\quad = \left[\begin{array}{rl} \mathsf{let}\ x,\varphi = & \mathit{Lift}[\![e_0]\!] \\ ee_i = & \mathit{Lift}[\![e_i]\!] \\ \mathsf{in} & (\mathit{Eff}_e[\![e_r]\!] \cup \bigcup_{i=0}^{n} \mathit{Eff}_e[\![e_i]\!])\ ; \\ & \mathsf{Reduce}(x,\varphi(ee_1,\ldots,ee_n)) \end{array}\right. \\
&\mathit{Eff}[\![x\ \mathtt{=}\ e]\!] \\
&\quad = \mathit{Eff}_e[\![e]\!]\,;\mathsf{GlobalWrite}(x)
\end{aligned}
$$

3.14 Context Analysis

Definition 3.24 (Control Predicate).

$$
\begin{aligned}
&\mathsf{CtrlPred} : \mathsf{Ctxt} \to \mathsf{Stmt} \to \mathsf{EffExpr}\\
&\mathsf{CtrlPred}\,[\![\bullet]\!]\,s && = \mathsf{true}\\
&\mathsf{CtrlPred}\,[\![C\,;s_2]\!]\,s && = \mathsf{CtrlPred}\,[\![C]\!]\,s\\
&\mathsf{CtrlPred}\,[\![s_1\,;C]\!]\,s && = \mathbf{ValG}\,[\![s_1]\!]\,(\mathsf{CtrlPred}\,[\![C]\!]\,s)\\
&\mathsf{CtrlPred}\,[\![\texttt{if}\ e\ \texttt{then}\ C]\!]\,s && = \mathit{Lift}\,[\![e]\!] \wedge \mathsf{CtrlPred}\,[\![C]\!]\,s\\
&\mathsf{CtrlPred}\,[\![\texttt{for}\ x\ \texttt{in}\ e_{lo} \mathinner{..} e_{hi}\ \texttt{do}\ C]\!]\,s\\
& && = \mathit{bds} \wedge G(\mathsf{CtrlPred}\,[\![C]\!]\,s)\\
&\quad \text{where } s_b = [x \mapsto x'](C[s])\\
&\qquad\quad G = \mathbf{ValG}\,[\![\texttt{for}\ x'\ \texttt{in}\ e_{lo} \mathinner{..} x\ \texttt{do}\ s_b]\!]\\
&\qquad\quad \mathit{bds} = \mathit{Lift}\,[\![e_{lo}]\!] \le x < \mathit{Lift}\,[\![e_{hi}]\!]\\
&\mathsf{CtrlPred}\,[\![\texttt{proc}\ \ p : \tau_s\ \texttt{assert}\ e\ \texttt{do}\ C]\!]\,s\\
& && = \mathit{Lift}\,[\![e]\!] \wedge \mathsf{CtrlPred}\,[\![C]\!]\,s
\end{aligned}
$$

Definition 3.25 (Pre-statement Global Values).

$$
\begin{aligned}
&\mathsf{PreValG} : \mathsf{Ctxt} \to \mathsf{Stmt} \to \mathsf{EffEnv}\\
&\mathsf{PreValG}\,[\![\bullet]\!]\,s && = \emptyset\\
&\mathsf{PreValG}\,[\![C\,;s_2]\!]\,s && = \mathsf{PreValG}\,[\![C]\!]\,s\\
&\mathsf{PreValG}\,[\![s_1\,;C]\!]\,s && = (\mathbf{ValG}\,[\![s_1]\!]) \cdot (\mathsf{PreValG}\,[\![C]\!]\,s)\\
&\mathsf{PreValG}\,[\![\texttt{if}\ e\ \texttt{then}\ C]\!]\,s && = \mathsf{PreValG}\,[\![C]\!]\,s\\
&\mathsf{PreValG}\,[\![\texttt{for}\ x\ \texttt{in}\ e_{lo} \mathinner{..} e_{hi}\ \texttt{do}\ C]\!]\,s\\
& && = G \cdot (\mathsf{PreValG}\,[\![C]\!]\,s)\\
&\quad \text{where } s_b = [x \mapsto x'](C[s])\\
&\qquad\quad G = \mathbf{ValG}\,[\![\texttt{for}\ x'\ \texttt{in}\ e_{lo} \mathinner{..} x\ \texttt{do}\ s_b]\!]
\end{aligned}
$$

Definition 3.26 (Post-statement Effect).

$$
\begin{aligned}
&\mathsf{PostEff} : \mathsf{Ctxt} \to \mathsf{Stmt} \to \mathsf{Effect}\\
&\mathsf{PostEff}\,[\![\bullet]\!]\,s && = \emptyset\\
&\mathsf{PostEff}\,[\![C\,;s_2]\!]\,s && = (\mathsf{PostEff}\,[\![C]\!]\,s)\,;\mathbf{ValG}\ C[s](a_2)\\
& && \quad\text{where } a_2 = \mathit{Eff}\,[\![s_2]\!]\\
&\mathsf{PostEff}\,[\![s_1\,;C]\!]\,s && = \mathbf{ValG}\,[\![s_1]\!]\,(\mathsf{PostEff}\,[\![C]\!]\,s)\\
&\mathsf{PostEff}\,[\![\texttt{if}\ e\ \texttt{then}\ C]\!]\,s && = \mathsf{Guard}(\mathit{Lift}\,[\![e]\!]\,, \mathsf{PostEff}\,[\![C]\!]\,s)\\
&\mathsf{PostEff}\,[\![\texttt{for}\ x\ \texttt{in}\ e_{lo} \mathinner{..} e_{hi}\ \texttt{do}\ C]\!]\,s\\
& && = (\gamma \cdot G)(a_1\,;G_x(a_2))\\
&\quad \text{where } s_b = [x \mapsto x'](C[s])\\
&\qquad\quad \gamma = [x_{lo} \mapsto \mathit{Lift}\,[\![e_{lo}]\!]][x_{hi} \mapsto \mathit{Lift}\,[\![e_{hi}]\!]]\\
&\qquad\quad G = \mathbf{ValG}\,[\![\texttt{for}\ x'\ \texttt{in}\ x_{lo} \mathinner{..} x\ \texttt{do}\ s_b]\!]\\
&\qquad\quad a_1 = \mathsf{Guard}(x_{lo} \le x' < x_{hi}, \mathsf{PostEff}\,[\![C]\!]\,s)\\
&\qquad\quad G_x = \mathbf{ValG}\,[\![\texttt{if}\ x_{lo} \le x < x_{hi}\ \texttt{then}\ s_b]\!]\\
&\qquad\quad a_2 = \mathit{Eff}\,[\![\texttt{for}\ x'\ \texttt{in}\ x+1 \mathinner{..} x_{hi}\ \texttt{do}\ s_b]\!]
\end{aligned}
$$

Note that all three of these rules expose the loop iteration variable as a free-variable in the resulting object. This represents the “current loop iteration”. If a property can be shown for all values of this free-variable, then we can recover the property by induction.

3.15 Gemmini User Library

This section includes a user library defined for Gemmini accelerators. It consists of full definitions of Gemmini scratchpad (`GEMM_SCRATCH`), Gemmini accumulator (`GEMM_ACCUM`), Gemmini load instruction (`ld_i8`), and Gemmini load configuration and a config load instruction (ConfigLoad). A complete list of Gemmini user library functionality can be found in Exo's GitHub repository (https://github.com/exo-lang/exo).

3.15.1 User-defined Gemmini scratchpad memory

```
class GEMM_SCRATCH(Memory):
    @classmethod
    def global_(cls):
        _here_ = os.path.dirname(os.path.abspath(__file__))
        return _configure_file(Path(_here_) / 'gemm_malloc.c',
                               heap_size=100000,
                               dim=16)

    @classmethod
    def alloc(cls, new_name, prim_type, shape, srcinfo):
        if len(shape) == 0:
            return f"{prim_type} {new_name};"

        size_str = shape[0]
        for s in shape[1:]:
            size_str = f"{s} * {size_str}"
        if not _is_const_size(shape[-1], 16):
            raise MemGenError(f"{srcinfo}: "
                              "Cannot allocate GEMMINI Scratchpad Memory "
                              "unless innermost dimension is exactly 16.  "
                              f"got {shape[-1]}")
        return (f"{prim_type} *{new_name} = "
                f"({prim_type}*) ((uint64_t)gemm_malloc ({size_str} *
                ↪  sizeof({prim_type})));")

    @classmethod
    def free(cls, new_name, prim_type, shape, srcinfo):
        if len(shape) == 0:
            return ""

        return f"gemm_free((uint64_t)({new_name}));"

    @classmethod
```

```
    def window(cls, basetyp, baseptr, indices, strides, srcinfo):
        # assume that strides[-1] == 1
        #    and that strides[-2] == 16 (if there is a strides[-2])
        assert len(indices) == len(strides) and len(strides) >= 2
        prim_type = basetyp.basetype().ctype()
        offset = " + ".join(
            [f"({i}) * ({s})" for i, s in zip(indices, strides)])
        return (f"*({prim_type}*)((uint64_t)( "
                f"((uint32_t)((uint64_t){baseptr})) + "
                f"({offset})/16))")
```

3.15.2 User-defined Gemmini accumulator memory

```
class GEMM_ACCUM(Memory):
    @classmethod
    def global_(cls):
        _here_ = os.path.dirname(os.path.abspath(__file__))
        return _configure_file(Path(_here_) / 'gemm_acc_malloc.c',
                               heap_size=100000,
                               dim=16)

    @classmethod
    def alloc(cls, new_name, prim_type, shape, srcinfo):
        if len(shape) == 0:
            return f"{prim_type} {new_name};"

        size_str = shape[0]
        for s in shape[1:]:
            size_str = f"{s} * {size_str}"
        if not _is_const_size(shape[-1], 16):
            raise MemGenError(f"{srcinfo}: "
                              "Cannot allocate GEMMINI Accumulator Memory "
                              "unless innermost dimension is exactly 16.  "
                              f"got {shape[-1]}")
        return (f"{prim_type} *{new_name} = "
                f"({prim_type}*) ((uint32_t)gemm_acc_malloc ({size_str} *
                ↪  sizeof({prim_type})));")

    @classmethod
    def free(cls, new_name, prim_type, shape, srcinfo):
        if len(shape) == 0:
            return ""
        return f"gemm_acc_free((uint32_t)({new_name}));"

    @classmethod
    def window(cls, basetyp, baseptr, indices, strides, srcinfo):
        # assume that strides[-1] == 1
        #    and that strides[-2] == 16 (if there is a strides[-2])
        assert len(indices) == len(strides) and len(strides) >= 2
        prim_type = basetyp.basetype().ctype()
```

```
        offset = " + ".join([f"({i}) * ({s})"
                             for i, s in zip(indices, strides)])
        return (f"*({prim_type}*)((uint64_t)( "
                f"((uint32_t)((uint64_t){baseptr})) + "
                f"({offset})/16))")
```

3.15.3 User-defined Gemmini load instruction

```
_gemm_ld_i8   = ("gemmini_extended3_config_ld({src}.strides[0]*1, "+
                 "1.0f, 0, 0);\n"+
                 "gemmini_extended_mvin( &{src_data}, "+
                              "((uint64_t) &{dst_data}), {m}, {n} );")
@instr(_gemm_ld_i8)
def ld_i8(
    n     : size,
    m     : size,
    src   : [i8][n, m] @ DRAM,
    dst   : [i8][n, 16] @ GEMM_SCRATCH,
):
    assert n <= 16
    assert m <= 16
    assert stride(src, 1) == 1
    assert stride(dst, 0) == 16
    assert stride(dst, 1) == 1

    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

3.15.4 User-defined Gemmini load configuration

`@config` decorates a Python class which is parsed and compiled as an Exo configuration object. Below is a configuration object for Gemmini load which has a stride parameter.

```
@config
class ConfigLoad:
    src_stride : stride
```

Below is the instruction procedure with Gemmini instruction string for setting the load configuration defined above. This instruction sets `ConfigLoad.src_stride` to the `stride` argument, thus changing the hardware parameter state.

```
_gemm_config_ld_i8   = ("gemmini_extended3_config_ld({src_stride}, "+
                        "1.0f, 0, 0);\n")
@instr(_gemm_config_ld_i8)
def config_ld_i8(
    src_stride : stride
):
    ConfigLoad.src_stride = src_stride
```

Chapter 4

Perceus: Precise Reference Counting with Reuse and Specialization

This chapter is based on the work in Reinking, Xie, Moura, and Leijen [135], which was a distinguished paper at PLDI '21.

4.1 Introduction

Reference counting [29], with its low memory overhead and ease of implementation, used to be a popular technique for automatic memory management. However, the field has broadly moved in favor of generational tracing collectors [107], partly due to various limitations of reference counting, including cycle collection, multi-threaded operations, and expensive in-place updates.

In this work we take a fresh look at reference counting. We consider a programming language design that gives strong compile-time guarantees in order to enable efficient reference counting at run-time. In particular, we build on the pioneering reference counting work in the Lean theorem prover [153], but we view it through the lens of language design, rather than purely as an implementation technique.

We demonstrate our approach in the Koka language [94, 96]: a functional language with mostly immutable data types together with a strong type and effect system. In contrast to the dependently typed Lean language, Koka is general-purpose, with support for exceptions, side effects, and mutable references via general algebraic effects and handlers [125, 126]. Using recent work on evidence translation [166, 167, 168], all these control effects are compiled into an internal core language with explicit control flow. Starting from this functional core, we can statically transform the code to enable efficient reference counting at runtime. In particular:

- Due to explicit control flow, the compiler can emit *precise* reference counting instructions where a (non-cyclic) reference is dropped as soon as possible. We call this *garbage free* reference counting as only live data is retained (§4.2.2).

- We show that precise reference counting enables many optimizations, in particular *drop specialization* which removes many reference count operations in the fast path (§4.2.3), *reuse analysis* which updates (immutable) data in-place when possible (§4.2.4), and *reuse specialization* which removes many in-place field updates (§4.2.5). The reuse analysis shows the benefit of a holistic approach: even though the surface language has immutable data types with strong guarantees, we can use dynamic run-time information, e.g. whether a reference is unique, to update in-place when possible.

- The in-place update optimization is guaranteed, which leads to a new programming paradigm that we call *FBIP: functional but in-place* (§4.2.6). Just like tail-call optimization lets us write loops with regular function calls, reuse analysis lets us write in-place mutating algorithms in a purely functional way. We showcase this approach by implementing a functional version of in-order Morris tree traversal [113], which is stack-less, using in-place tree node mutation via FBIP.

- We present a formalization of general reference counting using a novel linear resource calculus, λ^1, which is closely based on linear logic (§4.3), and we prove that reference counting is sound for any program in the linear resource calculus. We then present the *Perceus* algorithm as a deterministic syntax-directed version of λ^1, and prove that it is both *sound* (i.e. never drops a live reference), and *garbage free* (i.e. only retains reachable references).

- We demonstrate Perceus by providing a full implementation for the strongly typed functional language Koka [88]. The implementation supports typed algebraic effect handlers using evidence translation [167] and compiles into standard C11 code. The use of reference counting means no runtime system is needed and Koka programs can readily link with other C/C++ libraries.

- We show evidence that Perceus, as implemented for Koka, competes with other state-of-the-art memory collectors (§4.4). We compare our implementation in allocation intensive benchmarks against OCaml, Haskell, Swift, and Java, and for some benchmarks to C++ as well. Even though the current Koka compiler does not have many optimizations (besides the ones for reference counting), it has outstanding performance compared to these mature systems. As a highlight, on the tree insertion benchmark, the purely functional Koka implementation is within 10% of the performance of the in-place mutating algorithm in C++ (using `std::map` [49]).

Even though we focus on Koka in this chapter, we believe that Perceus, and the FBIP programming paradigm we identify, are both broadly applicable to other programming languages with similar static guarantees for explicit control flow. There is an accompanying technical report [134] containing all the proofs and further benchmark results.

4.2 Overview

Compared to a generational tracing collector, reference counting has low memory overhead and is straightforward to implement. However, while the cost of tracing collectors is linear in the live data, the cost of reference counting is linear in the number of reference counting operations. Optimizing the total cost of reference counting operations is therefore our main priority. There are at least three known problems that make reference counting operations expensive in practice and generally inferior to tracing collectors:

- *Concurrency*: when multiple threads share a data structure, reference count operations need to be atomic, which is expensive.
- *Precision*: common reference counted systems are not *precise* and hold on to objects too long. This increases memory usage and prevents aggressive optimization of many reference count operations.
- *Cycles*: if object references form a cycle, the runtime needs to handle them separately, which re-introduces many of the drawbacks of a tracing collector.

We handle each of these issues in the context of an eager, functional language using immutable data types together with a strong type and effect system. For concurrency, we precisely track when objects can become thread-shared (§4.2.7). For precision, we introduce Perceus, our algorithm for inserting precise reference counting operations that can be aggressively optimized. In particular, we eliminate and fuse many reference count operations with *drop specialization* (§4.2.3), turn functional matching into in-place updates with *reuse analysis* (§4.2.4), and minimize field updates with *reuse specialization* (§4.2.5).

Finally, although we currently do not supply a cycle collector, our design has two mitigations that reduces the occurrences of cycles in the first place. First, *(co)inductive* data types and eager evaluation prevent cycles outside of explicit mutable references, and it is statically known where cycles can possibly be introduced in the code (§4.2.7). Second, being a mostly functional language, mutable references are not often used – moreover, reuse analysis greatly reduces the need for them since in-place mutation is typically inferred.

The reference count optimizations are our main contribution and we start with a detailed overview in the following sections, ending with details about how we mitigate the impact of concurrency and cycles.

4.2.1 Types and Effects

We start with a brief introduction to Koka [94, 96] – a strongly typed, functional language that tracks all (side) effects. For example, we can define a squaring function as:

```
fun square( x : int ) : total int { x * x }
```

Here we see two types in the result: the effect type `total` and the result type `int`. The `total` type signifies that the function can be modeled semantically as a mathematically *total* function, which always terminates without raising an exception (or having any other observable side effect). Effectful functions get more interesting effect types, like:

```
fun println( s : string ) : console ()
fun divide( x : int, y : int ) : exn int
```

where `println` has a `console` effect and `divide` may raise an exception (`exn` when dividing by zero. It is beyond the scope of this chapter to go into full detail, but a novel feature of Koka is that it supports *typed algebraic effect handlers* which can define new effects like async/await, iterators, or co-routines without needing to extend the language itself [93, 95, 96].

Koka uses algebraic data types extensively. For example, we can define a polymorphic list of elements of type `a` as:

```
type list<a> {
  Cons( head : a, tail : list<a> )
  Nil
}
```

We can match on a list to define a polymorphic `map` function that applies a function `f` to each element of a list `xs`:

```
fun map( xs : list<a>, f : a -> e b ) : e list<b> {
  match(xs) {
    Cons(x,xx) -> Cons(f(x), map(xx, f))
    Nil        -> Nil
  }
}
```

Here we transform the list of generic elements of type `a` to a list of generic elements of type `b`. Since `map` itself has no intrinsic effect, the overall effect of `map` is polymorphic, and equals the effect `e` of the function `f` as it is applied to every element. The `map` function demonstrates many interesting aspects of reference counting and we use it as a running example in the following sections.

4.2.2 Precise Reference Counting

An important attribute that sets Perceus apart is that it is *precise*: an object is freed as soon as no more references remain. By contrast, common reference counting implementations tie the liveness of a reference to its lexical scope, which might retain memory longer than needed. Consider:

```
fun foo() {
  val xs = list(1,1000000) // create large list
  val ys = map(xs, inc) // increment elements
```

```
fun map( xs : list<a>, f : a -> e b ) : e list<b> {
  match(xs) {
    Cons(x,xx) -> Cons(f(x), map(xx, f))
    Nil        -> Nil
  }
}
```

(a) A polymorphic map function

(a)
↓
(b) ⟶ (c) ⟶ (d)

(e) ⟶ (f) ⟶ (g)

```
fun map ( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs)
      Cons( dup(f)(x), map(xx, f) )
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(b) dup/drop insertion (§4.2.2)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx)
      if (is-unique(xs))
        then drop(x); drop(xx);
             free(xs)
        else decref(xs)
      Cons(dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(c) drop specialization (§4.2.3)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      if (is-unique(xs))
        then free(xs)
        else dup(x); dup(xx);
             decref(xs)
      Cons(dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(d) push down dup and fuse (§4.2.3)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs)
      Cons@ru(dup(f)(x),
              map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(e) reuse token insertion (§4.2.4)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = \
        if (is-unique(xs))
          then drop(x); drop(xx); &xs
          else decref(xs); NULL
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(f) drop-reuse specialization (§4.2.5)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x, xx) {
      val ru = \
        if (is-unique(xs))
          then &xs
          else dup(x); dup(xx);
               decref(xs); NULL
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(g) push down dup and fuse (§4.2.5)

Figure 4.1: Drop specialization and reuse analysis for map.

```
  print(ys)
}
```

Many compilers emit code similar to:

```
fun foo() {
  val xs = list(1,1000000)
  val ys = map(xs, inc)
  print(ys)
  drop(xs)
  drop(ys)
}
```

where we use a colored background for generated operations. The `drop(xs)` operation decrements the reference count of an object and, if it drops to zero, recursively drops all children of the object and frees its memory. These "scoped lifetime" reference counts are used by the C++ `shared_ptr<T>` (calling the destructor at the end of the scope), Rust's `Rc<T>` (using the `Drop` trait), and Nim (using a `finally` block to call `destroy`) [169]. It is not required by the semantics, but Swift typically emits code like this as well [50].

Implementing reference counting this way is straightforward and integrates well with exception handling where the drop operations are performed as part of stack unwinding. But from a performance perspective, the technique is not always optimal: in the previous example, the large list `xs` is retained in memory while a new list `ys` is built. Both exist for the duration of `print`, after which a long, cascading chain of drop operations happens for each element in each list.

Perceus takes a more aggressive approach where *ownership* of references is passed down into each function: now `map` is in charge of freeing `xs`, and `ys` is freed by `print`: no `drop` operations are emitted inside `foo` as all local variables are *consumed* by other functions, while the `map` and `print` functions drop the list elements as they go. In this example, Perceus generates the code for `map` as given in fig. 4.1b. In the *`Cons`* branch, first the head and tail of the list are *dupped*, where a `dup(x)` operation increments the reference count of an object and returns itself. The `drop(xs)` then frees the initial list node. We need to `dup` `f` as well as it is used twice, while `x` and `xx` are consumed by `f` and `map` respectively.

At first blush, this seems more expensive than the scoped approach but, as we will see, this change enables many further optimizations. More importantly, transferring ownership, rather than retaining it, means we can free an object immediately when no more references remain. This both increases cache locality and decreases memory usage. For `map`, the memory usage is halved: the list `xs` is deallocated while the new list `ys` is being allocated.

4.2.3 Drop Specialization

Once we change to precise, ownership-based reference counting, there are many further optimization opportunities. After the initial insertion of `dup` and `drop` operations, we perform a *drop specialization* pass. The basic `drop` operation is defined in pseudocode as:

```
fun drop( x ) {
  if (is-unique(x)) then drop children of x; free(x)
                    else decref(x)
}
```

and drop specialization essentially inlines the `drop` operation specialized at a specific constructor. Figure 4.1c shows the drop specialization of our `map` example. Note that we only apply drop specialization if the children are used, so no specialization takes place in the *`Nil`* branch.

Again, it appears we made things worse with extra operations in each branch, but we can perform another transformation where we push down `dup` operations into branches followed by standard *dup/drop fusion* where corresponding `dup`/`drop` pairs are removed. Figure 4.1d shows the code that is generated for our `map` example.

After this transformation, almost all reference count operations in the fast path are gone. In our example, every node in the list `xs` that we map over is unique (with a reference count of 1) and so the `if (is-unique(xs))` test always succeeds, thus immediately freeing the node without any further reference counting.

4.2.4 Reuse Analysis

There is more we can do. Instead of freeing and immediately allocating a fresh *Cons* node, we can try to *reuse* `xs` directly as first described by Ullrich and de Moura [153]. *Reuse analysis* is performed before emitting the initial reference counting operations. It analyses each `match` branch, and tries to pair each matched pattern to allocated constructors of the same size in the branch. In our `map` example, `xs` is paired with the *Cons* constructor. When such pairs are found, and the matched object is not live, we generate a `drop-reuse` operation that returns a *reuse token* that we attach to any constructor paired with it:

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      val ru = drop-reuse(xs)
      Cons@ru( f(x), map(xx, f) )
    }
    Nil -> Nil
  }
}
```

The *Cons*`@ru` annotation means that (at runtime) if `ru==NULL` then the *Cons* node is allocated fresh, and otherwise the memory at `ru` is of the right size and can be used directly. Figure 4.1e shows the generated code after reference count insertion. Compared to the program in fig. 4.1b, the generated code now consumes `xs` using `drop-reuse(xs)` instead of `drop(xs)`.

Just like with *drop specialization* we can also specialize `drop-reuse`. The `drop-reuse` operation is specified in pseudocode as:

```
fun drop-reuse ( x ) {
  if (is-unique(x)) then drop children of x; &x
                    else decref(x); NULL
}
```

where `&x` returns the address of `x`. Figure 4.1f shows the code for `map` after specializing the `drop-reuse`. Again, we can push down and fuse the `dup` operations, which finally results in the code shown in fig. 4.1g. In the fast path, where `xs` is uniquely owned, there are no more reference counting operations at all! Furthermore, the memory of `xs` is directly reused to provide the memory for the *Cons* node for the returned list – effectively updating the list *in-place*.

4.2.5 Reuse Specialization

The final transformation we apply is *reuse specialization*, by which we can further reuse unchanged fields of a constructor. A constructor expression like `Cons@ru(x,xx)` is implemented in pseudocode as:

```
fun Cons@ru( x, xx ) {
  if (ru!=NULL)
    then { ru->head := x; ru->tail := xx; ru } // in-place
    else Cons(x,xx)                           // malloc'd
}
```

However, for our `map` example there would be no benefit to specializing as all fields are assigned. Thus, we only specialize constructors if at least one of the fields stays the same. As an example, we consider insertion into a red-black tree [58]. We define red-black trees as:

```
type color { Red; Black }
type tree {
  Leaf
  Node( color: color, left: tree, key: int,
        value: bool, right: tree )
}
```

The red-black tree has the invariant that the number of black nodes from the root to any of the leaves is the same, and that a red node is never a parent of red node. Together this ensures that the trees are always balanced. When inserting nodes, the invariants need to be maintained by rebalancing the nodes when needed. Okasaki's algorithm [119] implements this elegantly and functionally (the full algorithm can be found in the accompanying technical report [134]):

```
fun bal-left( l : tree, k : int, v : bool, r : tree ): tree {
  match(l) {
  Node(_, Node(Red, lx, kx, vx, rx), ky, vy, ry)
    -> Node(Red, Node(Black, lx, kx, vx, rx), ky, vy,
                 Node(Black, ry, k, v, r))
  ...
}
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) // second branch
      -> if (k < kx) then Node(Red, ins(l, k, v), kx, vx, r)
         ...
    Node(Black, l, kx, vx, r)
      -> if (k < kx && is-red(l))
           then bal-left(ins(l,k,v), kx, vx, r)
         ...
  }
}
```

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left; // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Figure 4.2: Morris in-order tree traversal algorithm in C.

For this kind of program, reuse specialization is effective. For example, if we look at the second branch in `ins` we see that the newly allocated *`Node`* has almost all of the same fields as `t` except for the left tree `l` which becomes `ins(l,k,v)`. After reuse specialization, this branch becomes:

```
Node(Red, l, kx, vx, r) { // second branch
  val ru = if (is-unique(t))
           then &t
           else { dup(l); dup(kx); dup(vx); dup(r); NULL }
  if (dup(k) < dup(kx)) {
    val y = ins(l,k,v)
    if (ru!=NULL) then { ru->left := y; ru } // fast path
                  else Node(Red, y, kx, vx, r)
  }
```

In the fast path, where `t` is uniquely owned, `t` is reused directly, and only its left child is re-assigned as all other fields stay unchanged. This applies to many branches in this example and saves many assignments.

Moreover, the compiler inlines the `bal-left` function. At that point, every matched *`Node`* constructor has a corresponding *`Node`* allocation – if we consider all branches we can see that we either match one *`Node`* and allocate one, or we match three nodes deep and allocate three.

With *reuse analysis* this means that every `Node` is reused in the fast path without doing any allocations!

Essentially this means that for a unique tree, the purely functional algorithm above adapts at runtime to an in-place mutating re-balancing algorithm (without any further allocation). Moreover, if we use the tree *persistently* [118], and the tree is shared or has shared parts, the algorithm adapts to copying exactly the shared *spine* of the tree (and no more), while still rebalancing in place for any unshared parts.

4.2.6 A New Paradigm: Functional but In-Place (FBIP)

The previous red-black tree rebalancing showed that with Perceus we can write algorithms that dynamically adapt to use in-place mutation when possible (and use copying when used persistently). Importantly, a programmer can rely on this optimization happening, e.g. they can see the `match` patterns and match them to constructors in each branch.

This style of programming leads to a new paradigm that we call FBIP: "functional but in place". Just like tail-call optimization lets us describe loops in terms of regular function calls, reuse analysis lets us describe in-place mutating imperative algorithms in a purely functional way (and get persistence as well). Consider mapping a function `f` over all elements in a binary tree in-order:

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}
fun tmap( t : tree, f : int -> int ) : tree {
  match(t) {
    Bin(l,x,r) -> Bin( tmap(l,f), f(x), tmap(r,f) )
    Tip        -> Tip
  }
}
```

This is already quite efficient as all the `Bin` and `Tip` nodes are reused in-place when `t` is unique. However, the `tmap` function is not tail-recursive and thus uses as much stack space as the depth of the tree.

In 1968, Knuth posed the problem of visiting a tree in-order while using no extra stack- or heap space [87] (For readers not familiar with the problem it might be fun to try this in your favorite imperative language first and see that it is not easy to do). Since then, numerous solutions have appeared in the literature. A particularly elegant solution was proposed by Morris [113]. This is an in-place mutating algorithm that swaps pointers in the tree to "remember" which parts are unvisited. It is beyond this chapter to give a full explanation, but a C implementation is shown in fig. 4.2. The traversal essentially uses a *right-threaded* tree to keep track of which nodes to visit. The algorithm is subtle, though. Since it transforms the tree into an intermediate graph, we need to state invariants over the so-called *Morris loops* [104] to prove its correctness.

```
type visitor {
  Done
  BinR(  right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) { // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip -> tmap(f,Tip,visit,Up) // B
    }
    Up -> match(visit) { // go up through the visitor
      Done -> t // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up) // E
} } }
```

Figure 4.3: FBIP in-order tree traversal algorithm in Koka.

We can derive a functional and more intuitive solution using the FBIP technique. We start by defining an explicit *visitor* data structure that keeps track of which parts of the tree we still need to visit. In Koka we define this data type as `visitor` given in fig. 4.3. (Interestingly, our visitor data type can be generically derived as a list of the derivative of the tree data type[1] [79, 106]). We also keep track of which `direction` we are going, either *Up* or *Down* the tree.

We start our traversal by going downward into the tree with an empty visitor, expressed as `tmap(f, t, Done, Down)`. The key idea is that we are either *Done* (*C*), or, on going downward in a left spine we remember all the right trees we still need to visit in a *BinR* (*A*) or, going upward again (*B*), we remember the left tree that we just constructed as a *BinL* while visiting right trees (*D*). When we come back (*E*), we restore the original tree with the result values. Note that we apply the function `f` to the saved value in branch *D* (as we visit *in-order*), but the functional implementation makes it easy to specify a *pre-order* traversal by applying `f` in branch *A*, or a *post-order* traversal by applying `f` in branch *E*.

Looking at each branch we can see that each *Bin* matches up with a *BinR*, each *BinR* with a *BinL*, and finally each *BinL* with a *Bin*. Since they all have the same size, if the tree is unique, each branch updates the tree nodes *in-place* at runtime without any allocation, where the `visitor` structure is effectively overlaid over the tree nodes while traversing the tree. Since all `tmap` calls are tail calls, this also compiles to a loop and thus needs no extra

[1] Conor McBride [106] describes how we can generically derive a *zipper* [79] visitor for any recursive type $\mu x.\ \mathsf{F}$ as a list of the derivative of that type, namely $\mathsf{list}\ (\frac{\partial}{\partial x}\ \mathsf{F}\ |_{x\ =\mu x.\mathsf{F}})$. In our case, calculating the derivative of the inductive `tree`, we get $\mu x.\ 1 + (\mathit{tree} \times \mathit{int} \times x) + (\mathit{tree} \times \mathit{int} \times x)$, which corresponds to the `visitor` datatype.

stack or heap space.

Finally, just like with re-balancing tree insertion, the algorithm as specified is still purely functional: it uses in-place updating when a unique tree is passed, but it also adapts gracefully to the persistent case where the input tree is shared, or where parts of the input tree are shared, making a single copy of those parts of the tree.

4.2.7 Static Guarantees and Language Features

So far we have shown that precise reference counting enables powerful analyses and optimizations of the reference counting operations. In this section, we use Koka as an example to discuss how strong static guarantees at compile-time can further allow the precise reference counting approach to be integrated with non-trivial language features.

Non-Linear Control Flow

An essential requirement of our approach is that programs have explicit control flow so that it is possible to statically determine where to insert `dup` and `drop` operations. However, it is in tension with functions that have non-linear control flow, e.g. may throw an exception, use a `longjmp`, or create an asynchronous continuation that is never resumed. For example, if we look at the code for `map` before applying optimizations, we have:

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs); dup(f)
      Cons( f(x). map(xx, f) )
    }
    ...
```

If `f` raised an exception and directly exited the scope of `map`, then `xx` and `f` would leak and never be dropped. This is one reason why a C++ `shared_ptr` is tied to lexical scope; it integrates nicely with the stack unwinding mechanism for exceptions that guarantees each `shared_ptr` is dropped eventually.

In Koka, we guarantee that all control-flow is compiled to explicit control-flow, so our reference count analysis does not have to take non-linear control-flow into account. This is achieved through *effect typing* (§4.2.1) where every function has an effect type that signifies if it can throw exceptions or not. Functions that can throw are compiled into functions that return with an explicit error type that is either *Ok*, or *Error* if an exception is thrown. This is checked and propagated at every invocation[2].

For example, for `map` the compiled code (before optimization) becomes like:

[2]Koka actually generalizes this using a multi-prompt delimited control monad that works for any control effect, with essentially the same principle.

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs); dup(f)
      match(f(x)) {
        Error(err) -> { drop(xx); drop(f); Error(err); }
        Ok(y) -> {
          match(map(xx, f)) {
            Error(err) -> drop(y); Error(err)
            Ok(ys) -> Cons(y,ys)
          }
    ...
```

At this point all errors are explicitly propagated and all control-flow is explicit again. Note the we have no reference count operations on the `error` values as these are implemented as *value* types which are not heap allocated.

This is similar to error handling in Swift [81] (although it requires the programmer to insert a `try` at every invocation), and also similar to various C++ proposals [145] where exceptions become explicit error values.

The example here is specialized for exceptions but the actual Koka implementation uses a generalized version of this technique to implement a multi-prompt delimited control monad [60] instead, which is used in combination with evidence translation [167] to express general algebraic effect handlers (which in turn subsume all other control effects, like exceptions, async/await, probabilistic programming, etc).

Concurrent Execution

If multiple threads share a reference to a value, the reference count needs to be incremented and decremented using atomic operations which can be expensive. Ungar et al. [154] report slowdowns up to 50% when atomic reference counting operations are used. Nevertheless, in languages with unrestricted multi-threading, like Swift, almost all reference count operations need to assume that references are potentially thread-shared.

In Koka, the strong type system gives us additional guarantees about which variables may need atomic reference count operations. Following the solution of Ullrich and de Moura [153], we mark each object with whether it can be thread-shared or not, and supply an internal polymorphic operation `tshare : forall a. a -> io ()` which marks any object and its children recursively as being thread-shared. Even though marking is linear, it happens at most once for any object since shared objects cannot be unshared. All objects start out as unshared, and are only marked through explicit operations. In particular, when starting a new thread, the argument passed to the thread is marked as thread-shared. The only other operation that can cause thread sharing is setting a thread-shared mutable reference but this is quite uncommon in typical Koka code. The `drop` and `dup` operations can be implemented efficiently by avoiding atomic operations in the fast path by checking the thread-shared flag.

For example, `drop` may be implemented in C as:

```
static inline void drop( block_t* b ) {
  if (b->header.thread_shared) {
    if (atomic_dec(&b->header.rc) == 1) drop_free(b);
  } else if (b->header.rc-- == 1) drop_free(b);
}
```

However, this may still present quite some overhead as many `drop` operations are emitted.

In Koka we encode the reference count for thread-shared objects as a negative value. This enables us to use a *single* inlined test to see if we need to take the slow path for either a thread-shared object or an object that needs to be freed; and we can use a fast inlined path for the common case[3]:

```
static inline void drop( block_t* b ) {
  if (b->header.rc <= 1) { drop_check(b); }  // slow path
                    else { b->header.rc--; }
}
```

The `drop_check` function checks if the reference count is 1 to release it, or otherwise it adjusts the reference count atomically. We also use the negative values to implement a *sticky* range where very large reference counts (2^{30} in our implementation) stay without being further adjusted (preventing overflow, and keeping them alive for the rest of the program).

Mutation

Mutation in Koka is done through explicit mutable references. Here we look at first-class mutable reference cells, but Koka also has second-class mutable local variables that can be more convenient. A mutable reference cell is created with `ref`, dereferenced with `(!)` and updated using `(:=)`:

```
fun ref( init : a )            : st<h> ref<h,a>
fun (!)( r : ref<h,a> )        : st<h> a
fun (:=)( r : ref<h,a>, x : a ) : st<h> ()
```

where each operation has a stateful effect `st<h>` in some heap `h`. A reference cell of type `ref<h,a>` is a first-class *value* that contains a reference to a value of type `a`. As such, there are always two reference counts involved: that of the reference itself, and that of value that is referenced.

When a mutable reference cell is thread-shared, this presents a problem as an update operation may *race* with a read operation to update the reference counts. The pseudocode implementation of both operations is:

```
fun (!)( r ) {             fun (:=)( r, x ) {
  val x = r->value           val y = r->value
  dup(x)                     r->value := x
  x                          drop(y)
}                          }
```

[3]Since the thread-shared sign-bit is *stable*, we can do the test `b->header.cr <= 1` without needing expensive atomic operations and can use a `memory_order_relaxed` atomic read.

The read operation `(!)` first reads the current reference in `x`, and then increments its reference count. Suppose though that before the `dup`, the thread is suspended and another thread writes to the same reference: it will read the same object into `y`, update the reference, and then drop `y` – and if `y` has a reference count of 1 it will be freed! When the other thread resumes, it will now try to `dup` the just-freed object.

To make this work correctly, we need to perform both operations atomically, either through a double-CAS [33], using hazard pointers [52, 109], or using some other locking mechanism. Either way, this can be quite expensive. Fortunately, in our setting, we can avoid the slow path in most cases. First of all, since FBIP allows for the efficiency of in-place updates with a purely functional specification (§4.2.6), we expect mutable references to be a last resort rather than the default. Secondly, as discussed in §4.2.7, we can also check if a mutable reference is actually thread-shared and thus avoid the atomic code path almost all of the time.

Cycles

A known limitation of reference counting is that it cannot release cyclic data structures. Just like with mutability, we try to mitigate its performance impact by reducing the potential for this to occur in the first place. In Koka, almost all data types are immutable and either *inductive* or *coinductive*. It can be shown that such data types are never cyclic (and functions that recurse over such data types always terminate).

In practice, mutable references are the main way to construct cyclic data. Since mutable references are uncommon in our setting, we leave the responsibility to the programmer to break cycles by explicitly clearing a reference cell that may be part of a cycle. Since this strategy is also used by Swift, a widely used language where most object fields are mutable, we believe this is a reasonable approach to take for now. However, we have plans for future improvements: since we know statically that only mutable references are able to form a cycle, we could generate code that tracks those data types at run time and may perform a more efficient form of incremental cycle collection.

Summary

In summary, we have shown how static guarantees at compile-time can be used to mitigate the performance impact of concurrency and the risk of cycles. This work does not yet present a general solution to all problems with reference counting and future work is required to explore how cycles can be handled more efficiently, and how well Perceus can be used with implicit control flow. Yet, we expect that our approach gives new insights in the general design space of reference counting, and showcase that precise reference counting can be a viable alternative to other approaches. In practice, we found that Perceus has good performance, which is discussed in §4.4.

Expressions

e	$::=$	$v \mid e\ e$	(value, application)
	$\mid$	$\mathsf{val}\ x = e;\ e$	(bind)
	$\mid$	$\mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}$	(match)
	$\mid$	$\mathsf{dup}\ x;\ e$	(duplicate)
	$\mid$	$\mathsf{drop}\ x;\ e$	(drop)
	$\mid$	$\mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}$	(match expr)
v	$::=$	$x \mid \lambda x.\ e$	(variables, functions)
	$\mid$	$C\ v_1 \ldots v_n$	(constructor of arity n)
p	$::=$	$C\ b_1 \ldots b_n$	(pattern)
b	$::=$	$x \mid \_$	(binder or wildcard)

Contexts $\Delta, \Gamma ::= \varnothing \mid \Delta \cup x$

Syntactic shorthands

$e_1;\ e_2$	$\triangleq$	$\mathsf{val}\ x = e_1;\ e_2$	sequence, $x \notin \mathsf{fv}(e_2)$
$\lambda\_.\ e$	$\triangleq$	$\lambda x.\ e$	$x \notin \mathsf{fv}(e)$
$\lambda x.\ e$	$\triangleq$	$\lambda^{ys} x.\ e$	$ys = \mathsf{fv}(e)$

Figure 4.4: Syntax of the linear resource calculus λ^1.

4.3 A Linear Resource Calculus

In this section we present a novel linear resource calculus, λ^1, which is closely based on linear logic. The operational semantics of λ^1 is formalized in an explicit heap with reference counting, and we prove that the operational semantics is sound. We then formalize Perceus as a sound and precise syntax-directed algorithm of λ^1 and thus provide a theoretic foundation for Perceus.

4.3.1 Syntax

Figure 4.4 defines the syntax of our linear resource calculus λ^1. It is essentially an untyped lambda calculus extended with explicit binding as $\mathsf{val}\ x = e_1; e_2$, and pattern matching as match. We assume all patterns in the match are mutually exclusive, and all pattern binders are distinct. Syntactic constructs in gray are only generated in derivations of the calculus and are not exposed to users. Among those constructs, dup and drop form the basic instructions of reference counting.

Contexts Δ, Γ are *multisets* containing variable names. We use the compact comma notation for summing (or splitting) multisets. For example, (Γ, x) adds x to Γ, and (Γ_1, Γ_2) appends two multisets Γ_1 and Γ_2. The set of free variables of an expression e is denoted by $\mathsf{fv}(e)$, and the set of bound variables of a pattern p by $\mathsf{bv}(p)$.

$$\boxed{\underset{\uparrow}{\Delta} \mid \underset{\uparrow}{\Gamma} \vdash \underset{\uparrow}{e} \;\rightsquigarrow\; \underset{\downarrow}{e'} \qquad (_{\uparrow} \text{ is input, while }_{\downarrow} \text{ is output})}$$

$$\frac{}{\Delta \mid x \vdash x \;\rightsquigarrow\; x}\;[\textsc{var}]$$

$$\frac{\Delta \mid \Gamma, x \vdash e \;\rightsquigarrow\; e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \;\rightsquigarrow\; \mathsf{dup}\ x; e'}\;[\textsc{dup}]$$

$$\frac{\Delta \mid \Gamma \vdash e \;\rightsquigarrow\; e'}{\Delta \mid \Gamma, x \vdash e \;\rightsquigarrow\; \mathsf{drop}\ x;\ e'}\;[\textsc{drop}]$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \;\rightsquigarrow\; e_1' \qquad \Delta \mid \Gamma_2 \vdash e_2 \;\rightsquigarrow\; e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\ e_2 \;\rightsquigarrow\; e_1'\ e_2'}\;[\textsc{app}]$$

$$\frac{\varnothing \mid \Gamma, x \vdash e \;\rightsquigarrow\; e' \qquad \Gamma \;=\; \mathsf{fv}(\lambda x.\ e)}{\Delta \mid \Gamma \vdash \lambda x.\ e \;\rightsquigarrow\; \lambda^{\Gamma} x.\ e'}\;[\textsc{lam}]$$

$$\frac{x \;\notin \Delta, \Gamma_1, \Gamma_2 \qquad \Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \;\rightsquigarrow\; e_1' \qquad \Delta \mid \Gamma_2, x \vdash e_2 \;\rightsquigarrow\; e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{val}\ x \;=\; e_1;\ e_2 \;\rightsquigarrow\; \mathsf{val}\ x \;=\; e_1';\ e_2'}\;[\textsc{bind}]$$

$$\frac{\Delta \mid \Gamma, \mathsf{bv}(p_i) \vdash e_i \;\rightsquigarrow\; e_i'}{\Delta \mid \Gamma, x \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \;\rightsquigarrow\; \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i'}\ \}}\;[\textsc{match}]$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash v_i \;\rightsquigarrow\; v_i' \qquad 1 \leq i \leq n}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash C\ v_1\ \ldots\ v_n \;\rightsquigarrow\; C\ v_1'\ \ldots\ v_n'}\;[\textsc{con}]$$

Figure 4.5: Declarative linear resource rules of λ^1.

$$\begin{array}{lll} \mathsf{E} & ::= & \square \mid \mathsf{E}\ e \mid v\ \mathsf{E} \\ & \mid & \mathsf{val}\ x = \mathsf{E}; e \end{array} \qquad \frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\;[\textsc{eval}]$$

$$\begin{array}{llll} (app) & (\lambda x.\ e)\ v & \longrightarrow & e[x := v] \\ (bind) & \mathsf{val}\ x \;=\; v\ ;\ e & \longrightarrow & e[x := v] \\ (match) & \mathsf{match}\ (C\ v_1 \ldots v_n)\ \{\ \overline{p_i \to e_i}\ \} & & \\ & & \longrightarrow & e_i[x_1 := v_1, \ldots, x_n := v_n] \\ & & & \text{with } p_i = C\ x_1 \ldots x_n \end{array}$$

Figure 4.6: Standard strict semantics for λ^1.

4.3.2 The Linear Resource Calculus

The derivation $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ in fig. 4.5 reads as follows: given a *borrowed environment* Δ, a *linear environment* Γ, an expression e is translated into an expression e' with explicit reference counting instructions. We call variables in the linear environment *owned*.

The key idea of λ^1 is that each resource (i.e., owned variable) is consumed *exactly* once. That is, a resource needs to be explicitly duplicated (in rule [DUP]) if it is needed more than once; or be explicitly dropped (in rule [DROP]) if it is not needed. The rules are closely related to linear typing.

Following the key idea, the variable rule [VAR] consumes a resource when we own and only own x exactly once in the owned environment. For example, to derive the K combinator, $\lambda x\ y.\ x$, we need to apply [DROP] to be able to discard y, which gives $\lambda x\ y.\ \mathsf{drop}\ y; x$.

The [APP] rule splits the owned environment Γ into two separate contexts Γ_1 and Γ_2 for expression e_1 and e_2 respectively. Each expression then consumes its corresponding owned environment. Since Γ_2 is consumed in the e_2 derivation, we know that resources in Γ_2 are surely alive when deriving e_1, and thus we can *borrow* Γ_2 in the e_1 derivation. The rule is quite similar to the [LET!] rule of Wadler's linear type rules [161, p. 14] where a linear type can be "borrowed" as a regular type during evaluation of a binding.

Borrowing is important as it allows us to conduct a dup as late as possible, or otherwise we will need to duplicate enough resources before we can divide the owned environment. Consider $\lambda f\ g\ x.\ (f\ x)\ (g\ x)$. Without borrowing, we have to duplicate x before the application, resulting in $\lambda f\ g\ x.\ \mathsf{dup}\ x; (f\ x)\ (g\ x)$. With the borrowing environment it is now possible to derive a translation with the dup right before passing x to f: $\lambda f\ g\ x.\ (f\ (\mathsf{dup}\ x; x))\ (g\ x))$. Notice rule [DUP] allows dup from the borrowing environment, where [DROP] only applies to the owned environment.

The [LAM] rule is interesting as it essentially derives the body of the lambda independently. The premise $\Gamma = \mathsf{fv}(\lambda x.e)$ requires that exactly the free variables in the lambda are owned – this corresponds to the notion that a lambda is allocated as a closure at runtime that holds all free variables of the lambda (and thus the lambda expression *consumes* the free variables). The body of a lambda is evaluated only when applied, so it is derived under an empty borrowed environment only owning the argument and the free variables (in the closure). The translated lambda is also annotated with Γ, as $\lambda^{\Gamma} x.\ e$, so we know precisely the resources the lambda should own when evaluated in a heap semantics. We often omit the annotation when it is irrelevant.

The [BIND] rule is similar to application and borrows Γ_2 in the derivation for the bound expression. This is the main reason to not consider $\mathsf{val}\ x = e_1; e_2$ as syntactic sugar for $(\lambda x.\ e_2)\ e_1$. The [MATCH] rule consumes the scrutinee and owns the bound variables in each pattern for each branch. For constructors (rule [CON]), we divide the owned environment into n parts for each component, and allow each component derivation to borrow the owned environment of the components derived later.

We use the notation $\lceil e \rceil$ to erase all drop and dup in the expression e. We can now state that derivations leave expressions unchanged except for inserting dup/drop operations: if

$$\begin{array}{lll} \mathsf{H} & : & x \to (\mathbb{N}^+, v) \\ \mathsf{E} & ::= & \square \mid \mathsf{E}\ e \mid x\ \mathsf{E} \mid \mathsf{val}\ x = \mathsf{E}; e \\ & \mid & C\ x_1 \ldots x_i\ \mathsf{E}\ v_j \ldots v_n \end{array} \qquad \frac{\mathsf{H} \mid e \longrightarrow_r \mathsf{H}' \mid e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_r \mathsf{H}' \mid \mathsf{E}[e']}\,[\textsc{eval}]$$

$$\begin{array}{llll} (lam_r) & \mathsf{H} \mid (\lambda^{ys}\ x.\ e) & \longrightarrow_r & \mathsf{H}, f \mapsto^1 \lambda^{ys} x.\ e \quad \mid f \quad \text{fresh } f \\ (con_r) & \mathsf{H} \mid C\ x_1 \ldots x_n & \longrightarrow_r & \mathsf{H}, z \mapsto^1 C\ x_1 \ldots x_n \quad \mid z \quad \text{fresh } z \\ \\ (app_r) & \mathsf{H} \mid f\ z & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys; \mathsf{drop}\ f; e[x := z] \\ & & & (f \mapsto^n \lambda^{ys} x.\ e) \ \in \mathsf{H} \\ (match_r) & \mathsf{H} \mid \mathsf{match}\ x\ \{\overline{p_i \to e_i}\} & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys; \mathsf{drop}\ x; e_i[xs := ys] \\ & & & \text{with } p_i = C\ xs \text{ and } (x \mapsto^n C\ ys) \in \mathsf{H} \\ (bind_r) & \mathsf{H} \mid \mathsf{val}\ x = y; e & \longrightarrow_r & \mathsf{H} \mid e[x := y] \end{array}$$

$$\begin{array}{llllll} (dup_r) & \mathsf{H}, x \mapsto^n v & \mid \mathsf{dup}\ x; e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n+1} v & \mid e \\ (drop_r) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{drop}\ x; e & \longrightarrow_r & \mathsf{H}, x \mapsto^n v & \mid e \quad \text{if } n \geqslant 1 \\ (dlam_r) & \mathsf{H}, x \mapsto^1 \lambda^{ys} z.e & \mid \mathsf{drop}\ x; e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys; e & \\ (dcon_r) & \mathsf{H}, x \mapsto^1 C\ ys & \mid \mathsf{drop}\ x; e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys; e & \end{array}$$

Figure 4.7: Reference-counted heap semantics for λ^1.

$\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ then $e \ = \ \lceil e' \rceil$.

4.3.3 Semantics

Figure 4.6 defines standard semantics for λ^1 using strict evaluation contexts [164]. The evaluation contexts uniquely determine where to apply an evaluation step. As such, evaluation contexts neatly abstract from the usual implementation context of a stack and program counter. Rule (*match*) relies on the internal form of expression $\mathsf{match}\ e\{\overline{p_i \to e_i}\}$: after substitution *app*, values may appear in positions where only variables were allowed, and this is exactly what enables us to do pattern match on a data constructor.

In fig. 4.7 we define our target semantics of a reference counted heap, so sharing of values becomes explicit and substitution only substitutes variables. Here, each heap entry $x \mapsto^n v$ points to a value v with a reference count of n (with $n \geqslant 1$). In these semantics, values other than variables are allocated in the heap with rule (lam_r) and rule (con_r). The evaluation rules discard entries from the heap when the reference count drops to zero. Any allocated lambda is annotated as $\lambda^{ys} x.\ e$ to clarify that these are essentially *closures* holding an environment ys and a code pointer $\lambda x.\ e$. Note that it is important that the environment ys is a multi-set. After the initial translation, ys will be equivalent to the free variables in the body (see rule [LAM]), but during evaluation substitution may substitute several variables with the same reference. To keep reference counts correct, we need to keep considering each one as a separate entry in the closure environment.

When applying an abstraction, rule (app_r) needs to satisfy the assumptions made when deriving the abstraction in rule [LAM]. First, the (app_r) rule inserts dup to duplicate variables ys, as these are owned in rule [LAM]. It then drops the reference to the closure itself. Rule $(match_r)$ is similar to rule (app_r), which duplicates the newly bound pattern bindings and drops the scrutinee[4]. Rule $(bind_r)$ simply substitutes the bound variable x with the resource y.

Duping a resource is straightforward as rule dup_r merely increments the reference count of the resource. Dropping is more involved. Rule $drop_r$ just decrements the reference count when there are still multiple copies of it. But when the reference count would drop to zero, rule $dlam_r$ and rule $dcon_r$ actually *free* a heap entry and then dynamically insert drop operations to drop their fields recursively.

The tricky part of the reference counting semantics is showing *correctness*. We prove this in two parts. First, we prove that the reference counting semantics is *sound* and corresponds to the standard semantics. Below we use heaps as substitutions on expressions. We write $[\mathsf{H}]e$ to mean H applied as a substitution to expression e.

Theorem 4.1 (Reference-counted heap semantics is sound)**.** *If we have* $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$ *and* $e \longmapsto^* v$*, then we also have* $\varnothing \mid e' \longmapsto_r^* \mathsf{H} \mid x$ *with* $[\mathsf{H}]x = v$*.*

To prove this theorem we need to maintain strong invariants at each evaluation step to ensure a variable is still alive if it is going to be referred-to later. Second, we prove that the reference counting semantics never *hold on* to unused variables. We first define the notion of *reachability*.

Definition 4.1 (Reachability)**.** We say a variable x is reachable in terms of a heap H and an expression e, denoted as $\mathsf{reach}(x, \mathsf{H} \mid e)$, if (1) $x \in \mathsf{fv}(e)$; or (2) for some y, we have $\mathsf{reach}(y, \mathsf{H} \mid e) \wedge y \mapsto^n v \in \mathsf{H} \wedge \mathsf{reach}(x, \mathsf{H} \mid v)$.

With reachability, we can formally show:

Theorem 4.2 (Reference counting leaves no garbage)**.** *Given* $\varnothing ; \varnothing \vdash e \rightsquigarrow e'$*, and* $\varnothing \mid e' \longmapsto_r^* \mathsf{H} \mid x$*, then for every intermediate state* $\mathsf{H}_i \mid e_i$*, we have for all* $y \in \mathsf{dom}(\mathsf{H}_i)$*,* $\mathsf{reach}(y, \mathsf{H}_i \mid e_i)$*.*

In the accompanying technical report [134], we further show that the reference counts are exactly equal to the number of actual references to the resource. Notably, to capture the essence of precise reference counting, λ^1 does not model *mutable references* (§4.2.7). From theorem 4.2 we see that mutable references are indeed the only source of cycles. A natural extension of the system is to include mutable references and thus cycles. In that case, we could generalize theorem 4.2, where the conclusion would be that for all resource in the heap, it is either reachable from the expression, or it is part of a cycle.

[4] A difference between (app_r) and $(match_r)$ is that, for application, the free variables ys are dynamic and thus the duplication must be done at runtime. In contrast, a match knows the the bound variables in a pattern statically. In practice, we therefore generate the required dup and drop operations during elaboration for each branch – this is essential as that enables the further optimizations as shown in §4.2.2.

$$\boxed{\begin{array}{ll} \underset{\uparrow}{\Delta} \mid \underset{\uparrow}{\Gamma} \vdash_{\mathsf{s}} \underset{\uparrow}{e} \ \rightsquigarrow\ \underset{\downarrow}{e'} & \Delta \cap \Gamma = \varnothing \quad \Gamma \subseteq \mathsf{fv}(e) \quad \mathsf{fv}(e) \subseteq \Delta, \Gamma \\ & \text{multiplicity of each member in } \Delta, \Gamma \text{ is } 1 \end{array}}$$

$$\frac{}{\Delta \mid x \vdash_{\mathsf{s}} x \ \rightsquigarrow\ x}\,[\textsc{svar}] \qquad \frac{}{\Delta, x \mid \varnothing \vdash_{\mathsf{s}} x \ \rightsquigarrow\ \mathsf{dup}\ x; x}\,[\textsc{svar-dup}]$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow\ e_1' \qquad \Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow\ e_2' \qquad \Gamma_2 = \Gamma \cap \mathsf{fv}(e_2)}{\Delta \mid \Gamma \vdash_{\mathsf{s}} e_1\ e_2 \ \rightsquigarrow\ e_1'\ e_2'}\,[\textsc{sapp}]$$

$$\frac{x \in \mathsf{fv}(e) \qquad ys = \mathsf{fv}(\lambda x.\ e) \qquad \varnothing \mid ys, x \vdash_{\mathsf{s}} e \ \rightsquigarrow\ e' \qquad \Delta_1 = ys - \Gamma}{\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \ \rightsquigarrow\ \mathsf{dup}\ \Delta_1; \lambda^{ys}\ x.\ e'}\,[\textsc{slam}]$$

$$\frac{x \notin \mathsf{fv}(e) \qquad ys = \mathsf{fv}(\lambda x.\ e) \qquad \varnothing \mid ys \vdash_{\mathsf{s}} e \ \rightsquigarrow\ e' \qquad \Delta_1 = ys - \Gamma}{\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \ \rightsquigarrow\ \mathsf{dup}\ \Delta_1; \lambda^{ys}\ x.\ (\mathsf{drop}\ x; e')}\,[\textsc{slam-d}]$$

$$\frac{\begin{array}{lll} x \in \mathsf{fv}(e_2) & \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow\ e_1' & \\ x \notin \Delta, \Gamma & \Delta \mid \Gamma_2, x \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow\ e_2' & \Gamma_2 = \Gamma \cap (\mathsf{fv}(e_2) - x) \end{array}}{\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x = e_1; e_2 \ \rightsquigarrow\ \mathsf{val}\ x = e_1'; e_2'}\,[\textsc{sbind}]$$

$$\frac{\begin{array}{lll} & \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow\ e_1' & \\ x \notin \mathsf{fv}(e_2), \Delta, \Gamma & \Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow\ e_2' & \Gamma_2 = \Gamma \cap \mathsf{fv}(e_2) \end{array}}{\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x = e_1; e_2 \ \rightsquigarrow\ \mathsf{val}\ x = e_1'; \mathsf{drop}\ x; e_2'}\,[\textsc{sbind-d}]$$

$$\frac{\Delta \mid \Gamma_i \vdash_{\mathsf{s}} e_i \ \rightsquigarrow\ e_i' \qquad \Gamma_i = (\Gamma, \mathsf{bv}(p_i)) \cap \mathsf{fv}(e_i) \qquad \Gamma_i' = (\Gamma, \mathsf{bv}(p_i)) - \Gamma_i}{\Delta \mid \Gamma, x \vdash_{\mathsf{s}} \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \ \rightsquigarrow\ \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i'; e_i'}\ \}}\,[\textsc{smatch}]$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash_{\mathsf{s}} v_i \ \rightsquigarrow\ v_i' \qquad 1 \leqslant i \leqslant n \qquad \Gamma_i = (\Gamma - \Gamma_{i+1} - \cdots - \Gamma_n) \cap \mathsf{fv}(v_i)}{\Delta \mid \Gamma \vdash_{\mathsf{s}} C\ v_1 \ldots v_n \ \rightsquigarrow\ C\ v_1' \ldots v_n'}\,[\textsc{scon}]$$

Figure 4.8: Syntax-directed linear resource rules of λ^{1}.

These theorems establish the correctness of the reference-counted heap semantics. However, correctness does not imply *precision*, ie. that the heap is *garbage free*. Eventually all live data is discarded but it may well hold on to live data too long by delaying drop operations. As an example, consider $y \mapsto^1 () \mid (\lambda x.\ x)\ (\mathsf{drop}\ y; ())$, where y is reachable but dropped too late: it is only dropped after the lambda gets allocated. In contrast, a *garbage free* algorithm would produce $y \mapsto^1 () \mid \mathsf{drop}\ y; (\lambda x.\ x)\ ()$. In the next section we present Perceus as a syntax directed algorithm of the linear resource calculus and show that it is *garbage free*.

4.3.4 Perceus

Figure 4.8 defines the syntax directed derivation $\vdash_{\mathsf{s}}$ for our resource calculus and as such specifies our *Perceus algorithm*. Like before, $\Delta \mid \Gamma \vdash_{\mathsf{s}} e \rightsquigarrow e'$ translates an expression e to e' under an borrowed environment Δ and an owned environment Γ. During the derivation, we maintain the following invariants: (1) $\Delta \cap \Gamma = \varnothing$; (2) $\Gamma \subseteq \mathsf{fv}(e)$; (3) $\mathsf{fv}(e) \subseteq \Delta, \Gamma$; and (4) multiplicity of each member in Δ, Γ is 1. We ensure these properties hold by construction at any step in a derivation.

The Perceus rules are set up to do precise reference counting: we delay a dup operation to come as late as possible, pushing them out to the leaves of a derivation; and we generate a drop operation as soon as possible, right after a binding or at the start of a branch.

Rule [svar-dup] borrows x by inserting a dup. The [sapp] rule now deterministically finds a good split of the environment Γ. We pass the intersection of Γ with the free variables in e_2 to the e_2 derivation. Otherwise the rule is the same as in the declarative system. For abstraction and binding we have two variants: one where the binding is actually in the free variables of the expression (rule [slam] and [sbind]), and one where the binding can be immediately dropped as it is unused (rule [slam-d] and [sbind-d]). In the abstraction rule, we know that $\Gamma \subseteq \mathsf{fv}(\lambda x.\ e)$ and thus $\Gamma \subseteq ys$. If there are any free variables not in Γ, they must be part of the borrowed environment (as Δ_1) and these must be duplicated to ensure ownership. The bind rules are similarly constructed as a mixture of [sapp] and [slam].

The [smatch] rule is interesting as in each branch there may be variables that can to be dropped as they no longer occur as free variables in that branch. The owned environment Γ_i in the ith branch is the intersection of $(\Gamma, \mathsf{bv}(p_i))$ and the free variables in that branch; any other owned variables (as Γ_i') are dropped at the start of the branch. Rule [scon] deterministically splits the environment Γ as in rule [sapp].

We show that the Perceus algorithm is sound by showing that for each rule there exists a derivation in the declarative linear resource calculus.

Theorem 4.3 (Syntax directed translation is sound.). *If* $\Delta \mid \Gamma \vdash_{\mathsf{s}} e \rightsquigarrow e'$ *then also* $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$.

More importantly, we prove that any translation resulting from the Perceus algorithm is *precise*, where any intermediate state in the evaluation is *garbage free*:

Theorem 4.4 (Perceus is precise and garbage free)**.** *If* $\varnothing \mid \varnothing \vdash_s e \rightsquigarrow e'$ *and* $\varnothing \mid e' \longmapsto_r^* \mathsf{H} \mid x$*, then for every intermediate state* $\mathsf{H}_i \mid e_i$ *that is not at a* dup/drop *operation* ($e_i \neq \mathsf{E}[\mathsf{drop}\,x; e'_i]$ *and* $e_i \neq \mathsf{E}[\mathsf{dup}\,x; e'_i]$)*, we have that for all* $y \in \mathsf{dom}(\mathsf{H}_i)$*,* $\mathsf{reach}(y, \mathsf{H}_i \mid \lceil e_i \rceil)$*.*

This theorem states that after evaluating any immediate reference counting instruction, every variable in the heap is reachable from the *erased* expression. This rules out, for example, $y \mapsto^1 () \mid (\lambda x.\ x)(\mathsf{drop}\ y; ())$ as y is not in the free variables of the erased expression. Just like theorem 4.2, if the system is extended with mutable references, then theorem 4.4 could be generalized such that every resource is either reachable from the erased expression, or it is part of a cycle.

The implementation of Perceus is further extended with the optimizations described in §4.2. As the component transformations, including inlining and dup/drop fusion, are standard, the soundness of those optimizations follows naturally and a proof is beyond the scope of this chapter.

4.4 Benchmarks

In this section we discuss initial benchmarks of Perceus as implemented in Koka, versus state-of-the-art memory reclamation implementations in various other languages. Since we compare across languages we need to interpret the results with care – the results depend not only on memory reclamation but also on the different optimizations performed by each compiler and how well we can translate each benchmark to that particular language. We view these results therefore mostly as *evidence that the Perceus reference counting technique is viable and can be competitive* and *not* as a direct comparison of absolute performance between systems.

As such, we selected only benchmarks that stress memory allocation, and we tried to select mature comparison systems that use a range of memory reclamation techniques and are considered best-in-class. The systems we compare are:

- Koka 2.0.3, compiling the generated C code with gcc 9.3.0 using a customized version of the mimalloc allocator [97]. We also run Koka "no-opt" with drop/reuse specialization and reuse analysis disabled to measure the impact of those optimizations.
- OCaml 4.08.1. This has a stop-the-world generational collector with a minor and major heap. The minor heap uses a copying collector, while a tracing collector is used for the major heap [37, 111, Chap.22]. The Koka benchmarks correspond essentially one-to-one to the OCaml versions.
- Haskell, GHC 8.6.5. A highly optimizing compiler with a multi generational garbage collector. The benchmark sources again correspond very closely, but since Haskell has lazy semantics, we used strictness annotations in the data structures to speed up the benchmarks, as well as to ensure that the same amount of work is done.

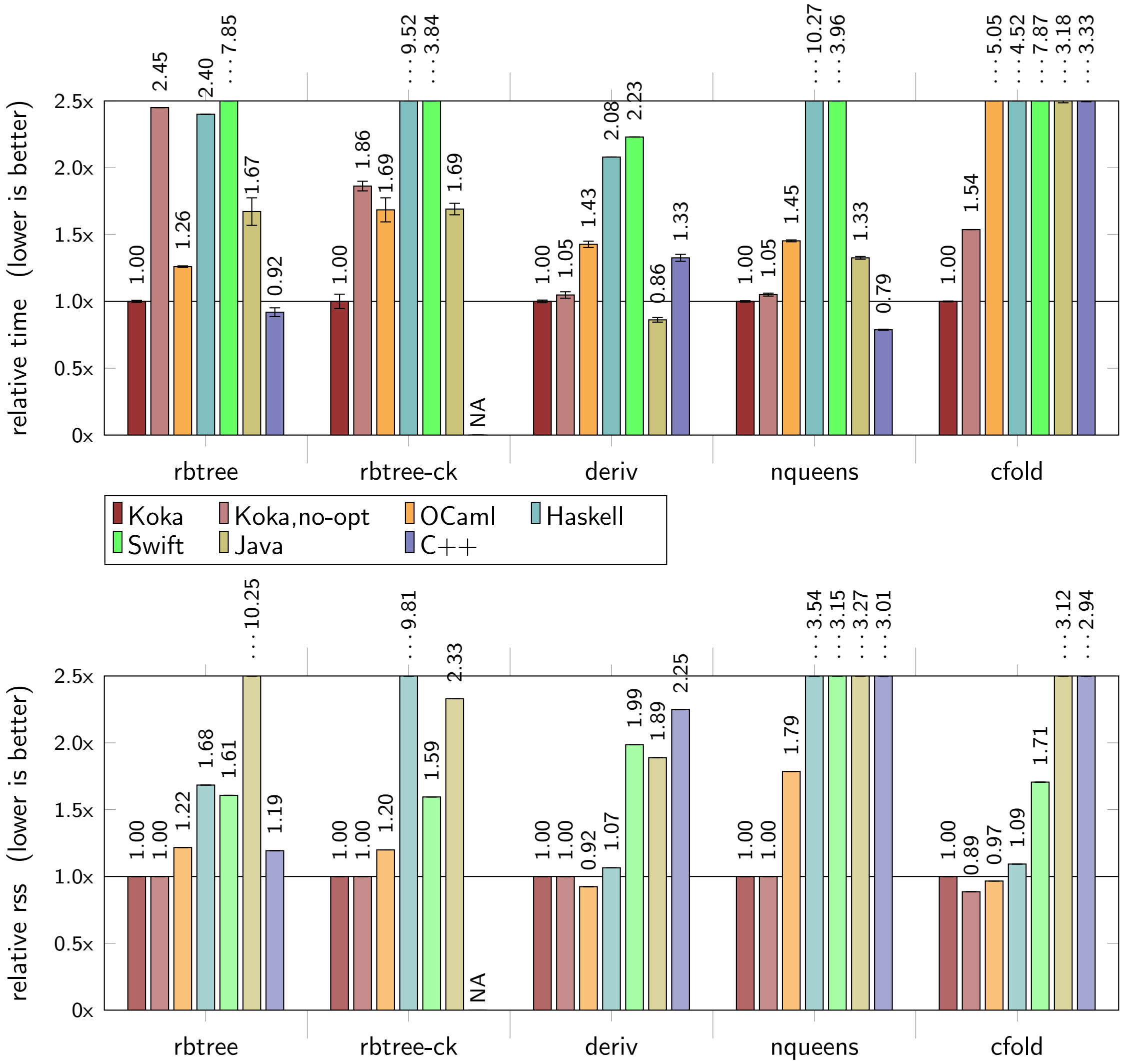


Figure 4.9: Relative execution time and peak working set with respect to Koka. Using a 6-core 64-bit AMD 3600XT 3.8Ghz with 64GiB 3600Mhz memory, Ubuntu 20.04.

- Swift 5.3. The only other language in this comparison where the compiler uses reference counting [24, 154]. The benchmarks are directly translated to Swift in a functional style without using direct mutation. However, we translated tail-recursive definitions to explicit loops with local variables.

- Java SE 15.0.1. Uses the HotSpot JVM and the G1 concurrent, low-latency, generational garbage collector. The benchmarks are directly translated from Swift.

- C++, gcc 9.3.0 using the standard libc allocator. A highly optimizing compiler with manual memory management. Without automatic memory management, many benchmarks are difficult to express directly in C++ as they use persistent and partially shared data structures. To implement these faithfully would essentially require manual reference counting. Instead, we use C++ as our performance baseline: if provided, we either use in-place updates without supporting persistence (as in `rbtree` which uses `std::map`) or we do not reclaim memory at all (as in `deriv`, `nqueens`, and `cfold`).

The benchmarks are all chosen to be medium sized and non-trivial, and all stress memory allocation with little computation. Most of these are based on the benchmark suite of Lean [153] and all are available in the Koka repository [88]. The execution times and peak working set as the median over 10 runs and normalized to Koka are given in fig. 4.9 (each benchmark runs between 1 to 5 seconds for Koka, and uses up to 300MiB of memory). When a benchmark is not available for a particular language, it is marked as "NA" in the figures.

- `rbtree`: this benchmark performs 42 million insertions into a red-black balanced tree and after that folds over the tree counting the `true` elements. Here the reuse analysis of Koka (as shown in §4.2.4) is doing well compared to the other systems. OCaml is close in performance – rebalancing generates lots of short-lived object allocation which are a great fit a minor heap copying-collector with fast aggregated bump-pointer allocation. The C++ benchmark is implemented using the in-place updating `std::map` implementation, which internally uses an optimized red-black tree implementation [49]. Surprisingly, the purely functional Koka implementation is within 10% of the C++ performance. Since the insertion operations are the same, we believe this is partly because C++ allocations must be 16-byte aligned while the Koka allocator can use 8-byte alignment in the allocations and thus allocate a bit less (as apparent in fig. 4.9, and similarly, bump pointer allocation in OCaml can be faster than general `malloc/free`). Java performs close to C++ here but also uses almost 10× the memory of Koka (1.7GiB vs. 170MiB, fig. 4.9). This can be reduced to about 1.5× by providing tuning parameters on the command line but that also made it slower on our system. This benchmark also shows the potential effectiveness of the reference count optimizations where the "no-opt" version is more than 2× slower. However, in benchmarks with lots of sharing, like `deriv` and `nqueens`, the optimizations are less effective. More generally, we expect a GC to do better when reuse optimization is not triggered, and there is lots of short-lived object allocation.

- `rbtree-ck`: it has been suggested that `rbtree` is biased to reference counting as it has no shared subtrees and thus reuse analysis can use in-place updates all the time. The `rbtree-ck` benchmark remedies this and is a variant of `rbtree` that keeps a list of every 5th tree generated and thus shares many subtrees. This pattern occurs often in practice, for example in compilers using scoped environments, or in backtracking searches where the original state is shared among different exploratory branches. Again though the reference counting strategy outperforms all other systems. Haskell and OCaml are now relatively slower than in `rbtree` – we conjecture this is due to extra copying between generations, and perhaps due to increased tracing cost. We have no C++ version of this benchmark as that would essentially require a persistent implementation of `std::map`.

- `deriv`: calculates the derivative of large symbolic expressions (up to 10M nodes). Interestingly, the memory usage of OCaml is slightly less here than Koka – since Perceus is *garbage free* we would expect though that Koka *always* uses less memory than a GC based system. From studying the generated code of OCaml we believe that it is because the optimizing OCaml compiler can avoid some allocations by applying inlining with "case of case" transformations [123] which the naive Koka compiler is not (yet) doing. It is also interesting to see that the "no-opt" Koka is only just slightly slower than optimized Koka here. This is probably due to the sharing of many sub-expressions when calculating the derivative – this in turn causes the code resulting from drop/reuse specialization and reuse analysis to mostly use the "slow" path which is equivalent to the one in "no-opt".

- `nqueens`: calculates all solutions for the n-queens problem of size 13 into a list, and returns the length of that list. The solution lists share many sub-solutions and, as in `deriv`, for the C++ version we do *not* `free` any memory (but do allocate the same objects as the other benchmarks). Again, Koka is quite competitive even with the large amount of shared structures, and the peak working set is significantly lower.

- `cfold`: performs constant-folding over a large symbolic expression (2M nodes). This benchmark is similar to the `deriv` benchmark and manipulates a complex expression graph. Koka does significantly better than other systems. Just as in `deriv`, we see that OCaml uses slightly less memory as it can avoid some allocations by optimizing well. The "no-opt" version of Koka also uses 11% less memory; this is because the reuse analysis essentially holds on to memory for later reuse. Just like with scoped based reference counting that may lead to increased memory usage in some situations.

An interesting overall observation is that the reference counting implementation of Swift seems less effective than Koka – this may be partly due to the language and compiler, but we also believe that this may be a confirmation of our initial hypothesis where we argue that a combination of static compiler optimizations with dynamic runtime checks (e.g. `is-unique`) are needed for best results. As discussed for example in §4.2.7, some of the optimizations we perform are difficult to do in Swift as the static guarantees of the language are not strong enough. More research is needed though to confirm this as there may be other causes well unrelated to reference counting as such.

Finally, we also ran our benchmarks using just atomic operations for our reference counts to see the impact of the thread-shared flag. We observed a slowdown from 5% (`rbtree`) up to 59% (`nqueens`) across our benchmarks. This matches the observations by Ungar et al.[154] who performed a similar experiment in Swift.

4.5 Conclusion and Future Work

In this chapter we present Perceus, a precise reference counting system with reuse and specialization, which is built upon λ^1, a novel linear resource calculus closely based on linear logic. Our implementation in Koka is competitive with other mature memory collectors over our benchmark suite but more experimentation in larger systems is needed. We would like to integrate selective "borrowing" into Perceus – this would make certain programs no longer be *garbage free*, but we believe it could deliver further performance improvements if judiciously applied. It also remains to be seen how to handle cycle collection efficiently. Finally, the explicit control-flow is not zero-cost (like C++ exception handling), and it would be interesting to see if this can be improved further.

Chapter 5

Practical considerations for DSL design in the C ecosystem

Programming systems research does not happen in a vacuum; a successful project will be used by programmers who are not only familiar with, but critically depend upon, decades of prior art. Therefore, if a research project is to have a practical impact, every incompatibility must measurably offset its cost with some clear benefit to its users. This chapter explores some practical considerations for language designers in the HPC space; these lessons were learned throughout the author's role as an open source maintainer for Halide, specifically for its build system, testing, continuous integration, and release cycle.

We focus in particular on C and C++ because they remain the primary implementation languages in most performance-sensitive domains. Operating system kernels, device drivers, databases, CAD tools, AAA video games, and more, are all typically written in C or C++. Even Python-based deep-learning frameworks, like TensorFlow, PyTorch, and Jax, spend the majority of their execution time in high-performance kernels written in C/C++. Indeed, Python performance infamously *depends* on maximizing the time spent executing C-code modules, rather than plain Python modules [11].

Even the burgeoning category of "better C" languages like Nim, Odin, Rust, and Zig all maintain compatibility with existing C code. Among these, Nim compiles to C code outright, while the others target LLVM and so rely on typical C-compatible linkers. Zig and Nim maintain ABI compatibility with C, while Rust and Odin provide first-class features for binding to C libraries. When these languages conservatively reject a performance-critical program, these interoperability features provide an important escape hatch.

It is therefore imperative that the DSL researcher understands the practical challenges in the C ecosystem. We will touch on two in this chapter. We first examine challenges with writing and generating correct, portable, high-performance C code. Then, we cover two issues with build systems in this context: the challenges of writing a correct build system for an ordinary C program and of integrating a DSL compiler into existing build systems.

5.1 Compiling DSLs to C

Writing a compiler backend is a daunting task, even with the advent of reusable compiler backends like LLVM [90]. Although LLVM can dramatically simplify compiler engineering, it is not perfect, and several issues prevent it from being an obvious choice for a new compiler.

First, LLVM is a very heavy dependency, consisting of millions of lines of code. It takes hours to compile on commodity hardware and compiles to gigabytes of binary code. Debugging issues with LLVM is complex and time-consuming, and upstreaming bug-fixes requires significant investment into to the project (either social or financial). The LLVM API and IR formats are incompatible between major releases, meaning downstream compilers must continuously upgrade or risk obsolescence. As a result, bindings to LLVM for languages besides C/C++ (like Python and Haskell) tend to lag several versions behind.

Second, LLVM supports only a handful of popular processor architectures. By contrast, GCC can compile C to AVR, Motorola 68k, MSP430, VAC, and *dozens* of less-common architectures that are still in use today. Unfortunately, GCC's intermediate representation (GIMPLE) is not effectively reusable outside of GCC. When designing a new architecture, it is common to prioritize writing a C compiler for it, but these compilers are usually proprietary.

Finally, C code is easier to read, generate, and debug, than LLVM. It is higher level, which makes mapping common control structures simpler. A variety of mature tools exist for analyzing and debugging C code. Compiling a higher level language to LLVM typically requires one or more IRs to apply language-specific optimizations before LLVM code can be emitted; see Rust's MIR [105], and Swift's SIL [146].

For these reasons and more, many programming languages (especially research languages) compile to C instead [32, 72, 74, 91, 94, 117], including Exo (chapter 3). Some languages, like Standard ML [112], Chapel [17], and Halide [129] have *optional* C backends for expanding compatibility beyond their default backends. Even C++ originally compiled to C when it was still called "C with classes"[142].

5.1.1 Undefined integer behavior

Yet, C is a veritable minefield of undefined or implementation-defined behavior and precarious performance cliffs. Some of the worst and most subtle bugs come from the misuse of integer types in C. Indeed, very little can be assumed about how the basic types work at all. Language authors must be aware of the differences between C's integer semantics and their language's integer semantics when writing a compiler.

For one, the number of bits in a byte is not standardized. Although POSIX defines a `char` (a "byte") to have 8 bits exactly, C has no such restriction. Prior to C11, one `char` did not even have to fit an octet (to support 7-bit systems). Since C11, a byte must consist of at *least* 8 bits, but could be wider. No version of the C standard enforces that integers be represented in two's complement form. However, C++20 and beyond *does* require two's complement integers. The rules governing the widths of `int`, `short`, and `long` allow them to all be equal in size.

One solution to is to always generate the sized types from `stdint.h`, such as `int32_t`, which specify an exact bit width. If your source language has an 8-bit signed integer type, it should be stored in an `int8_t` after translation. The existence of this type is implementation-defined, so compilers for exotic DSP architectures with 16-bit bytes (like the C55x series from TI [148]) will correctly reject programs that use it (or, perhaps, emulate it with more expensive operations). Such targets tend to have other peculiarities [147], too, so special care needs to be taken when supporting them, anyway.

Another issue is that C's rules for type promotion are byzantine. When an operation is performed across two integer types, one is "promoted" to the other's type. For a concise example, consider the expression `-1L > 1U`; the value of this expression is implementation defined. On systems (such as POSIX x86-64[1]) where `long` is wider than `unsigned int`, the shorter `unsigned int` will be promoted to `long` and the comparison will evaluate to `0`. However, on systems (such as x86-32) where the sizes are the same, then `unsigned` wins instead; this means that the value -1 will wrap to `UINT_MAX`, which is guaranteed to be greater than 1, hence the expression evaluates to `1`. When generating code, care should be taken to ensure that types align prior to C code generation.

Even when all the types agree, expressing certain "obvious" operation identities can be unsafe or invoke undefined behavior. For instance, `x << 0` is safe only when x is positive since shifting a negative value by *any* number of bits, including zero, is undefined. Double negations, i.e. `-(-x)`, are also tricky because negating the most negative signed value is undefined (and in fact traps on some architectures).

These operations, rather than doing nothing, indicate to modern compilers that certain undefined behaviors do not happen on paths that reach them; this can result in hard to detect bugs (and even security vulnerabilities) when the compiler uses this information to delete edge case checks. Due to function inlining, these problems can appear very far from the implicating code. As such, arithmetic simplifications in accordance with the source language's semantics should be performed prior to C code generation.

Enumerating all integer undefined and implementation-defined behaviors, to say nothing of *all* such behavior, is out of scope for this chapter. The reader is referred to Dietz et al. [36] for further understanding the full scope of this problem. The difficulty of mapping integer semantics from a DSL source language to C should at this point be evident. For more on undefined behavior generally, the reader is directed to Chen [20], Lattner [89], Lee et al. [92], and Regehr [132].

5.1.2 Vectorization

Although C/C++ remain the most popular choices for writing high-performance SIMD code, the standard has so far declined to add vector types or SIMD to the language specification. Hence, such programs will necessarily be less portable than their scalar counterparts. The main decision, then, is *how* to break portability in the narrowest way possible.

[1] But not *Microsoft* x86-64! It sets `int` and `long` to the same size for backwards compatibility reasons.

One popular approach is to leverage compiler-specific features. GCC and Clang both provide an architecture-independent vector type extension that is simple to use and target with a code generator. For example, to declare a type that holds eight floating point values and a function to compute a fused multiply-add (FMA), one could write:

```
typedef float float8 __attribute__((vector_size(8 * sizeof(float))));

float8 do_fma(float8 x, float8 y, float8 z) {
  return x + y * z;
}
```

Clang 15, targeting x86, compiles this to a single FMA instruction (plus a return). On ARM NEON, however, it compiles to *two* FMAs plus the requisite loads and stores since NEON vectors are only four floats wide. On the one hand, this might be convenient since the code does not need to be modified for the new architecture in order to compile; on the other, it is difficult to audit the assembly for a large program using these features. Nothing indicates that an 8-wide vector doesn't exist on the target.

This extension is only one of many alternatives implemented in C compilers. The IBM XL C compiler supports a separate vector type extension based on AltiVec. OpenCL C has yet another extension unique to it. The Microsoft C compiler lacks any vector extension. The Intel compiler (prior to adopting the Clang frontend) supports the GCC extensions described above. ARM defines a set of language extensions for NEON and SVE vectors; Clang attempts to support every aforementioned extension. Among all these options, the GCC-compatible extensions are the most feature-complete.

Somewhat more portable is to use vector *intrinsics*, C functions that directly model certain vector assembly instructions. When targeting x86 systems the <immintrin.h> header is portable across most major C compilers, including Intel's, GCC, Clang, and MSVC. Similarly, the <arm_neon.h> header provides intrinsics for ARM NEON vectors. The example above written with x86 intrinsics would be:

```
#include <immintrin.h>

__m256 do_fma(__m256 x, __m256 y, __m256 z) {
  return _mm256_fmadd_ps(x, y, z);
}
```

Exo, described in chapter 3, side-steps this portability issue by generating exclusively standard C11 in the core compiler. Because users achieve vectorization by defining custom instructions via C code strings, they retain control over the relevant portability trade-offs. Naturally, users will not exclude their own toolchain. We believe this is a good design, though we hope to add abstract vector types and operations in the future.

```c
void stencil_sum(float *input, int m, int n, float *output) {
  for (int jo = 0; jo < ((n - 2) / 64); jo++) {
    for (int io = 0; io < ((m - 2) / 64); io++) {
      float tile_buf[66 * 64] = {0};
      // First loop nest:
      for (int ji = 0; ji < 66; ji++) {
        for (int ii = 0; ii < 64; ii++) {
          tile_buf[64 * ji + ii] =
              input[m * (jo * 64 + ji) + io * 64 + ii]
            + input[m * (jo * 64 + ji) + io * 64 + ii + 1]
            + input[m * (jo * 64 + ji) + io * 64 + 2 + ii];
      } }
      // Second loop nest
      for (int ji = 0; ji < 64; ji++) {
        for (int ii = 0; ii < 64; ii++) {
          output[(m - 2) * (64 * jo + ji) + 64 * io + ii] =
              tile_buf[64 * ji + ii]
            + tile_buf[64 * (ji + 1) + ii]
            + tile_buf[64 * (ji + 2) + ii];
} } } } }
```

Figure 5.1: A simple two-stage stencil that exhibits several subtle vectorization issues on Clang 11.0.1 with flags `-O2 -ffast-math -mavx`. The input and output should be declared `restrict` to indicate that they cannot alias. The *second* loop nest vectorizes cleanly, but the *first* does not. There are several resolutions: (1) the subexpression `2 + ii` can be rewritten to `ii + 2`, (2) the loop counter variables can be changed to `long` or `int_fast32_t`, or (3) the literal `2` can be manually expanded to a `long` by writing it as `2L`.

The one common strategy that *does not* work, however, is to trust the C compiler to automatically vectorize code. Consider the simple two-stage stencil in fig. 5.1[2]. It consists of two loop nests: the first computes a horizontal sum and the second computes a vertical sum. Clang 11.0.1 (using flags `-O2 -ffast-math -mavx`) compiles the second loop to this:

```
.LBB0_15:
        vmovups ymm0, ymmword ptr [rsp + rcx + 384]
        vaddps  ymm0, ymm0, ymmword ptr [rsp + rcx + 128]
        vaddps  ymm0, ymm0, ymmword ptr [rsp + rcx + 640]
        vmovups ymmword ptr [rax - 224], ymm0
      # ... seven similar repetitions elided ...
        add     rcx, 256
        add     rax, rdx
        cmp     rcx, 16384
        jne     .LBB0_15
```

This is reasonable vector code for this loop. The loads, stores, and additions are exactly

[2]For an interactive version of this example, see https://godbolt.org/z/nrM3E6Wcz for the original code only and https://godbolt.org/z/eYdMK44z3 for a side-by-side comparison.

```c
void stencil_sum(float const * restrict input,
                 int64_t m, int64_t n,
                 float * restrict output) {
  for (int_fast32_t jo = 0; jo < ((n - 2) / 64); jo++) {
    for (int_fast32_t io = 0; io < ((m - 2) / 64); io++) {
      float tile_buf[66 * 64] = {0};
      // First loop nest:
      for (int_fast32_t ji = 0; ji < 66; ji++) {
        for (int_fast32_t ii = 0; ii < 64; ii++) {
          tile_buf[64 * ji + ii] =
              input[m * (jo * 64 + ji) + io * 64 + ii]
            + input[m * (jo * 64 + ji) + io * 64 + ii + 1]
            + input[m * (jo * 64 + ji) + io * 64 + ii + 2];
      } }
      // Second loop nest
      for (int_fast32_t ji = 0; ji < 64; ji++) {
        for (int_fast32_t ii = 0; ii < 64; ii++) {
          output[(m - 2) * (64 * jo + ji) + 64 * io + ii] =
              tile_buf[64 * ji + ii]
            + tile_buf[64 * (ji + 1) + ii]
            + tile_buf[64 * (ji + 2) + ii];
} } } } }
```

Figure 5.2: A corrected version of the program in fig. 5.1. This version uses precise types at the API boundary, including correct `const` qualifications. It uses the `int_fast32_t` type for loop iteration counters, which is no worse than before in terms of overflow correctness, but prevents excess sign extensions. Finally, commuting terms in index expressions are sorted.

those in the original program, with nothing extraneous in between. On the other hand, the first loop is not so lucky. It is compiled to:

```asm
.LBB0_12:
        lea     edi, [rdx + r10]
        imul    edi, r13d
        add     edi, dword ptr [rsp + 120]   # 4-byte Folded Reload
        mov     rsi, rdx
        movsxd  rdi, edi
        vmovups ymm0, ymmword ptr [rbx + 4*rdi + 4]
        vaddps  ymm0, ymm0, ymmword ptr [rbx + 4*rdi]
        lea     ebp, [rdi + 2]
        movsxd  rbp, ebp
        vaddps  ymm0, ymm0, ymmword ptr [rbx + 4*rbp]
        shl     rsi, 6
        vmovups ymmword ptr [rsp + 4*rsi + 128], ymm0
      # ... seven similar repetitions elided ...
.LBB0_13:                                    #   in Loop: Header=BB0_5 Depth=3
      # ... five instructions elided ...
.LBB0_5:                                     #   Parent Loop BB0_2 Depth=1
```

```
        # ... ten instructions elided ...
         jl      .LBB0_12
```

This is a mess. Many instructions are dedicated to computing addresses, some intermediate values have spilled onto the stack, the loop control logic has ballooned from three instructions to increment a loop counter to fifteen (elided for space) instructions with extra jumps and labels. Why isn't this code as tidy as the previous code?

It turns out there are at least three different, surprising, ways to get better code generation:

1. Rewrite the sub-expression `2 + ii` to `ii + 2`.
2. Replace the `2` in that same expression with `2L`.
3. Use (on this platform) `long` everywhere instead of `int`.

The underlying reason common to each of these cases is that the type of a pointer is wider than `int` on this platform. Hence, sign-extension IR instructions must be emitted to preserve C's semantics. However, this frustrates common sub-expression elimination in this case, and so the expression `m * (jo * 64 + ji) + io * 64 + ii` is not factored out of all three indices. Downstream, this disables important loop optimizations that eliminate the index math in the loop. The latter two points strategically remove the sign extensions, which allow LLVM's CSE to work correctly and the loop is then vectorized cleanly.

The situation is *even worse* on newer versions of Clang (all through 15.0.0, the latest at time of writing), which emit no vector operations for the first loop whatsoever. Additionally, versions of GCC below 11.1 are unable to generate good vector code for this example, and even good versions do so only at the highest optimization level (lower levels vectorize, but do not unroll the loop, as should be done). So automatic vectorization is unreliable both between compiler vendors and across versions from a single vendor.

An improved version of this program can be found in fig. 5.2, but this should only serve to underscore the following: automatic vectorization is fickle and attempting to anticipate its peculiarities from a code generator is quixotic. Do not design DSL compilers that expect auto-vectorization to happen in C.

This is one area where targeting LLVM directly can pay off. LLVM has built in vector types for generic vector backends, but also exposes finer-grained instruction selection. One pitfall is that LLVM's loop optimizations assume that their input was produced by Clang and so tend to *de-optimize* vector code[3]. Hopefully MLIR will further increase the level of abstraction and reliability at which one can generate good vector code for common platforms.

5.1.3 Compiler optimization

Vectorization is not the only compiler optimization that is too unreliable for C code generation. C optimizers are engineered to process code written by humans and tend to struggle on

[3]In fact, Halide offers an option to disable them; eventually, they will be disabled by default.

machine generated code. For instance, some optimizations scale super-linearly with the number of basic blocks in a function, so very large functions can be slow to optimize. On the other hand, breaking a computation up into many small functions will stress inlining heuristics and risk wasting too much runtime on function call overhead. In the next few sections, we will look at commonly missed optimizations that particularly affect DSL authors.

Recursion

Control flow in numerical DSLs tends to be oriented around for-loops, which have a direct translation to C. Optimizing this sort of flat, static control flow has been the subject of decades of engineering effort in C compilers, and so one can generally expect such loops and basic blocks to be well optimized.

However, considerably less effort has been spent optimizing across functions calls. Compiler engineers familiar with functional languages like Haskell and OCaml might be surprised to learn that C compilers often miss opportunities for profitable function inlining, recursion elimination, and more. These missed optimizations can render recursive programs unsuitable for practical use, as their consumption of stack space becomes prohibitively high[4].

These types of problems plague even formally verified systems, such as [162]. The system described in the paper synthesizes the following code to copy a singly linked list:

```
void sll_copy(loc r) {
  loc x2 = READ_LOC(r, 0);
  if (x2 == NULL) {
    return;
  } else {
    int v = READ_INT(x2, 0);
    loc nxt = READ_LOC(x2, 1);
    WRITE_LOC(r, 0, nxt);
    sll_copy(r);
    loc y12 = READ_LOC(r, 0);
    loc y2 = (loc)malloc(2 * sizeof(loc));
    WRITE_LOC(r, 0, y2);
    WRITE_LOC(y2, 1, y12);
    WRITE_INT(y2, 0, v);
    return;
  }
}
```

This program is recursive, but not *tail*-recursive. To be efficient, the compiler would need to perform a program transformation called *tail recursion modulo cons*[98]. TRMC applies to functions whose recursive calls *would be* in tail position, except that the recursive value

[4] Microsoft Windows has a default stack size of only 1MB, which is tiny. The Halide compiler, which uses recursion to manage tree traversals, has to manually offload work onto a separate thread (actually a "fiber") with a larger stack space allocated.

is passed to a data constructor. TRMC safely converts such functions into an effectful destination-passing style, allowing normal tail-recursion elimination to proceed.

However, I know of no C compiler that performs TRMC, including the latest versions of Clang, GCC, MSVC, and Intel ICC. This program therefore uses $O(n)$ auxiliary stack space when copying a list of n elements. Although it is formally verified to produce a true copy without violating heap integrity, it cannot actually be deployed. The formalism in the paper does not model a call stack and so constraints on resource consumption due to recursion cannot be stated to the synthesizer.

Yet, the paper admirably dedicates an entire section to explaining how its intermediate DSL is faithfully translated to C. The `loc` type and read/write macros in the code above are designed to avoid undefined behavior that would come from punning pointer and integer types. Clearly, much thought was put into C code generation but, as Donald Knuth might say, the program was only proven correct, not tried.

An accompanying blog post[55] positions this work as “synthesising completely correct C-code” and “removing any need to trust fallible human developers”. Yet, this stack space issue reminds us that formal verification can only prove properties it was asked to prove, and that accounting for every constraint is not necessarily easier than writing a correct program. This is my attempt:

```c
typedef struct node node;
struct node { int64_t data; node *next; };

node *sll_copy(node const *cur) {
  node *new_list = NULL, *prev = NULL;
  for (; cur; cur = cur->next) {
    node *tail = malloc(sizeof(*tail));
    tail->data = cur->data;
    if (new_list == NULL) {
      new_list = tail;
    } else {
      prev->next = tail;
    }
    prev = tail;
  }
  if (prev) { prev->next = NULL; }
  return new_list;
}
```

It is not especially difficult to see this program produces a true copy of the input list using constant stack space and while avoiding undefined behavior. It also uses a somewhat more idiomatic API: the function returns a pointer to the new list, rather than writing it back to the source location, and uses a struct type for nodes instead of an array of untagged unions.

Memory layout

The primary language feature for creating data structures in C is the struct, which is a nominal product type over a list of members. Empty structs are not portable between C and C++, as they are given different sizes, and so should be avoided during code generation. The `void` type should be preferred.

The memory layout of a struct is dictated by the declaration order of its members and the compilation target. Members are placed in memory in the same order as their declaration but *padding* might be inserted between members to ensure proper alignment for the target. The compiler is not allowed to reorder struct members into a more compact layout. Generated structs will need to sort their members for optimal layout prior to C code generation.

When choosing a memory layout, there are several competing pressures to consider. First, although the C standard does not guarantee anything about the size or amount of padding for a struct, it does guarantee that no padding is present before the first member. This rule explicitly allows casting a pointer to a struct to a pointer to its first member, which can be used to efficiently implement a form of single-inheritance subtyping. Second, large structs will want to place temporally correlated members next to one another, ideally not crossing a cache line boundary. Third, the members should be arranged to minimize padding, which is simply wasted space. Finally, flexible array members may only occur last.

Many compilers offer language extensions for controlling struct layout more precisely. Most compilers offer a `#pragma pack` directive that eliminates padding at the cost of extra computation time to extract members. This is most useful at data serialization boundaries, where knowing the exact layout of a struct is important. It is not typically a good idea to use packed structs for normal data representation since the extraction costs can be quite high.

Arrays of structs are laid out in memory with each element placed whole, one after the other, with members and padding interleaved. As such, iterating over a single member of each array element (such as just the red channel in an image), forces the cache to load unused data and padding. Thus, it can be very profitable to represent arrays of structs (AoS) as structs of arrays (SoA), instead, with one array dedicated to each member of the original struct. The Dex[121] language always represents arrays of product types in SoA format.

There are several strategies for implementing SoA. The simplest is to define a record-keeping struct that holds the size and start pointers for n arrays, one for each member of the original struct. This requires $n + 1$ allocations and frees, each, to create and destroy the SoA. This is easy to implement and is broadly portable. It also enables some interesting data structure operations, like swapping out one member array for another, or freeing a member if its lifetime is shorter than the other members.

If the number of allocations or memory fragmentation are an overriding concern, one can use a flexible array member to hold the data for all n arrays. The start pointers in the SoA point into the flexible array member, with care taken to ensure that alignment is respected. This way, a single call to `malloc` and `free` can allocate and release the entire SoA. Note that flexible array members are not standard C++. In practice, however, every major compiler supports them.

Communicating assumptions

It is typical for a DSL to have stricter semantics than C, and for these semantics to enable optimizations that are not possible in C without making stronger assumptions about the structure of a program.

For instance, in C, pointers are allowed to alias one another freely. This forces the compiler to be conservative about ordering accesses to memory when manipulating multiple pointers simultaneously. The `restrict` keyword, when applied to a pointer declaration, promises the compiler that the given pointer will not alias any other pointer in scope. This is fairly coarse, but is essential to achieving high performance out-of-place computations.

Other assumptions can be communicated via non-portable language extensions such as Clang's `__builtin_assume` function, which takes a boolean expression that should be assumed to be true after the call. This is useful for communicating alignment constraints on a pointer, assuring the compiler a value (such as the length of a list) is positive, and so forth.

Absent explicit features for communicating assumptions, one can place the assumption in a branch that, if violated, would surely invoke undefined behavior. This is unreliable, so a standard way to communicate these assumptions is being added in C++23.

Security considerations

Some compiler optimizations have serious security implications. For instance, calls to the standard `memset` and `memcpy` functions are commonly elided by modern C compilers, even though they have highly optimized implementations. This unfortunately means that these functions must not be used when writing to memory is *required*, such as when clearing a buffer that stores a cryptographic secret or communicating with an MMIO device. The functions `memset_s` and `memcpy_s` were introduced in C11 to guarantee that copies actually happen.

5.2 Building C and C++ programs

Once one has written a fast C program, one will most likely need to *share* that program with colleagues, artifact review committees, industry contacts, and so forth. Yet, none of these people are guaranteed to use the same compiler, operating system, or even processor architecture. Thus, portability must be considered. Each person should be able to build and run the code or, if they cannot, understand precisely why it is impossible.

Unfortunately, C is unusual in that there are innumerable competing implementation and platform combinations, each with mutually incompatible interfaces and non-standard extensions. Any assumption based on the behavior of a single point in this space is unlikely to hold elsewhere, even on supposedly similar systems.

Merely *building* a C program is shockingly complex, much more so than is widely appreciated, and even in the best-case scenario where the code itself adheres strictly to language standards. There is no standardized build system for C or C++, and so nearly every project ends up developing some amount of custom tooling for driving their builds.

The DSL author will encounter challenges integrating their new languages into existing build systems without an understanding of the challenges involved with building C and C++ code in the first place. It is easy to unintentionally introduce portability constraints.

5.2.1 Writing a build system

To illustrate this, consider the Makefile. Makefiles are taught in nearly every top-tier computer science program as the preferred way to build C code. Yet, the examples of Makefiles appearing in these curricula [3, 4, 40, 41, 56, 102, 120, 122, 127] contain serious portability bugs and demonstrate bad practices[5].

Let us examine one Makefile taken from MIT OpenCourseWare[3]. It builds an application, `program`, from two source files: `main.c` and `iodat.c`. The latter has a corresponding header file which is included by both source files. What is non-portable about the following?

```
program: main.o iodat.o
    cc -o program main.o iodat.o
main.o: main.c
    cc -c main.c
iodat.o: iodat.c
    cc -c iodat.c
```

There is very much that is non-portable. One of its most basic mistakes is to hard-code the compiler as `cc`. This makes switching the compiler to cross-compile the program for another architecture needlessly difficult. One would have to write a wrapper script, also named `cc`, that forwards its arguments to the real compiler to avoid patching or outright replacing the build. There is an established convention to use the `$(CC)` variable to invoke the C compiler.

It also makes some strong assumptions about the compiler command line. First, it assumes the compiler understands the `-c` and `-o` flags at all, which is not the case with at least the Microsoft compiler. Second, it assumes that an appropriate output filename will be computed from the input filename when none is specified. It would not be surprising to learn that some compiler considers this an error.

The commands are also incomplete. POSIX systems generally expect certain environment variables to affect the command lines of compiler invocations. In particular, the `CFLAGS` and `LDFLAGS` variables should be expanded into compiler and linker commands, respectively. Without these, the Makefile must be edited to enable program optimizations.

There are also some basic engineering flaws. Each rule repeats the names of files in both the dependency list and the command. Discrepancies between these lists, especially as the Makefile grows, will be hard to notice and pinpoint when errors arise. In a command specification, the automatic variable `$^` expands to the full list of dependencies and should be used instead.

[5] One Makefile[75], written in 1998 at UC Berkeley, stands out as the only (nearly) correct example.

There is no attempt to establish dependencies on header files, whether they are part of the project (such as `iodat.h`) or provided by the system. If a header is changed, or the system is upgraded, the project must be rebuilt from scratch. There is also no attempt to establish dependencies on the Makefile commands themselves, so incremental builds cannot accurately update files whose command lines changed[6].

Finally, and this is common to many Makefiles, it writes build outputs directly to the source directory. This runs the risk of overwriting files unintentionally, and frustrates common workflows that keep separate binary directories for incompatible configurations. For instance, one might use one directory to keep the artifacts for a build that has debugging features and instrumentation enabled and another directory for an optimized build.

The lecture goes on to correct a few of these issues, but even so, the lesson fails to teach students to write robust build systems. Again, MIT is not an outlier in this respect; sadly, most top-tier computer science programs fail to teach these skills, too. More disappointing is that these pitfalls have been well known for decades. The 1998 article "Recursive Make Considered Harmful"[110] details some particularly egregious misuses of Make, but also discusses a correct approach to declaring header and inter-project dependencies.

Such discussions tend to implicitly assume that Make is a single system, but this is not true. There are in fact many implementations of Make, including GNU Make, yes, but also BSD Make and Microsoft NMake. Each one supports mutually incompatible extensions on top of a tiny and inexpressive core language. For instance, BSD Make uses the variable `${.IMPSRC}` to refer to the implicit source file in a so-called "suffix rule". GNU Make uses "`$<`" for this, instead. NMake uses "`$<`" too, but then uses a distinct syntax for declaring the suffix rule in the first place.

So even writing a perfect Makefile, with none of the aforementioned issues, using one implementation of Make is no guarantee of portability to another one. The solution is to use an abstraction layer. For example, CMake[26] can express a correct, portable build system for this same program in only three lines of code:

```
cmake_minimum_required(VERSION 3.24)
project(example LANGUAGES C)

add_executable(program main.c iodat.c)
```

This isn't an argument to use CMake specifically, but it is perhaps an argument to stop teaching Make as a C build system. Similar benefits can be found by using any modern build system, including Bazel[10], Meson[149], and others.

Still, this is not a panacea. Even when the operating system and compiler is predetermined, portability concerns remain. In 2021, the Linux kernel attempted to enable a "warnings as errors" compiler setting, which backfired badly.

Linux is highly configurable and exposes hundreds of settings; multiplied by a few dozen compiler versions and a handful of target architectures, the total number of build

[6]In fairness, this is quite tricky to accomplish with any flavor of Make

configurations is staggering. Probabilistically speaking, there is no reason to believe that every point in this combinatorially large space would be free of compiler warnings. And indeed, it is not the case.

The backlash was swift. Within a day, the Linux Kernel Mailing List was abuzz with users whose continuous integration systems were broken by this change. After some discussion, it was decided that warnings-as-errors should be enabled by default only for builds that are explicitly meant to test the build process itself.

Ultimately, this fiasco could easily have been avoided by following the *golden rule of build specification*: include in the specification exactly what *must be* included, no more or less.

Warning flags never impact the correctness of a build or code generation. They must therefore be enabled through the build system's injection facilities. These include the `CFLAGS` environment variable for Make and compatible systems, and the `CMAKE_C_FLAGS` and `CMAKE_COMPILE_WARNING_AS_ERROR` variables in CMake. On the other hand, the list of source files always matters. Configuring include paths and library linking flags are also non-negotiable.

One trickier case is source files that use SIMD intrinsics; such files usually *require* special compiler configuration in order to compile successfully. Enabling these flags on a per-source-file basis is not necessarily safe, since it can impact the ABI. In these cases, the appropriate response is to run a *test* compilation to check whether the user has already configured the build system correctly. If so, do nothing. When this is not the case, a configuration point may be added that contains a *presumed* flag that will be added. For AVX2, this might be `-mavx2` on Clang and GCC and `/arch:AVX2` on MSVC. It is acceptable to use heuristics to determine a default value[7].

Even small projects must be aware of these issues. The sensitivity of warning flags is not stable between versions of a single compiler. New warnings are sometimes added to the default settings, which would make upgrading the compiler nearly impossible if warnings-as-errors is enabled.

This anecdote should caution a C developer to be mindful of the combinatorially large space of development environments that exist. One should never ship a C build system with warnings promoted to errors by default. Yet, warnings should not be ignored and one's development process should not allow merging code with known warnings.

The best solution is usually to provide some mechanism for *injecting* these flags from outside the build. Makefiles that respect `CFLAGS` allow the user to set `-Wall -Werror` when their compiler supports it. The CMake `CMAKE_C_FLAGS` and `CMAKE_COMPILE_WARNING_AS_ERROR` variables serve a similar purpose. In any case, these injection points can be set by CI systems and developers who are working directly on a project, but ignored by users who simply wish to integrate said project into another system.

[7]To support runtime dispatch between kernels with different ISA mixes, run test compiles for each ISA extension and include only those kernels whose requirements are met. If no kernels match, issue a configure-time error. Include logic to report which kernels were enabled in the build output. Ideally, empower the user to disable runtime dispatch and force a single kernel.

In all, there is enough to make one's head spin, and the discussion has so far been limited to build systems and compiler settings. Here are a few concrete recommendations that will help any project play nicely with other systems.

1. A build system is software, too, and should be tested. Free continuous integration systems have emerged in the last five or six years that provide easy access to Windows, macOS, and Linux machines on the cloud. This is perhaps the single most effective way to achieve basic portability.

2. Every hard-coded compiler flag is a liability. The build definitions should limit themselves only to settings that are *absolutely required* to produce usable binaries on *any* platform. Warning flags are not in this category. Common sets of settings can be stored in non-portable build scripts strictly as a convenience. These scripts should not sit on the critical path between obtaining the code and running it.

Speaking from experience developing Exo, the Chipyard build system was a constant source of delays and confusion. Time spent dealing with bugs and restarting from scratch on failures cost our team multiple person-weeks of time. Eventually, I spent two days writing a new build system for the pieces of the code that we needed. Scaled up to an entire industry, the consequences of bad build systems on research productivity are disturbing.

5.2.2 Integration with existing build systems

In the previous section, we saw that Make's lack of abstraction for the C and C++ compilation process makes it a cumbersome choice for building portable software. However, the solutions provided by modern build systems tend to be more rigid and restricted than is necessary.

The most effective way to integrate a new DSL compiler into an existing build system depends on two major factors: the format of the compiler output and the discovery of dependencies needed for a compiler invocation.

Output format

Most build systems can easily handle DSL compilers that output C or C++ source code (as discussed in §5.1). As an example, CMake provides a function called `add_custom_command` that defines a rule for producing a set of object files from a set of input files using a given program. If the output files are plain C, they can be added to the list of sources for an executable or library target.

However, if the compiler intends to output a binary, things get more complex. Somehow, the DSL compiler will need a description of the target platform to use, and there is no standard or portable format for describing platforms, so a custom solution is needed for each build system. Halide, for example, must translate CMake's toolchain descriptions into a "target string", which is then translated to API calls to LLVM. We must also do this in Bazel, and we have seen users attempt to do it in Meson (though, so far, only incorrectly).

There is also the question of whether it is better to emit object files or fully prepared build artifacts like executables or libraries. In general, it is better to emit object files. First, the end user might need to configure the linker specially, such as to rename certain symbols for ABI versioning, or for including multiple copies of the library. This is impossible to do once the linker has run. Second, the semantics of each are different: linking object files into a final artifact is guaranteed to preserve global constructors. However, linking to a static library will strip symbols that are not directly referenced by the linkee, including these constructors. This is particularly annoying when trying to register a plugin to an object factory. This pitfall does not apply to shared libraries, but then the target platform must support shared libraries to begin with (not guaranteed on embedded platforms) and on other platforms, the cost of the library loader increases. Finally, whole-program or link-time optimizations work better with object files.

Dependency specification

When source files change, the build system needs to be able to bring dependent targets up to date. In the simplest case, this is a function only of the command line arguments to the DSL compiler. For instance, lex and yacc accept an input file argument and an output path argument that fully determines the dependency edges. This is good.

Other languages are more complex: Fortran (and now C++20) has a *module* system that requires scanning the source code in order to determine a language-level module graph which must be communicated to the build system. Again, this is highly build-system specific, but one popular approach is implemented in Ninja[103]. This involves running a separate command that writes a list of dependencies to a file; ninja runs this command and remembers the output when bringing a target up to date. Note that higher-order versions of this are possible, but inadvisable.

A simpler case of discovered dependencies can be found in C and LaTeX, which both have features for including another file directly into the token stream. This differs from above because the dependencies can be determined while the compiler is running, not in a separate discovery pass. In these cases, it is a *de facto* standard to write out lists of discovered dependencies in Makefile format to a file with a `.d` extension alongside the normal compiler output.

Staging

Compiling a Halide program involves two calls to a C++ compiler: one to create a so-called "generator" executable, which acts as the compiler, and another to link the final executable or library containing the emitted object. This structure is called *staging*: one program is written that must be run to produce another.

Staging has many successful implementations in other contexts, including Terra in Lua [35] and LMS in Scala [136]. However, experience suggests it is a poor fit for C++. In cross-compiling scenarios, two separate C++ toolchains (or at least separate and mutually incompatible configurations of the same toolchain) are needed, one for each of the steps above (generate and link). This is a serious issue for some build systems. CMake has a deeply ingrained assumption that a single toolchain is active; therefore Halide projects using CMake must be built twice. We provide helpers to avoid redundant work and to make this easier, but engineering them was challenging. Makefiles, too, have only a single set of variables to control the compiler selection and no de-facto standards have emerged. Meson does maintain a separate notion of a "host" toolchain that suffices for Halide's purposes, but Meson does not provide any means of abstracting build rules for custom languages and so is a poor fit for DSLs.

Chapter 6

Related Work

This thesis is rooted in a great body of prior art on compiler construction, formal semantics, language runtimes, and performance engineering. This chapter presents a brief tour of this work to place this thesis in context.

6.1 User-schedulable languages

The idea of explicit control over compiler transformations developed earlier in many script- or pragma-based compiler tools from the high-performance computing space, including Xlang[38], Sequoia[42], POET[170], Orio[70], OpenMP[31], and CHiLL[19]. The definition of parametric spaces of optimizations, which lies at the heart of user-scheduling, was originally introduced by SPIRAL[48].

The computational and scheduling models of Halide evolved through a series of extensions and generalizations[128, 129, 130, 143] before they were formalized in this work. Halide's algorithm language is closely related to both array languages and image processing DSLs such as Popi[76] and Shantzis[139]. Notable array languages include APL[82] and ZPL[18], as well as more recent developments such as Chapel[17], Accelerate[16] (for Haskell), and Futhark[74]. Halide's computational model is most closely related to that of the lazy functional image language Pan[39]. Bounds inference is related to array shape analyses and type systems [73, 83, 84]. Our treatment of bounds inference is (to the best of our knowledge) the first formulation via a constraint-based program synthesis problem [59].

A growing family of high performance DSLs since the introduction of Halide have directly adopted the concept of a programmer-visible scheduling language, including Legion[9], TVM[22], TACO[86, 156], GraphIt[172], SWIRL[157], FireIron[62], and Taichi[77].

The polyhedral loop optimization community has explored user-scheduling in its own context, in such systems as PENCIL[5], URUK[27, 28, 54], CHiLL[19], and Tiramisu[6]. ISL[158] is a reusable system for manipulating integer sets and polyhedral program schedules; its internal representation of schedules as "schedule trees"[159] is similar to Halide's original conception of schedules.

Egg[163], ELEVATE[61], and the X language[38] all provide generic transformation or rewriting infrastructure. These could be used to implement the mechanism of a scheduling language like Exo's, but do not provide the definitions and metatheory needed to establish correctness for it or any specific language.

6.2 Program analysis

Virtually all of these languages and systems do not have formally specified semantics, proofs of soundness, or other such metatheory. POET[170] and TeML[144] are notable exceptions for being defined formally, but their scheduling or transformation languages are not shown to be correctness-preserving. Legion defined a core calculus and proved a form of soundness for their dynamic, user-configurable distributed scheduler[150]. However, many of the details are unnecessary for formalizing Halide, and redundant recomputation and overcomputation on uninitialized values, both essential to Halide, remain outside their scope.

The correctness of many compiler transformations has been treated in the context of verified compilers like CompCert [14, 140, 151]. The closest component to the present work is the CompCert instruction-scheduling optimization, which is designed to be applied after register allocation. (By contrast, we are concerned with less local and harder-to-validate loop transformations.) Verification is based on the *translation validation* strategy, where a certified validator program attempts to prove that the pre- and post-optimization programs are equivalent. This strategy is effective in the CompCert scenario because (a) it is (potentially) generic with respect to the choice of optimization pass and (b) when validation fails, CompCert can always (correctly) fall back to a less optimized version of the code. Once scheduling is exposed to the user (our scenario), these design choices are inappropriate. The semantics must make predictable and defensible guarantees to users about the results of schedules that they write.

Concurrent work by Newcomb et al. [116] uses program synthesis to build a verified term-rewriting expression simplifier for the Halide expression language. Their verification conditions are based on the expression language semantics described in this work. More concurrent work by Clément and Cohen [25] applies translation validation to an affine subset of Halide, and can verify individual compiler outputs. However, this goal differs from the present work, which attempts to verify a full formal specification of Halide.

Exo builds on attempts to formalize guarantees of safety and equivalence under scheduling in Halide[133]. In sharp contrast to Halide, Exo adopts the approach of implementing scheduling via algebraic rewrites within a core language. While prior systems which follow this approach work mostly on restricted functional languages, where equivalence before and after rewrites is straightforward (and often not formally checked) [61, 100, 141], Exo rewrites *imperative* code, and relies on effect analyses which reduce to SMT for verification.

Exo's framework for verifying equivalence and safety builds on several threads from type systems and dependence analysis. Dependently-typed arrays, especially as adapted in the formalization of Halide, inform Exo's treatment of memory safety[83, 84, 165]. Dependence analysis, especially on static control programs, forms a common basis for reasoning about the safety of loop transformations [43, 47].

6.2.1 Polyhedral analysis

When combined with reasoning about affine indexing, this is the basis of polyhedral compilation[44]. CHiLL and Tiramisu defer correctness claims to polyhedral dependence analysis using ISL[158]. As will be discussed in §2.3, dependence analysis is only sufficient to justify *re-ordering* transformations—not transformations such as Halide's `compute-at`, which *recompute* or *over-compute* values and might introduce novel statement instances. For instance, in correspondence with authors of the Tiramisu paper and system[30], we discovered that the relevant safety checks for `compute-at` transformations had neither been implemented in the system artifact, nor described in the paper.

Older automated polyhedral analyses[43] work on static control programs with denotational or functional semantics. In that setting, dataflow and dependence graphs are equivalent. This is also the case for functional DSLs such as PolyMage[114], which also supports redundant recomputation. Recent developments in the Alpha system[171] are notable for maintaining a complete denotational form of the program throughout transformation, not just a dependence analysis. As with Halide, functional semantics are crucial for reasoning about such non-reordering code transformations.

6.2.2 Effect types

In contrast, Exo's approach builds on effect types, as proposed by Gifford and Lucassen[53]. While these approaches are distinct, the earliest foundations of dependencies for program parallelization define conditions on read and write sets closely related to our effect analyses[13].

Despite this difference, Exo can be seen as a polyhedral compiler, in the sense that it is built on linear integer arithmetic and static control programs. However, the program analysis used in Exo goes beyond what is normally called "polyhedral analysis" in two respects: mutable control state (for which we must rely on an approximating symbolic dataflow analysis §3.5.3), and justifying code deletion/insertion (§3.5.7 and 3.6.2). Both of these phenomena are necessary to support scheduling of hardware accelerators that make use of configuration state. They also force us to adopt ternary logic at the base of our program analysis in order to safely propagate the dataflow approximations. If configuration state were eliminated, Exo would more closely resemble traditional polyhedral compilers focused purely on reordering statement instances.

6.3 Instruction selection

Exo's instruction/procedure mapping mechanism is related to the classic problem of instruction selection[2]. Traditional instruction selection applies local pattern matching rules to replace small IR fragments with equivalent instructions, but this struggles to effectively exploit accelerator instructions which correspond to large, complex program fragments. Recent work applies more powerful search techniques to target more complex SIMD instructions using program synthesis[124] and equality saturation[155]. Exo allows substitution of much larger program fragments with arbitrary equivalent procedures, under explicit programmer control, and allows these substitutions to be interleaved with further scheduling transformations rather than confined to the compiler backend. TVM provides a related "tensorization" directive for replacing loop fragments with instructions asserted as equivalent[22], but it lacks the combination of automation and checking provided by Exo's unification procedure.

6.4 Reference counting systems

Perceus is closely based on the reference counting algorithm in the Lean theorem prover as described by Ullrich and de Moura [153]. They describe reuse analysis based on `reset`/`reuse` instructions, and describe both reference counting based on ownership (i.e. precise) but also support borrowed parameters. We extend their work with drop- and reuse specialization, and generalize to a general purpose language with side-effects and complex control flow. We also introduce a novel formalization of reference counting with the linear resource calculus, and define our algorithm in terms of that. As such, the Perceus algorithm may differ from Lean's as that is specified over a lower-level calculus that uses explicit partial application nodes (`pap`) and has no first-class lambda expressions. Schulte [138] describes an algorithm for inserting reference count instructions in a small first-order language and shows a limited form of reuse analysis, called "reusage" (transformation T14).

Using explicit reference count instructions in order to optimize them via static analysis is described as early as Barth [8]. Mutating unique references in place has traditionally focused on array updates [78], as in functional array languages like Sisal [108] and SaC [57, 137]. Férey and Shankar [46] provide functional array primitives that use in-place mutation if the array has a unique reference; we plan to add these to Koka. We believe this would work especially well in combination with reuse-analysis for BTree-like structures using trees of small functional arrays.

The λ^1 calculus is closely based on linear logic. Turner and Wadler [152] give a heap-based operational interpretation which does not need reference counts as linearity is tracked by the type system. In contrast, Chirimar, Gunter, and Riecke [23] give an interpretation of linear logic in terms of reference counting, but in their system, values with a linear type are not guaranteed to have a unique reference at runtime.

Generally, a system with linear types [161], like linear Haskell [12], or the uniqueness typing of Clean [7, 160], can offer *static* guarantees that the corresponding objects are unique at runtime, so that destructive updates can always be performed safely. However, this usually also requires writing multiple versions of a function for each case (unique versus shared argument). By contrast, reuse analysis relies on dynamic runtime information, and thus reuse can be performed generally. This is also what enables FBIP to use a single function that can be used for both unique or shared objects (since the uniqueness property is *not* part of the type). These two mechanisms could be combined: if our system is extended with unique types, then reuse analysis could statically eliminate corresponding uniqueness checks.

The Swift language is widely used in iOS development and uses reference counting with an explicit representation in its intermediate language. There is no reuse analysis but, as remarked by Ullrich and de Moura [153], this may not be so important for Swift as typical programs mutate objects in-place. There is no cycle collection for Swift, but despite the widespread usage of mutation this seems to be not a large problem in practice. Since it can be easy to create accidental cycles through the `self` pointer in callbacks, Swift has good support for *weak* references to break such cycles in a declarative manner. Ungar, Grove, and Franke [154] optimize atomic reference counts by tagging objects that can be potentially thread-shared. Later work by Choi, Shull, and Torrellas [24], uses *biased* reference counting to avoid many atomic updates.

The CPython implementation also uses reference counting, and uses ownership-based reference counts for parameters but still only drops the reference count of local variables when exiting the frame. Another recent language that uses reference counting is Nim. The reference counting method is scope-based and uses non-atomic operations (and objects cannot be shared across threads without extra precautions). Nim can be configured to use ORC reference counting which extends the basic ARC collector with a cycle collection [169]. Nim has the `acyclic` annotation to identify data types that are (co)-inductive, as well as the (unsafe) `cursor` annotation for variables that should not be reference counted.

In our work we focus on *precise* and *garbage free* reference counting which enables static optimization of reference count instructions. On the other extreme, Deutsch and Bobrow [34] consider *deferred* reference counting—any reference count operations on stack-based local variables are *deferred* and only the reference counts of fields in the heap are maintained. Much like a tracing collector, the stack roots are periodically scanned and deferred reference counting operations are performed. Levanoni and Petrank [99] extend this work and present a high performance reference counting collector for Java that uses the *sliding view* algorithm to avoid many intermediate reference counting operations and needs no synchronization on the write barrier.

Chapter 7

Conclusion

7.1 Impact

This thesis advances the design and implementation of three languages: Halide, Exo, and Koka. It provides a framework for understanding the metatheory of user-schedulable languages, rooted in Halide's design, and refines that framework in the design of Exo. The thesis addresses semantic issues related to program transformations that alter the set of statement instances, globally visible hardware configuration state, and object lifetimes, and develops practical solutions for instruction selection for accelerator hardware and reference counting efficiency.

Building a formal model of Halide identified and resolved many important design issues and made the practical system more stable. It has informed other ongoing efforts to formalize parts of Halide, including its expression simplification and bounds inference systems. We are currently working with industry partners to further Exo's development and meet the needs of the next generation of accelerator hardware.

I am particularly optimistic about Perceus's impact. It is already being used as the core memory management system for new programming languages, including Roc [45], a language under development by members of the Elm community. Perceus was also cited in the keynote talk at the International Symposium on Memory Management as a particularly promising new research direction [15]. The reuse system in Perceus was further advanced by Lorenzen and Leijen [101].

7.2 Future work

There are many exciting research directions for future work. We are most excited about exploring parts of the user-schedulable language design space in between Halide and Exo: in particular, the phase ordering from the Halide formalism suggests that a similar decomposition could be useful in the exocompilation settings. For example, one could imagine designing a multi-level algorithm language where certain high-level features are eliminated at certain user-controlled stages. The scheduling language could operate on these features at a higher level while they are still available. This appears to be a particularly promising approach for handling tail cases in tiling and vectorization, both of which Exo currently struggles with.

There is much work to be done designing more expressive scheduling languages. One ongoing project is to have the scheduling directives operate on *cursors*, which are *rewrite-stable* pointers into a procedure. A cursor into a procedure can be *forwarded* to another procedure that was derived from it. This forwarding operation might be subject to certain semantic or structural constraints. For instance, it might be desirable to have the forwarding operation be homomorphic over the dependent effect type of the selected code fragment. On the other hand, this might be too restrictive: perhaps certain transformations care more about the structure of the selection than its meaning.

Exo's instruction selection features might be possible to retrofit into Halide, at least at the level of defining new intrinsics as libraries and scheduling these intrinsics into algorithm expressions via the scheduling language. A system of proxy expressions already exists for computing the bounds of external Halide func definitions; it might be possible to generalize this to entire funcs, at least when the bounds are affine.

Bibliography

[1] Andrew Adams et al. “Learning to optimize halide with tree search and random programs”. In: *ACM Trans. Graph.* 38.4 (2019), 121:1–121:12. DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, 2006.

[3] Kevin Amaratunga. *Makefile Primer*. 2000. URL: https://ocw.mit.edu/courses/1-124j-foundations-of-software-engineering-fall-2000/pages/lecture-notes/makefile_primer/.

[4] Unnamed Teaching Assistant. *A Quick Introduction to Makefiles*. 1998. URL: https://courses.cs.washington.edu/courses/cse451/98sp/Section/makeintro.html.

[5] Riyadh Baghdadi et al. “PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming”. In: *PACT*. San Francisco, CA, USA: IEEE Computer Society, 2015, pp. 138–149. DOI: 10.1109/PACT.2015.17.

[6] Riyadh Baghdadi et al. “Tiramisu: a Polyhedral Compiler for Expressing Fast and Portable Code”. In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019*. Washington, DC, USA: IEEE, 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197. URL: https://doi.org/10.1109/CGO.2019.8661197.

[7] Erik Barendsen and Sjaak Smetsers. “Uniqueness typing for functional languages with graph rewriting semantics”. In: *Mathematical Structures in Computer Science* 6.6 (1996), pp. 579–612. DOI: 10.1017/S0960129500070109.

[8] Jeffrey M. Barth. *Shifting Garbage Collection Overhead to Compile Time*. Tech. rep. UCB/ERL M524. EECS Department, University of California, Berkeley, June 1975. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1975/29109.html.

[9] Michael Bauer et al. “Legion: expressing locality and independence with logical regions”. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12*. Salt Lake City, UT, USA: IEEE, Nov. 2012, p. 66. DOI: 10.1109/SC.2012.71. URL: https://doi.org/10.1109/SC.2012.71.

[10] *Bazel*. Accessed: 2022-11-03. URL: https://bazel.build/.

[11] Emery D. Berger. "Scalene: Scripting-Language Aware Profiling for Python". In: (June 2020). DOI: 10.48550/arxiv.2006.03879. URL: https://arxiv.org/abs/2006.03879v2.

[12] Jean-Philippe Bernardy et al. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093.

[13] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[14] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. "A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis". In: *Types for Proofs and Programs*. Ed. by Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 66–81. ISBN: 978-3-540-31429-5. DOI: 10.1007/11617990_5. URL: http://xavierleroy.org/publi/proofs_dataflow_optimizations.pdf.

[15] Steve Blackburn. *We Live in Interesting Times*. YouTube. 2022. URL: https://youtu.be/HdC01ApFaa0?t=3025.

[16] Manuel M. T. Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming*. Ed. by Manuel Carro and John H. Reppy. New York, NY, USA: Association for Computing Machinery, 2011, pp. 3–14. DOI: 10.1145/1926354.1926358. URL: https://doi.org/10.1145/1926354.1926358.

[17] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[18] Bradford L. Chamberlain. "The design and implementation of a region-based parallel programming language". PhD thesis. The University of Washington, 2001.

[19] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Southern California, 2008.

[20] Raymond Chen. *Undefined behavior can result in time travel (among other things, but time travel is the funkiest)*. The Old New Thing. June 27, 2014. URL: https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633 (visited on 11/26/2022).

[21] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 2018, pp. 3393–3404. URL: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.

[22] Tianqi Chen et al. "TVM: An Automated End-to-end Optimizing Compiler for Deep Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291211.

[23] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. "Reference Counting as a Computational Interpretation of Linear Logic". In: *Journal of Functional Programming* 6 (1996), pp. 6–2.

[24] Jiho Choi, Thomas Shull, and Josep Torrellas. "Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus, 2018. DOI: 10.1145/3243176.3243195.

[25] Basile Clément and Albert Cohen. "End-to-End Translation Validation for the Halide Language". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527328. URL: https://doi.org/10.1145/3527328.

[26] *CMake*. Accessed: 2022-11-03. URL: https://cmake.org/.

[27] A. Cohen, N. Vasilache, and L. Pouchet. "Automatic Correction of Loop Transformations". In: *2007 16th International Conference on Parallel Architectures and Compilation Techniques*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2007, pp. 292–304. DOI: 10.1109/PACT.2007.17. URL: https://doi.ieeecomputersociety.org/10.1109/PACT.2007.17.

[28] Albert Cohen et al. "Facilitating the Search for Compositions of Program Transformations". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: Association for Computing Machinery, 2005, pp. 151–160. ISBN: 1595931678. DOI: 10.1145/1088149.1088169. URL: https://doi.org/10.1145/1088149.1088169.

[29] George E Collins. "A method for overlapping and erasure of lists". In: *Communications of the ACM* 3.12 (1960), pp. 655–657.

[30] Tiramisu Contributors. *Does Tiramisu have legality check of scheduling? • Issue #300 • Tiramisu-Compiler/tiramisu*. June 2019. URL: https://github.com/Tiramisu-Compiler/tiramisu/issues/300.

[31] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[32] Ankush Desai and Shaz Qadeer. "P: Modular and Safe Asynchronous Programming". In: *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. Ed. by Shuvendu K. Lahiri and Giles Reger. Vol. 10548. Lecture Notes in Computer Science. Springer, 2017, pp. 3–7. DOI: 10.1007/978-3-319-67531-2_1. URL: https://doi.org/10.1007/978-3-319-67531-2_1.

[33] David L. Detlefs et al. "Lock-Free Reference Counting". In: *in Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. 2001, pp. 190–199.

[34] L. Peter Deutsch and Daniel G. Bobrow. "An Efficient, Incremental, Automatic Garbage Collector". In: *Communications of the ACM* 19.9 (Sept. 1976), pp. 522–526. ISSN: 0001-0782. DOI: 10.1145/360336.360345.

[35] Zachary DeVito et al. "Terra: A Multi-Stage Language for High-Performance Computing". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 105–116. ISSN: 0362-1340. DOI: 10.1145/2499370.2462166. URL: https://doi.org/10.1145/2499370.2462166.

[36] Will Dietz et al. "Understanding Integer Overflow in C/C++". In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015). ISSN: 1049-331X. DOI: 10.1145/2743019. URL: https://doi.org/10.1145/2743019.

[37] Damien Doligez and Xavier Leroy. "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML". In: *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*. ACM press, Jan. 1993, pp. 113–123.

[38] Sébastien Donadio et al. "A Language for the Compact Representation of Multiple Program Versions". In: *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 136–151. DOI: 10.1007/978-3-540-69330-7_10. URL: https://doi.org/10.1007/978-3-540-69330-7_10.

[39] Conal Elliott. *Functional Image Synthesis*. 2001. URL: http://conal.net/papers/bridges2001/.

[40] Princeton University Faculty. *Building Multifile Programs with make*. 2003. URL: https://www.cs.princeton.edu/courses/archive/spr21/cos217/lectures/07_Building.pdf.

[41] Yale University Faculty. *How To Create A Makefile*. Jan. 2020. URL: https://zoo.cs.yale.edu/classes/cs223/doc/Makefile.

[42] Kayvon Fatahalian et al. "Sequoia: Programming the Memory Hierarchy". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 83–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188543. URL: https://doi.org/10.1145/1188455.1188543.

[43] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *Int. J. Parallel Program.* 20.1 (1991), pp. 23–53. DOI: 10.1007/BF01407931. URL: https://doi.org/10.1007/BF01407931.

[44] Paul Feautrier and Christian Lengauer. "The Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011, pp. 1581–1592. DOI: 10.1007/978-0-387-09766-4_502.

[45] Richard Feldman. *Outperforming Imperative with Pure Functional Languages*. YouTube. 2021. URL: https://youtu.be/vzfy4EKwG_Y?t=384.

[46] Gaspard Férey and Natarajan Shankar. "Code Generation Using a Formal Model of Reference Counting". In: *NASA Formal Methods*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Springer International Publishing, 2016, pp. 150–165. ISBN: 978-3-319-40648-0.

[47] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041.

[48] Franz Franchetti et al. "SPIRAL: Extreme Performance Portability". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1935–1968. DOI: 10.1109/JPROC.2018.2873289. URL: https://doi.org/10.1109/JPROC.2018.2873289.

[49] Free Software Foundation, Silicon Graphics, and Hewlett–Packard Company. "Internal red-black tree implemention for "stl::map"". URL: https://code.woboq.org/gcc/libstdc++-v3/src/c++98/tree.cc.html.

[50] Matt Gallagher. "Reference Counted Releases in Swift". Blog post. Dec. 2016. URL: https://www.cocoawithlove.com/blog/resources-releases-reentrancy.html.

[51] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[52] A. Gidenstam et al. "Efficient and reliable lock-free memory reclamation based on reference counting". In: *8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN'05)*. 2005.

[53] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 28–38. ISBN: 0897912004. DOI: 10.1145/319838.319848. URL: https://doi.org/10.1145/319838.319848.

[54] Sylvain Girbal et al. "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies". In: *Int. J. Parallel Program.* 34.3 (2006), pp. 261–317. DOI: 10.1007/s10766-006-0012-3. URL: https://doi.org/10.1007/s10766-006-0012-3.

[55] Kiran Gopinathan. *Goodbye C developers: The future of programming with certified program synthesis*. URL: https://gopiandcode.uk/logs/log-certified-synthesis.html (visited on 11/26/2022).

[56] Chris Gregg, Kevin Montag, and Nick Troccoli. *Compiling Programs with Make*. 2019. URL: https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/make.

[57] Clemens Grelck and Kai Trojahner. "Implicit Memory Management for SAC". In: *6th International Workshop on Implementation and Application of Functional Languages (IFL'04)*. Lübeck, Germany, Sept. 2004.

[58] Leo J Guibas and Robert Sedgewick. "A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE. 1978, pp. 8–21.

[59] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010. URL: http://dx.doi.org/10.1561/2500000010.

[60] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. "A Generalization of Exceptions and Control in ML-like Languages". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 12–23. ISBN: 0897917197. DOI: 10.1145/224164.224173.

[61] Bastian Hagedorn et al. *A Language for Describing Optimization Strategies*. 2020. arXiv: 2002.02268 [cs.PL].

[62] Bastian Hagedorn et al. *Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs*. 2020. arXiv: 2003.06324 [cs.PL].

[63] Halide Contributors. *Accesses inside indexing expressions should be in compute bounds • Issue #6131 • halide/Halide*. July 2021. URL: https://github.com/halide/Halide/issues/6131.

[64] Halide Contributors. *Check RDom::where predicates for race conditions • Pull Request #6842 • halide/Halide*. July 2022. URL: https://github.com/halide/Halide/pull/6842.

[65] Halide Contributors. *Directly realizing a func with an RDom aborts when realization does not contain RDom bounds. • Issue #3883 • halide/Halide*. May 2019. URL: https://github.com/halide/Halide/issues/3883.

[66] Halide Contributors. *Fix floated pure stage • Issue #3947 • halide/Halide*. June 2019. URL: https://github.com/halide/Halide/issues/3947.

[67] Halide Contributors. *RoundUp behavior on integer funcs can cause arithmetic exceptions • Issue #4423 • halide/Halide*. Nov. 2019. URL: https://github.com/halide/Halide/issues/4423.

[68] Halide Contributors. *The check for race conditions doesn't consider where clauses • Issue #6808 • halide/Halide*. June 2022. URL: https://github.com/halide/Halide/issues/6808.

[69] Halide Contributors. *What does it mean to have an RDom with a negative extent? • Issue #4385 • halide/Halide*. Nov. 2019. URL: https://github.com/halide/Halide/issues/4385.

[70] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. "Annotation-based empirical performance tuning using Orio". In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. Rome, Italy: IEEE, 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161004. URL: https://doi.org/10.1109/IPDPS.2009.5161004.

[71] Samuel W. Hasinoff et al. "Burst photography for high dynamic range and low-light imaging on mobile cameras". In: *ACM Trans. Graph.* 35.6 (2016), 192:1–192:12. URL: http://dl.acm.org/citation.cfm?id=2980254.

[72] *Haxe*. Accessed: 2022-11-24. URL: https://haxe.org/.

[73] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. "Size Slicing: A Hybrid Approach to Size Inference in Futhark". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '14. Gothenburg, Sweden: ACM, 2014, pp. 31–42. ISBN: 978-1-4503-3040-4. DOI: 10.1145/2636228.2636238. URL: http://doi.acm.org/10.1145/2636228.2636238.

[74] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[75] Paul N. Hilfinger. *Basic Compilation Control with GMake*. 1998. URL: https://people.eecs.berkeley.edu/~jrs/61bf98/reader/ucb/gmake.pdf.

[76] Gerard Holzmann. *Beyond Photography: The Digital Darkroom*. Upper Saddle River, NJ, USA: Prentice Hall, 1988.

[77] Yuanming Hu et al. "Taichi: a language for high-performance computation on spatially sparse data structures". In: *ACM Trans. Graph.* 38.6 (2019), 201:1–201:16. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

[78] Paul Hudak and Adrienne Bloss. "The Aggregate Update Problem in Functional Programming Systems". In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '85. New Orleans, Louisiana, USA: ACM, 1985, pp. 300–314. ISBN: 0897911474. DOI: 10.1145/318593.318660.

[79] Gérard P. Huet. "The Zipper". In: *Journal of Functional Programming* 7.5 (1997), pp. 549–554.

[80] Yuka Ikarashi et al. "Exocompilation for productive programming of hardware accelerators". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 703–718. DOI: 10.1145/3519939.3523446. URL: https://doi.org/10.1145/3519939.3523446.

[81] Apple Inc. "The Swift Guide: Error Handling". 2017. URL: https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html.

[82] Kenneth E. Iverson. *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 0-471430-14-5.

[83] C. B. Jay and P. A. Steckler. "The functional imperative: Shape!" In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 139–153. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053568.

[84] C. Barry Jay and Milan Sekanina. *Shape Checking of Array Programs*. Tech. rep. Sydney, Australia: In Computing: the Australasian Theory Seminar, Proceedings, 1997.

[85] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558602860.

[86] Fredrik Kjolstad et al. "The tensor algebra compiler". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–29. DOI: 10.1145/3133901. URL: https://doi.org/10.1145/3133901.

[87] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[88] "Koka repository". 2019. URL: https://github.com/koka-lang/koka.

[89] Chris Lattner. *What Every C Programmer Should Know About Undefined Behavior #1/3*. Section: posts. May 13, 2011. URL: https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html (visited on 11/26/2022).

[90] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[91] Michael Lauer. *Introducing Vala Programming*. Apress, 2019. DOI: 10.1007/978-1-4842-5380-9. URL: https://doi.org/10.1007/978-1-4842-5380-9.

[92] Juneyoung Lee et al. “Taming Undefined Behavior in LLVM”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 633–647. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062343. URL: http://doi.acm.org/10.1145/3062341.3062343.

[93] Daan Leijen. *Algebraic Effects for Functional Programming*. Tech. rep. MSR-TR-2016-29. Extended version of [96]. Microsoft Research technical report, Aug. 2016.

[94] Daan Leijen. “Koka: Programming with Row Polymorphic Effect Types”. In: *MSFP’14, 5th workshop on Mathematically Structured Functional Programming*. 2014. DOI: 10.4204/EPTCS.153.8.

[95] Daan Leijen. “Structured Asynchrony with Algebraic Effects”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2017. Oxford, UK, 2017, pp. 16–29. ISBN: 978-1-4503-5183-6. DOI: 10.1145/3122975.3122977.

[96] Daan Leijen. “Type Directed Compilation of Row-typed Algebraic Effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*. Paris, France, Jan. 2017, pp. 486–499. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009872.

[97] Daan Leijen, Zorn Ben, and Leo de Moura. “Mimalloc: Free List Sharding in Action”. In: *Programming Languages and Systems*. LNCS 11893 (2019). APLAS’19. DOI: 10.1007/978-3-030-34175-6_13.

[98] Daan Leijen and Anton Lorenzen. *Tail Recursion Modulo Context – An Equational Approach*. MSR-TR-2022-18. Microsoft, July 2022. URL: https://www.microsoft.com/en-us/research/publication/tail-recursion-modulo-context-an-equational-approach/.

[99] Yossi Levanoni and Erez Petrank. “An On-the-Fly Reference-Counting Garbage Collector for Java”. In: *ACM Trans. Program. Lang. Syst.* 28.1 (Jan. 2006), pp. 1–69. ISSN: 0164-0925. DOI: 10.1145/1111596.1111597.

[100] Amanda Liu et al. “Verified Tensor-Program Optimization Via High-level Scheduling Rewrites”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498717.

[101] Anton Lorenzen and Daan Leijen. “Reference counting with frame limited reuse”. In: *Proceedings of the ACM on Programming Languages* 6 (ICFP Aug. 31, 2022), 103:357–103:380. DOI: 10.1145/3547634. URL: https://doi.org/10.1145/3547634 (visited on 11/26/2022).

[102] David Malan. *Speller*. 2022. URL: https://cs50.harvard.edu/college/2022/fall/psets/5/speller/%5C#makefile.

[103] Evan Martin. *Ninja, a small build system with a focus on speed*. 2012. URL: https://ninja-build.org/.

[104] Prabhaker Mateti and Ravi Manghirmalani. "Morris' tree traversal algorithm reconsidered". In: *Science of Computer Programming* 11.1 (1988), pp. 29–43. ISSN: 0167-6423. DOI: 10.1016/0167-6423(88)90063-9.

[105] Niko Matsakis. *Introducing MIR*. Apr. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html.

[106] Conor McBride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. (Extended Abstract). 2001. URL: http://strictlypositive.org/diff.pdf.

[107] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[108] J. McGraw et al. *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1. 1.* Tech. rep. LLL/M-146, ON: DE83016576. CA, USA: Lawrence Livermore National Lab., July 1983.

[109] Maged M. Michael. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects". In: *IEEE Trans. Parallel Distrib. Syst.* 15.6 (June 2004), pp. 491–504. DOI: 10.1109/TPDS.2004.8.

[110] Peter Miller. "Recursive Make Considered Harmful". In: *AUUGN Journal of AUUG Inc* 19 (1 1998), pp. 14–25.

[111] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. 2012. ISBN: 978-1449323912. URL: https://dev.realworldocaml.org.

[112] *MLton*. Accessed: 2022-11-24. URL: http://mlton.org/.

[113] Joseph M. Morris. "Traversing binary trees simply and cheaply". In: *Information Processing Letters* 9.5 (1979), pp. 197–200. DOI: https://doi.org/10.1016/0020-0190(79)90068-1.

[114] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 429–443. ISBN: 9781450328357. DOI: 10.1145/2694344.2694364. URL: https://doi.org/10.1145/2694344.2694364.

[115] Ravi Teja Mullapudi et al. "Automatically Scheduling Halide Image Processing Pipelines". In: *ACM Trans. Graph.* 35.4 (2016), 83:1–83:11. DOI: 10.1145/2897824.2925952. URL: https://doi.org/10.1145/2897824.2925952.

[116] Julie L. Newcomb et al. "Verifying and Improving Halide's Term Rewriting System with Program Synthesis". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428234. URL: https://doi.org/10.1145/3428234.

[117] *Nim programming language*. Accessed: 2022-11-24. URL: https://nim-lang.org/.

[118] Chris Okasaki. *Purely Functional Data Structures*. New York: Colombia University, June 1999. ISBN: 9780521663502.

[119] Chris Okasaki. "Red-black trees in a functional setting". In: *Journal of Functional Programming* 9.4 (1999), pp. 471–477. DOI: 10.1017/S0956796899003494.

[120] Everi Osofsky. *Using Makefiles to Compile Code*. Nov. 2017. URL: https://www.cs.cmu.edu/~07131/f22/topics/makefiles/compiling-code/.

[121] Adam Paszke et al. "Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473593. URL: https://doi.org/10.1145/3473593.

[122] Andrew J. Pershing. *How to write a Makefile*. Jan. 2003. URL: https://www.cs.cornell.edu/courses/cs403/2003sp/Lecture06/makefile.html.

[123] Simon L. Peyton Jones and André L. M. Santos. "A transformation-based optimiser for Haskell". In: *Science of Computer Programming* 32.1 (1998), pp. 3–47. DOI: http://dx.doi.org/10.1016/S0167-6423(97)00029-4.

[124] Phitchaya Mangpo Phothilimthana et al. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 65–78. ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. URL: https://doi.org/10.1145/3297858.3304059.

[125] Gordon D. Plotkin and John Power. "Algebraic Operations and Generic Effects". In: *Applied Categorical Structures* 11.1 (2003), pp. 69–94. DOI: 10.1023/A:1023064908962.

[126] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: vol. 9. 4. 2013. DOI: 10.2168/LMCS-9(4:23)2013.

[127] Riccardo Pucella. *Lecture 7: Introduction to Make*. Mar. 2002. URL: https://www.cs.cornell.edu/courses/cs214/2002sp/lect7.pdf.

[128] Jonathan Ragan-Kelley et al. "Decoupling algorithms from schedules for easy optimization of image processing pipelines". In: *ACM Trans. Graph.* 31.4 (2012), 32:1–32:12. DOI: 10.1145/2185520.2185528. URL: https://doi.org/10.1145/2185520.2185528.

[129] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[130] Jonathan Ragan-Kelley et al. "Halide: decoupling algorithms from schedules for high-performance image processing". In: *Commun. ACM* 61.1 (2018), pp. 106–115. DOI: 10.1145/3150211. URL: https://doi.org/10.1145/3150211.

[131] Jason Redgrave et al. “Pixel Visual Core: Google’s Fully Programmable Image, Vision, and AI Processor For Mobile Devices”. In: *2018 IEEE Hot Chips 30 Symposium (HCS)*. Cupertino, CA, USA: IEEE, Aug. 2018, pp. 1–28.

[132] John Regehr. *A Guide to Undefined Behavior in C and C++, Part 1 – Embedded in Academia*. July 9, 2010. URL: https://blog.regehr.org/archives/213 (visited on 11/26/2022).

[133] Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. *Formal Semantics for the Halide Language*. 2022. DOI: 10.48550/ARXIV.2210.15740. URL: https://arxiv.org/abs/2210.15740.

[134] Alex Reinking et al. *Perceus: Garbage Free Reference Counting with Reuse*. Tech. rep. MSR-TR-2020-42. Microsoft Research, Nov. 2020.

[135] Alex Reinking et al. “Perceus: garbage free reference counting with reuse”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 96–111. DOI: 10.1145/3453483.3454032. URL: https://doi.org/10.1145/3453483.3454032.

[136] Tiark Rompf and Martin Odersky. “Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs”. In: *SIGPLAN Not.* 46.2 (Oct. 2010), pp. 127–136. ISSN: 0362-1340. DOI: 10.1145/1942788.1868314. URL: https://doi.org/10.1145/1942788.1868314.

[137] Sven-Bodo Scholz. “Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting”. In: *Journal of Functional Programming* 13.6 (Nov. 2003), pp. 1005–1059. DOI: 10.1017/S0956796802004458.

[138] Wolfram Schulte. “Deriving residual reference count garbage collectors”. In: *Programming Language Implementation and Logic Programming (PLILP)*. Ed. by Manuel Hermenegildo and Jaan Penjam. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 102–116. ISBN: 978-3-540-48695-4.

[139] Michael A. Shantzis. “A Model for Efficient and Flexible Image Computing”. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’94. New York, NY, USA: Association for Computing Machinery, 1994, pp. 147–154. ISBN: 0897916670. DOI: 10.1145/192161.192191. URL: https://doi.org/10.1145/192161.192191.

[140] Cyril Six, Sylvain Boulmé, and David Monniaux. “Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428197. URL: https://doi.org/10.1145/3428197.

[141] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. “LIFT: A Functional Data-parallel IR for High-performance GPU Code Generation”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[142] Bjarne Stroustrup. “Evolving a Language in and for the Real World: C++ 1991-2006”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 4–1–4–59. ISBN: 9781595937667. DOI: 10.1145/1238844.1238848. URL: https://doi.org/10.1145/1238844.1238848.

[143] Patricia Suriana, Andrew Adams, and Shoaib Kamil. “Parallel Associative Reductions in Halide”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO ’17. Austin, USA: IEEE Press, 2017, pp. 281–291. ISBN: 978-1-5090-4931-8. URL: http://dl.acm.org/citation.cfm?id=3049832.3049863.

[144] Adilla Susungi et al. “Meta-Programming for Cross-Domain Tensor Optimizations”. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 79–92. ISBN: 9781450360456. DOI: 10.1145/3278122.3278131. URL: https://doi.org/10.1145/3278122.3278131.

[145] Herb S. Sutter. “Zero-overhead deterministic exceptions: Throwing values”. C++ open-std proposal P0709 R2. Oct. 2018.

[146] *Swift Intermediate Language (SIL)*. Accessed 11-24-2022. URL: https://apple-swift.readthedocs.io/en/latest/SIL.html.

[147] Texas Instruments. *Reading and Writing Binary Files on Targets With More Than 8-Bit Chars*. Tech. rep. July 2001. URL: https://www.ti.com/lit/an/spra757/spra757.pdf.

[148] Texas Instruments. *TMS320C55x Optimizing C/C++ Compiler User’s Guide*. Tech. rep. Dec. 2003. URL: https://www.ti.com/lit/ug/spru281f/spru281f.pdf.

[149] *The Meson Build system*. Accessed: 2022-11-03. URL: https://mesonbuild.com/.

[150] Sean Treichler, Michael Bauer, and Alex Aiken. “Language Support for Dynamic, Hierarchical Data Partitioning”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 495–514. ISBN: 9781450323741. DOI: 10.1145/2509136.2509545. URL: https://doi.org/10.1145/2509136.2509545.

[151] Jean-Baptiste Tristan and Xavier Leroy. “Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 17–27. ISBN: 9781595936899. DOI: 10.1145/1328438.1328444. URL: https://doi.org/10.1145/1328438.1328444.

[152] David N. Turner and Phillip Wadler. “Operational interpretations of linear logic”. In: 227 (1999), pp. 231–248.

[153] Sebastian Ullrich and Leonardo de Moura. “Counting Immutable Beans – Reference counting optimized for purely functional programming”. In: *Proceedings of the 31st symposium on Implementation and Application of Functional Languages (IFL’19)*. Singapore, Sept. 2019.

[154] David Ungar, David Grove, and Hubertus Franke. “Dynamic Atomicity: Optimizing Swift Memory Management”. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. DLS 2017. Vancouver, BC, Canada, 2017, pp. 15–26. DOI: 10.1145/3133841.3133843.

[155] Alexa VanHattum et al. “Vectorization for Digital Signal Processors via Equality Saturation”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. URL: https://doi.org/10.1145/3445814.3446707.

[156] Nicolas Vasilache et al. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730 [cs.PL].

[157] Anand Venkat et al. “SWIRL: High-performance many-core CPU code generation for deep neural networks”. In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1275–1289. DOI: 10.1177/1094342019866247. eprint: https://doi.org/10.1177/1094342019866247. URL: https://doi.org/10.1177/1094342019866247.

[158] Sven Verdoolaege. “isl: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49.

[159] Sven Verdoolaege et al. “Schedule Trees”. In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Ed. by Sanjay Rajopadhye and Sven Verdoolaege. Vienna, Austria: INRIA, Jan. 2014, pp. 1–9.

[160] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. "Uniqueness Typing Simplified". In: *Implementation and Application of Functional Languages (IFL'08)*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Springer, 2008, pp. 201–218. ISBN: 978-3-540-85373-2.

[161] Phillip Wadler. "Linear Types can Change the World!" In: *Programming Concepts and Methods*. 1990.

[162] Yasunari Watanabe et al. "Certifying the synthesis of heap-manipulating programs". In: *Proceedings of the ACM on Programming Languages* 5 (ICFP Aug. 19, 2021), 84:1–84:29. DOI: 10.1145/3473589. URL: https://doi.org/10.1145/3473589 (visited on 11/26/2022).

[163] Max Willsey et al. *egg: Fast and Extensible E-graphs*. 2020. arXiv: 2004.03082 [cs.PL].

[164] Andrew K. Wright and Matthias Felleisen. "A syntactic approach to type soundness". In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94. DOI: 10.1006/inco.1994.1093.

[165] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 214–227. ISBN: 1581130953. DOI: 10.1145/292540.292560. URL: https://doi.org/10.1145/292540.292560.

[166] Ningning Xie and Daan Leijen. "Effect Handlers in Haskell, Evidently". In: *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*. Haskell'20. Jersey City, NJ, Aug. 2020. DOI: 10.1145/3406088.3409022.

[167] Ningning Xie and Daan Leijen. *Generalized Evidence Passing for Effect Handlers*. Tech. rep. MSR-TR-2021-5. Microsoft Research, Mar. 2021.

[168] Ningning Xie et al. "Effect Handlers, Evidently". In: *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020)*. ICFP '20. Jersey City, NJ, Aug. 2020. DOI: 10.1145/3408981.

[169] Danil Yarantsev. "ORC - Nim's cycle collector". Oct. 2020. URL: https://nim-lang.org/blog/2020/10/15/introduction-to-arc-orc-in-nim.html.

[170] Qing Yi et al. "POET: Parameterized Optimizations for Empirical Tuning". In: *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. Rome, Italy: IEEE, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370637. URL: https://doi.org/10.1109/IPDPS.2007.370637.

[171] Tomofumi Yuki et al. "AlphaZ: A System for Design Space Exploration in the Polyhedral Model". In: *Languages and Compilers for Parallel Computing*. Ed. by Hironori Kasahara and Keiji Kimura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 17–31. ISBN: 978-3-642-37658-0. DOI: 10.1007/978-3-642-37658-0_2.

[172] Yunming Zhang et al. “GraphIt: a high-performance graph DSL”. In: *PACMPL* 2.OOPSLA (2018), 121:1–121:30. DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.

[173] Lianmin Zheng et al. “Ansor: Generating High-Performance Tensor Programs for Deep Learning”. In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI’20. 2020, pp. 863–879. DOI: 10.5555/3488766.3488815.