



NORTHWESTERN UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-44

August 2026

Preserving Data Collections in the Compiler

John Thomas “Tommy” McMichen

Abstract

Data collections — such as arrays, lists, sets, and maps — are commonplace in modern programs, yet their rich semantic information is lowered before compilers perform the bulk of their analysis and optimization. This leaves the compiler to reason about low-level memory operations using undecidable pointer analyses. In this thesis, I present MEMOIR: a general-purpose compiler intermediate representation that preserves data collections, making their logical structure explicit for analysis and optimization, and I demonstrate its benefits with novel optimizations that achieve orders-of-magnitude speedups over traditional compilers. This approach introduces two challenges. First, MEMOIR represents collection updates using value semantics. Lowering them to efficient imperative code risks copies that cause catastrophic performance regressions; I formally prove that MEMOIR programs can always be lowered without such copies by applying minor changes to existing optimizations. Second, references and iterators identify elements independently of their owning collection, which cannot be expressed with value semantics; I resolve this tension by introducing bookmark semantics.

Keywords

Compilers, Programming Languages, Computer Systems, Data Collections

NORTHWESTERN UNIVERSITY

Preserving Data Collections in the Compiler

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

John Thomas “Tommy” McMichen

EVANSTON, ILLINOIS

September 2026

© Copyright by John Thomas “Tommy” M^cMichen, 2026

All Rights Reserved

Abstract

Data collections—such as arrays, lists, sets, and maps—are among the most commonly used abstractions in modern programs, yet compilers discard their rich semantic structure before performing analysis and optimization. Once lowered to raw memory, the logical organization of a collection is lost, leaving the compiler to reason about undecidable problems, such as pointer aliasing.

I present MEMOIR: a general-purpose compiler intermediate representation (IR) that preserves data collections in static single assignment (SSA) form, making their logical structure explicit for analysis and optimization. I demonstrate the benefits of MEMOIR with novel layout-agnostic optimizations that operate directly on logical collections. Furthermore, I show how the compiler can manufacture data properties to enable specialized implementations, such as bitsets, achieving significant performance and memory savings compared to industry-standard implementations. Representing collections in SSA form makes these analyses efficient through sparse dataflow, but introduces two challenges.

First, SSA represents updates functionally. Lowering them back to imperative code risks introducing copies that turn an $O(1)$ update into a $O(n)$ operation. I formally prove that MEMOIR programs can always be lowered without such copies, and show that standard compiler transformations can be made to maintain this guarantee with 100% effectiveness and negligible overhead. Second, references and iterators, which are ubiquitous in languages like C++, identify elements independently of the collection that owns them. However, SSA value semantics have no way to express this. I introduce bookmark semantics to resolve this tension.

Acknowledgments

To my advisor, Simone Campanoni, for the years of enthusiastic support, advice and freedom that made this work possible. To my many mentors and teachers, especially Christos Dimoulas, Peter Dinda, Nikos Hardavellas, Robby Findler, Daniel Chang, Sriram Mohan, Zak Estrada, Eric Noble, Jed Wandland, Matthew Skura, and Diana Lamb.

To the many labmates who have shared this journey with me: Enrico Deiana, Brian Homerding, Hussain Khajanchi, Kasia Dunajewski, Yian Su, Nathan Greiner, Atmn Patel, Federico Sossai, Jasper Liang, David Dlott, Bangyen Pham, Haocheng Gao, and Sophia Boksenbaum. I would especially like to thank Nick Wanninger and Federico Sossai, who were always willing to talk about research, and Nathan Greiner, who was always willing to talk about anything else.

To my research advisees: Benjamin Ye, Akash Deo and Leyla Latifova, for trusting me with their time; I hope you had fun, because I sure did.

To my family, for their unconditional love, support and trust throughout my life and studies. To my sister Katie and my papaw Donald, who taught me so much in the time I had with them.

To my church family at New Hope Christian Church of Whitestown, Indiana and St. Luke's Episcopal Church of Evanston, Illinois.

Most importantly, to my Lord and Savior Jesus Christ, through whom all things are possible. For granting me peace beyond understanding through terrible loss. For giving us a world full of discovery, that we may find glimpses of God in His creation before we behold Him in full and endless sight.

“If any of you lacks wisdom, let him ask of God, who gives to all liberally and without reproach, and it will be given to him.”—James 1:5

Dedicated to my mother and father, for teaching me the joy of patient study.

Contents

1	Introduction	20
1.1	Data Collections	20
1.2	Why do Compilers Fail to Optimize Memory?	21
1.3	How to Preserve Data Collections in the Compiler?	23
2	Background	27
2.1	Static Single Assignment Form	27
2.2	Rewrite Rules	29
2.3	Dataflow Analysis	30
2.3.1	Dense Dataflow Analysis	33
2.3.2	Sparse Dataflow Analysis	34
2.3.3	Widening and Narrowing	35
2.4	Static Single Information	36
2.5	Dominance and Strict SSA Form	37
3	Representing Data Collections in an SSA Form	39
3.1	Data Collections	40
3.2	MEMOIR by Example	42
3.3	MEMOIR Syntax and Informal Semantics	48
3.4	Element-Level Sparse Dataflow Analysis	50
3.5	Decoupling Logical Organization and Layout	53
3.5.1	Layout-Agnostic Analysis	54
3.5.2	Layout Optimization	54
3.6	Construction	56

	8
3.7	Destruction 59
3.8	MEMOIR Extensions 62
3.8.1	Bookmarks 62
3.8.2	Iterators 63
3.8.3	Implementation Annotations 65
3.9	Related Work 67
3.9.1	Virtual SSA 67
3.9.2	Aggregate SSA 68
3.9.3	Relations 71
3.9.4	Locality 72
3.9.5	Typed Intermediate Languages 73
3.10	Discussion 74
4	Optimizing Logical Organization 76
4.1	Live Element Analysis and Dead Element Elimination 77
4.1.1	Dead Field Elimination 78
4.1.2	Symbolic Range Analysis 79
4.1.3	Lineage Functions 80
4.1.4	Live Element Analysis, Formally 81
4.1.5	Dead Element Elimination, Formally 84
4.2	Field Elision 86
4.3	Redundant Indirection Elimination 88
4.4	Evaluation 89
4.4.1	Experimental Setup 90
4.5	Discussion 93
5	Unlocking Specialized Collection Implementations 97
5.1	Automatic Data Enumeration, Informally 100

5.2	Automatic Data Enumeration, Formally	102
5.2.1	Inserting enumerations and Translations	102
5.2.2	Eliminating Redundant Translations	105
5.2.3	Sharing enumerations Between Collections	107
5.2.4	Propagating Identifiers through Collection Elements	110
5.2.5	Interprocedural Transformation	112
5.2.6	Enumerating Nested Collections	113
5.2.7	Collection Selection	114
5.2.8	Performance Engineering with Optimization Directives	114
5.3	Evaluation	116
5.3.1	Experimental Setup	117
5.4	Discussion	129
5.4.1	Prior Works	130
5.4.2	Limitations	132
6	Representing References to SSA Data Collections	133
6.1	Bookmark Semantics, Informally	136
6.1.1	Lineage Functions by Example	138
6.2	Bookmark Semantics, Formally	139
6.3	Extending Bookmarks to Represent Iterators	141
6.3.1	Iterator Operations	143
6.3.2	Iterators as Bookmarks	144
6.3.3	Updates using Iterators	144
6.4	Lowering to Destructive Operations	146
6.4.1	Pointer Stability	149
6.4.2	Implementation Specification	150
6.5	Bookmark Safety	150
6.5.1	Lineage Graph	151

	10
6.5.2	Validity Analysis 152
6.5.3	Stability Analysis 153
6.5.4	Interprocedural Analysis 156
6.6	C++ Library 158
6.6.1	Towards Zero Overhead Abstraction 159
6.7	Discussion 161
6.7.1	Expressiveness 161
6.7.2	Optimizations Enabled by Bookmarks 161
6.7.3	Limitations 162
7	SSA Destruction with No Spurious Copies 163
7.1	Language Definition 168
7.1.1	Core Language (CORE) 169
7.1.2	IMP Language and Reference Semantics 171
7.1.3	ssa Language and Value Semantics 172
7.1.4	Handling Memory Aliasing 173
7.2	SSA Destruction Without Spurious Copies 174
7.3	Construction Yields Consumption-Safe Programs 175
7.4	Transformation 176
7.4.1	Elimination 178
7.4.2	Inlining 181
7.4.3	Injection 182
7.4.4	Code Motion 182
7.4.5	Replacement 183
7.4.6	Missed Opportunities Due to Tightening 183
7.5	Repairing Violations 184
7.5.1	Schedule 185
7.5.2	Rethreading 185

7.5.3	Speculative Transformation	186
7.6	Evaluation	187
7.6.1	Experimental Setup	187
7.6.2	Tightened Transformations with ALMANAC	188
7.6.3	Repairing Violations with ERRATA	189
7.7	Related Work	197
7.7.1	The Aggregate Update Problem	198
7.7.2	Linearity, Uniqueness, and Lazy Linearity	198
7.7.3	Type-Preserving Compilation	199
7.8	Discussion	199
8	Conclusion	202
A	Supplemental Proofs for SSA Destruction with No Spurious Copies	221
A.1	Operational Semantics	221
A.1.1	Semantic Domain	221
A.1.2	Instruction Evaluation	221
A.1.3	Small-Step Operational Semantics	222
A.2	Formal Definition of SSA Destruction	224
A.3	Formal Definition of SSA Construction	224
A.4	Elimination Preserves Consumption-Safety	225
A.5	Inlining Preserves Consumption-Safety	225
A.6	Block Injection Preserves Consumption-Safety	225
A.7	Let Injection Requires Tightening	226
A.8	Code Motion Requires Tightening	227
A.9	Replacement Requires Tightening	228
A.10	SSA Destruction of a Consumption-Safe Program Guarantees No Spurious Copies	229
A.11	SSA Construction yields a Consumption-Safe Program	230

A.12 SSA Construction and Destruction are Semantics Preserving	231
A.12.1 State Equivalence	231
A.12.2 Copy-Free SSA Destruction Preserves Semantics	232
A.12.3 SSA Construction Preserves Semantics	234
A.13 Mapping Optimizing Transformations to Building Blocks	236

List of Figures

1.1	Complex memory behavior of data collection implementations blocks optimization.	21
1.2	Breakdown of attempts to apply constant folding, comparing scalar instructions with memory loads.	22
1.3	Attempts to apply code sinking, broken down by the root cause that blocked transformation. Higher successes are better.	22
1.4	Percent of value numbers describing memory loads in the LLVM global value numbering (GVN) pass. Lower is better.	22
2.1	Example C program with multiple definitions.	28
2.2	Example C program in SSA form.	29
2.3	Example control-flow (red, solid) and data-flow (blue, dashed) graph.	31
2.4	Example Hasse diagram of a meet-semilattice.	32
2.5	Finite meet-semilattice of 32-bit integers.	33
2.6	Solution to constant folding dataflow problem using dense analysis on the control-flow graph.	34
2.7	Solution to constant folding dataflow problem using sparse analysis on def-use chains.	35
2.8	Example C program in Extended SSA form.	37
3.1	Memory allocations and accesses in SPEC CPU 2017 Integer suite, classified by high-level data structure.	41
3.2	Syntax of MEMOIR.	50
3.3	Simple MEMOIR program accessing a map.	51

3.4	C++ program to compute the sum of squares with array-of-structs (AoS) to struct-of-arrays (SoA) optimization.	55
3.5	Imperative source program.	57
3.6	Renamed imperative program.	58
3.7	Constructed MEMOIR program.	59
3.8	Destructed MEMOIR program.	60
3.9	MEMOIR program where a variable is live after an update.	60
3.10	Destructed MEMOIR program introduces a spurious copy.	61
3.11	MEMOIR syntax extensions with implementation annotations.	66
3.12	Array program rewritten to use aggregate operators, and then put into SSA form.	69
4.1	C++ program computing the histogram of a list of items before and after applying redundant indirection elimination (RIE).	89
4.2	Relative execution time of ported SPECINT 2017 benchmarks	91
4.3	Relative memory usage of ported SPECINT 2017 benchmarks.	91
4.4	Relative execution time for breakdown of mcf optimizations.	92
4.5	Relative memory usage for breakdown of mcf optimizations.	92
4.6	MEMOIR layout optimizations applied to hot struct in mcf.	94
5.1	Visualization of unifying parameters in Algorithm 6.	113
5.2	Hierarchical clustering of benchmarks based on the breakdown of dynamic collection operations executed.	118
5.3	Performance and memory usage of ADE compared to MEMOIR running on Intel-x64.	120
5.4	Performance and memory usage of ADE relative to MEMOIR on AArch64.	121
5.5	Ablation study, whole-program speedup with optimization techniques disabled on Intel-x64 relative to ADE with all optimizations enabled.	125

5.6	Ablation study, memory usage with sharing disabled on Intel-x64 relative to ADE with all optimizations enabled. RTE had no impact on memory usage, while propagation only impacted FIM with similar results to sharing.	126
5.7	Whole-program speedup of MEMOIR and ADE with and against <code>Swiss{Set, Map}</code> on Intel-x64, higher is better.	128
5.8	Maximum resident set size comparison with <code>Swiss{Set, Map}</code> on Intel-x64, lower is better.	129
6.1	MEMOIR syntax extensions to support bookmarks.	140
6.2	Semantic domain of CORE.	141
6.3	Instruction Evaluation Rules of CORE.	142
6.4	Small-step operational semantics of SSA.	143
6.5	Syntax extensions for SSA to support iterators.	143
6.6	Evaluation of iterator instructions.	146
6.7	Instruction Evaluation Rules for Destructive Updates.	147
6.8	Hasse diagram of the lattice used in abstract interpretation of a lineage graph. . .	152
6.9	Program with an if-then-else pattern and its lineage graph, which widens symbolic arithmetic to $?$ and over-approximates $\text{meet } ? \wedge \perp = \perp$	153
6.10	Program with a loop and its cyclic lineage graph, evaluated for bookmarks b_0 and b_2	154
6.11	Stability analysis over a program where bookmark b_0 is stable across the erase operation, and b_2 is not, which would result in undefined behavior.	157
7.1	Compilation pipeline model used in this chapter.	168
7.2	Syntax of CORE, IMP and SSA	169
7.3	Semantic domain of CORE, SSA and IMP.	170
7.4	Instruction Evaluation of CORE.	171
7.5	Small-step operation semantics of CORE.	171

7.6	Instruction Evaluation of IMP.	172
7.7	Instruction Evaluation of SSA.	173
7.8	SSA Destruction relation rules of interest. D-INPLACE destructs a functional update to an in-place destructive update. D-COPY introduces a <i>spurious copy</i> that was not present in the original program.	174
7.9	Consumption-safety of SSA, expressing the conditions for all update operations in an SSA expression to be destructed into in-place updates (D-INPLACE).	175
7.10	Key rule of the SSA Construction relation.	176
7.11	Transformations that naturally preserve consumption-safety.	179
7.12	Transformations that do not preserve consumption-safety, and need to be tightened.	179
7.13	Tightened transformations from Figure 7.12.	180
7.14	Speculative transformation conditioned on successful repair. Combining repair with rollback enables optimistic transformation with guaranteed consumption-safety.	186
7.15	Experimental setup used to evaluate ALMANAC tightened transformations.	189
7.16	Experimental setup used to evaluate ERRATA repair transformations.	190
7.17	Comparison of generated executable performance, shown as per-benchmark speedup relative to LLVM (log scale). All pipelines run the same optimization pipeline (-O3). The x axis of plot is sorted by ALMANAC speedup with ERRATA speedup for the same benchmark placed to its right. ALMANAC sees significant performance regressions on some benchmarks due to tightened guards leading to missed optimizations, while ERRATA does not see a statistically significant regression from LLVM.	194
7.18	Raincloud diagrams comparing compilation time between the four compilation pipelines, evaluated on the LLVM test suite. All pipelines run the same optimization pipeline (-O3). Neither ALMANAC nor ERRATA have a significant impact on compilation time.	195

A.1	Semantic domain of CORE, SSA and IMP.	221
A.2	Instruction Evaluation of CORE.	222
A.3	Instruction Evaluation of IMP.	222
A.4	Instruction Evaluation of SSA.	222
A.5	Small-step operation semantics of CORE.	223
A.6	SSA Destruction relation.	224
A.7	SSA Construction Relation ($\xrightarrow{\text{SSA}}$).	224

List of Tables

3.1	Solution to constant analysis on Figure 3.3	53
5.1	Collection selection candidates and the average case complexity of basic operations, where n is the number of elements in the collection k is the largest integer key in the collection.	115
5.2	List of benchmarks used to evaluate ADE, and their baseline execution time. . . .	118
5.3	Sparse/dense access counts relative to MEMOIR.	123
5.4	Per-operation speedup relative to Hash{Set,Map}.	123
7.1	Summary of transformation building blocks, if they preserve consumption-safety, and example compiler passes.	177
7.2	Efficacy of tightened transformations evaluated with the cross product of 1,000,000 random LLVM programs and 29 transformations (24 passes and 5 default pipelines) for 29,000,000 total test cases. Successes, false positives and failures are computed as described in Figure 7.15.	191
7.3	Pass statistics showing transformations blocked by tightened guards in ALMANAC on the LLVM test suite. As expected, transformations that may violate consumption-safety are immediately blocked. However, the cascading effects of these blocked transformations is only seen in a few cases for SLP vectorization and backend instruction selection.	193

7.4	Breakdown of repair success by transformation, shown as percentage of violations successfully repaired (higher is better). The full ERRATA pipeline achieves a 100% success rate for all violations. I evaluate the efficacy of individual ERRATA transformations and ablate each repair transformation. No single repair transformation is sufficient to repair all violations. Furthermore, each repair transformation is necessary to achieve no violations in the full pipeline.	196
A.1	Mapping of Allen & Cocke’s catalogue of optimizing transformations [110] to our transformation building blocks, and whether they naturally preserve consumption-safety.	237

CHAPTER 1

Introduction

This dissertation provides arguments in support of the following thesis:

Preserving the logical structure of data collections in the compiler intermediate representation, rather than prematurely lowering them to raw memory, enables a novel class of analysis and optimization.

1.1 Data Collections

Data collections are everywhere. From arrays and tensors to lists, sets and maps, they make up the vast majority of data accessed in modern programs.

Data collections provide a logical, high-level abstraction for how data is organized in a program, abstracting away the low-level data structure implementation and raw memory operations used to implement them. This level of abstraction helps developers to more easily comprehend code, and enables a separation of concerns between modules that share data.

However, modern general-purpose compilers do not benefit from this rich, high-level information. Instead, the compiler is given a low-level view of memory after all of these abstractions have been stripped away. As a result, information that is apparent to developers is incomprehensible to the compiler. Take the following C++ code for example, which uses a hash table to store key-value pairs:

```

1 int foo(std::unordered_map<int, int> &table) {
2     table[1] = 10;    // insert (1,10)
3     table[2] = 20;    // may rehash the container
4     return table[1];  // always returns 10, but no
5 }                    // production compiler can optimize!

```

Figure 1.1: Complex memory behavior of data collection implementations blocks optimization.

While the read of `table[1]` on line 4 always yields the value `10`, no production compiler can perform this simple constant propagation. This is caused by the intermediate insertion `table[2] = 20` on line 3, which may cause the underlying container to be rehashed. Following the rehash, the key-value pair (1,10) may have been moved to a new location in memory. As a result, the compiler conservatively assumes that it does not know what value it will load from memory and the constant propagation is blocked.

1.2 Why do Compilers Fail to Optimize Memory?

While simple examples, like the one above are helpful illustrations of the underlying problem, I find that such complex memory operations frequently block compiler optimizations in real benchmarks. For example, I find that the `clang` compiler frequently fails to perform constant folding, code sinking, and global value numbering in the presence of memory operations when compiling the SPEC CPU 2017 Integer benchmark suite, as shown in Figures 1.2 to 1.4.

From this data, I summarize the three primary classes of complex memory operations that block the compiler: stateful data accesses, linked data structures, and premature optimization.

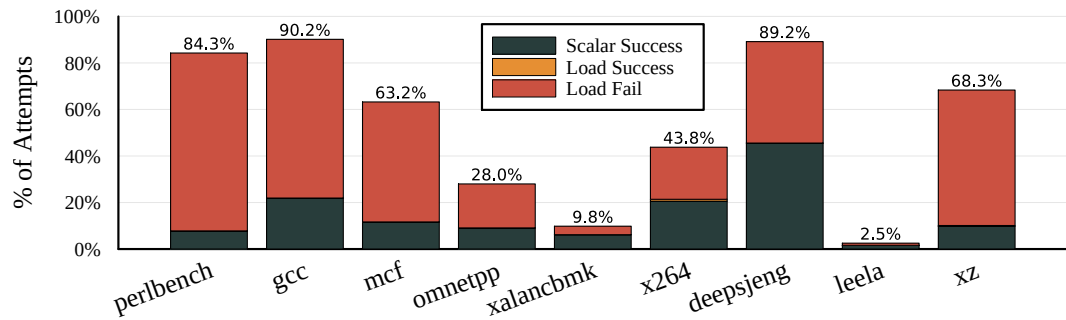


Figure 1.2: Breakdown of attempts to apply constant folding, comparing scalar instructions with memory loads.

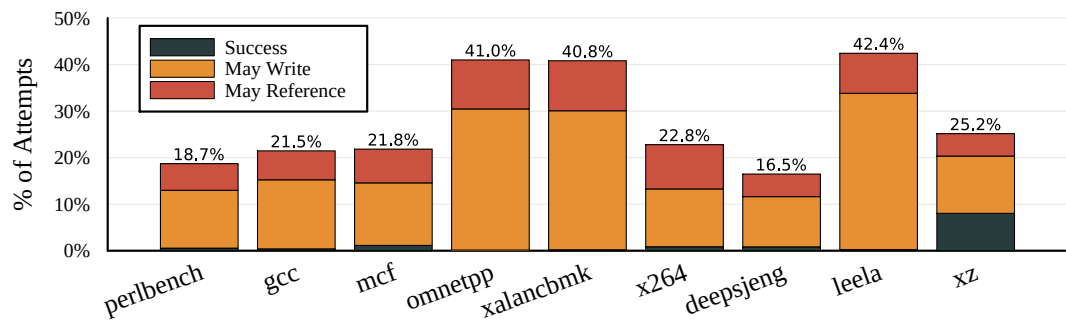


Figure 1.3: Attempts to apply code sinking, broken down by the root cause that blocked transformation. Higher successes are better.

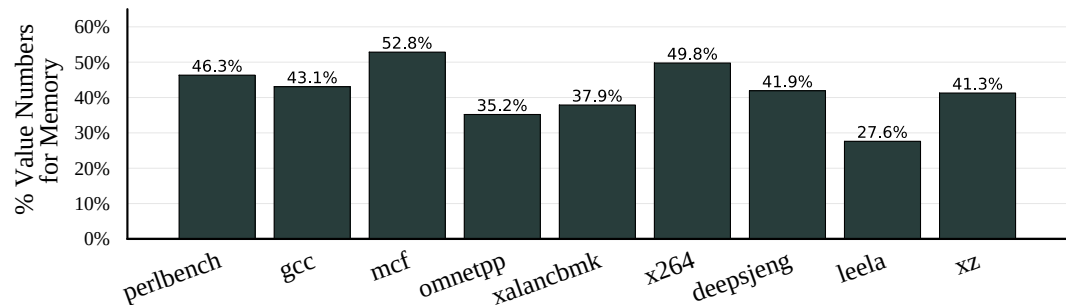


Figure 1.4: Percent of value numbers describing memory loads in the LLVM global value numbering (GVN) pass. Lower is better.

Stateful data accesses create data dependencies between unrelated memory operations. The hash table example above is a prime example, since every insertion to the map may update the state of the entire collection. Beyond that example, stateful data access are seen in data structures like swiss tables and splay trees.

Linked data structures—such as linked lists, trees and skip lists—are commonly used for fast insertion and removal. However, they are the source of the undecidability of pointer alias analysis [1], which compilers rely upon to understand how values flow through memory. As a result, the compiler cannot precisely track the value of individual elements within such data collections.

Premature optimizations performed by developers, such as reusing memory or overprovisioning arrays, cause memory analyses to fail when details about element-level liveness is needed. This results from memory locations with disjoint lifetimes being aggregated into the same, long lived lifetime. As a result, the compiler cannot disambiguate accesses between unique objects, creating unresolvable false dependencies.

While these implementation-level details are difficult to reason about, the logical operations they perform is not. For example, a hash table represents a key-value store and a linked-list represents contiguous values. By preserving these high-level abstractions in the compiler, we can elide the issues introduced by prematurely lowering them before optimization.

1.3 How to Preserve Data Collections in the Compiler?

The vast majority of compiler analysis and optimization occur on the compiler intermediate representation. However, modern IRs do not represent the high-level data collections commonly seen in today’s programs, forcing the compiler to rea-

son about collections through the lens of raw memory operations.

This is not for lack of infrastructure. Modern compiler frameworks like LLVM and MLIR [2], [3] have enabled a surge in new compilers for specific applications, higher-level languages, and emerging hardware targets. Despite this proliferation, no general-purpose IR abstraction for data collections exist to serve as shared infrastructure.

The closest existing approaches are the vector and tensor types in LLVM and MLIR, which demonstrate that preserving collections in the IR is both feasible and beneficial for analysis and optimization. However, these types only capture a small subset of data collections used in practice. More complex collections—such as key-value stores and linked structures—cannot be represented. As a result, they must be prematurely lowered, discarding the semantic information that would make them amenable to optimization.

In this thesis, I present MEMOIR: a general-purpose compiler intermediate representation for data collections in an SSA form. By designing MEMOIR in static single assignment (SSA) form, existing compiler analysis and transformation can be applied to it with little to no additional effort. The key insight of MEMOIR is the decoupling of a collection’s logical organization from its low-level memory implementation. This decoupling has two benefits. First, the compiler can explicitly reason about logical collections, enabling element-level analyses and generalizations of conventional scalar optimizations that are intractable on lowered representations. Second, the compiler is free to optimize the memory layout of a collection independent of its logical semantics, enabling data structure transformations that are infeasible in conventional compilers. With a compact set of types and operations, existing compiler infrastructures can feasibly incorporate MEMOIR as an IR extension.

Crucially, preserving collections in the IR only helps if the compiler can lower them back to efficient imperative code. I address this by deriving *consumption-safety*: a property of MEMOIR programs that guarantees SSA destruction will not introduce any copies that were not already present in the original program. I prove that conventional SSA construction naturally produces consumption-safe programs, show that most compiler transformations preserve it, and derive the necessary applicability guards for those that do not. This makes construction and destruction of MEMOIR transparent, so that users solely see the benefits of improved analysis and optimization.

Finally, I extend MEMOIR with *bookmarks*, which represent references and iterators to collection elements as immutable SSA values. With bookmarks, MEMOIR can represent by-reference accesses, views, and iterators without violating SSA form. Doing so, MEMOIR can be applied to programs that rely on references and iterators, without lowering them prematurely.

The remainder of this dissertation is structured as follows. Chapter 2 provides the background necessary to understand the contributions of this thesis. Chapter 3 describes the high-level design of MEMOIR. Chapter 4 illustrates the benefits of MEMOIR by introducing novel analysis and optimization on the logical organization of collections, which apply across source languages, target architectures and collection implementations. Chapter 5 presents a novel optimization that enables specialized collection implementations, achieving orders of magnitude speedup and memory reduction. Chapter 6 formally defines MEMOIR semantics and its extensions to represent references and iterators using bookmarks. Chapter 7 proves that the round trip from imperative code to MEMOIR and back introduces no spurious copies, and demonstrates how compiler transformations can be constructed to preserve this guarantee. Chapter 8 closes the dissertation by reflecting on the

contributions of MEMOIR and its impact on modern compiler infrastructure.

CHAPTER 2

Background

This chapter provides the background necessary to understand the contributions of this dissertation. The central concept is static single assignment (SSA) form, which underlies both the design of MEMOIR and the formal results that support it.

This chapter is organized as follows: Section 2.1 begins by introducing SSA form and its key property, referential transparency. Section 2.3 continues with an introduction to dataflow analysis, and describes how SSA enables efficient solvers via its sparse representation of dataflow. Section 2.2 describes how referential transparency enables compiler transformations to be expressed as simple rewrite rules, enabling optimizations to be applied without needing to inspect the entire program. Sections 2.4 and 2.5 describe extensions to SSA form, which enable both analysis and transformation by making more program properties explicit.

2.1 Static Single Assignment Form

Static single assignment (SSA) form [4], [5] is the dominant design for modern compiler intermediate representations, used in both academia and production [2], [3], [6]. Programs in SSA form are restricted such that each variable has a single assignment in the program. A program is converted into static single assignment

form by converting each assignment operation into a definition operation, and patching up uses to maintain the original data flow. Take the following C code as an example:

```

1 int foo(const int a) {
2   int x = 1;      // definition of x
3   int y = 3;      // definition of y
4   if (a == 0)
5     y = y * a;    // assignment to y
6   y = 7;          // assignment to y
7   x = x + y;      // assignment to x
8   return x;
9 }
```

Figure 2.1: Example C program with multiple definitions.

In this program, the variables `x` and `y` are both assigned to multiple times. As a result, it is not immediately apparent which version of `x` is used in the final **return** statement. By constructing the static single assignment form for this program, such information becomes immediately apparent:

In SSA form, many program properties become trivial, i.e., they are explicit in the syntax. For example, the **return** `x1` statement above must refer to the definition `int x1 = x0 + y3`. We no longer need to perform any reachability analysis to determine which version of a variable is being used at a given use site.

Furthermore, in SSA form, all uses of a variable refer to the same assignment. As a result, an expression can be freely replaced by its value (or vice versa) without changing the programs behavior. For example, the expression `x0` can be replaced with `1` without changing the meaning of the statement `int x1 = x0 + y3`. This property is referred to as *referential transparency*, and is the foundation upon which SSA compiler are built.

```

1 int foo(const int a) {
2     int x0 = 1;
3     int y0 = 3;
4     if (a == 0)
5         int y1 = y0 * a;
6     int y2 = phi y1, y0; // y1 if branch taken, y0 otherwise
7     int y3 = 7;
8     int x1 = x0 + y3;
9     return x1;           // always returns x1
10 }

```

Figure 2.2: Example C program in SSA form.

Referential transparency enables local reasoning [7], where the compiler can analyze and transform a use of a variable without needing to track how its value may change elsewhere in the program. From this property, many analyses and transformations become both tractable, scalable and easy to express.

2.2 Rewrite Rules

When a variable usage is referentially transparent, as in SSA, they can be freely replaced with an equivalent expression. For example, in Figure 2.2 the usage of `y3` in `x0 + y3` can be replaced with the constant `7` since it has a single definition `y3 = 7`. Having this freedom to rewrite expressions without needing to inspect other program points drastically simplifies transformation, since one only needs to inspect the definitions of used variables.

With this freedom, most compilers define their transformations as a set of *rewrite rules*. A rewrite rule specifies a syntax pattern to match and replace expressions. For example, the compiler could rewrite `x - x` into `0` as an algebraic

simplification. Some rewrites require additional semantic *applicability guards* to apply. For example, dead code elimination can rewrite `int x = y; ...` into ... iff `x` is never used again.

2.3 Dataflow Analysis

Dataflow analysis is the most common analysis run in the compiler. At its core, a dataflow analysis computes facts about the data that is held by variables or memory locations throughout program execution. There are many flavors and frameworks for computing dataflow analyses, however in this section I will describe how SSA form improves the efficiency of dataflow analysis via sparsity, and provide the background information necessary to understand analyses that I introduce later in this dissertation.

A dataflow analysis computes the possible set of values held by a variable or memory location at a given program point. It does this by inspecting the definitions and uses of variables in the program. There are a variety of program representations that are traversed by a dataflow analysis. For the purposes of this dissertation, I will focus on two:

- The *control-flow graph* (CFG) [8] represents all possible execution paths in the program. See Figure 2.3, where control-flow edges are shaded red.
- The *def-use chains* connect each variable definition with all of its uses. When multiple def-use chains are combined, it is colloquially referred to as a *data-flow graph*¹ (DFG). See Figure 2.3, where def-use chains are shaded blue.

¹This is different from a program dependence graph [9], which represents the data-flow graph and memory dependences.

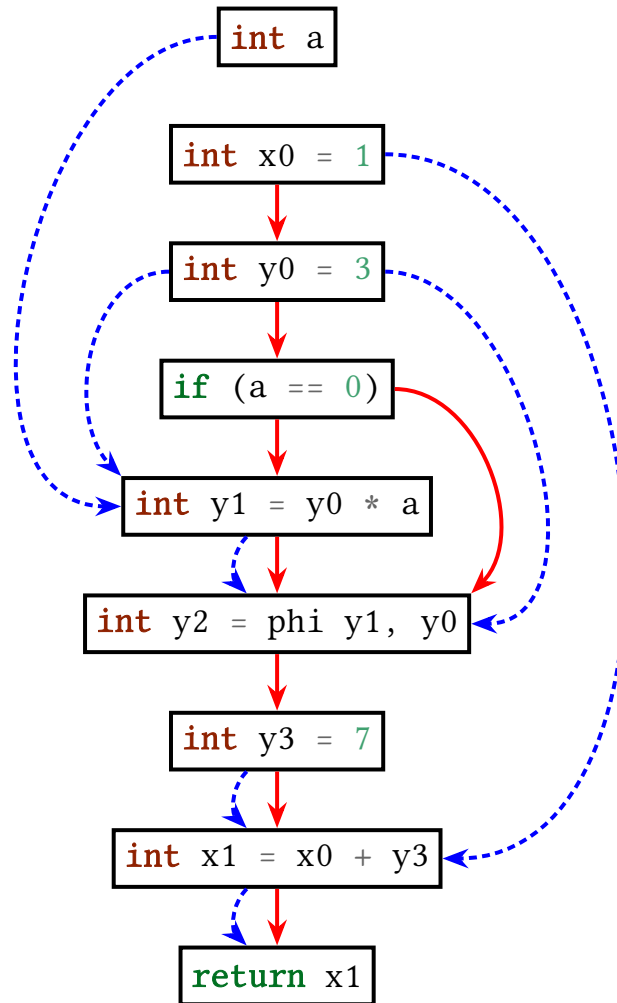


Figure 2.3: Example control-flow (red, solid) and data-flow (blue, dashed) graph.

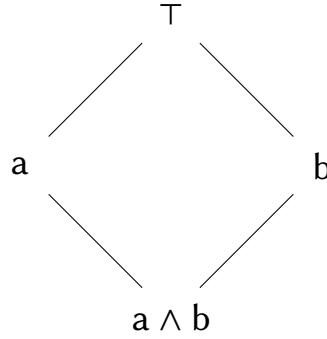


Figure 2.4: Example Hasse diagram of a meet-semilattice.

Kildall [10] introduced a generalized approach to dataflow analyses using a meet-semilattice, defined as a partially ordered set of values $\mathcal{V}$ and a meet operator ($\wedge$). Such lattices are commonly expressed using a Hasse diagram, as shown in Figure 2.4, where $\top$ represents the ‘uninitialized’ state (where there is no information). A transfer function $\mathcal{F}$ defines how each operation in the language computes dataflow facts from their inputs.

To solve a dataflow problem in this framework, each node in the graph being traversed is assigned an initial value $\top$. Then, the transfer function is applied to each value and propagate along all edges. For nodes with multiple incoming edges—for example, after an if-then-else region—the meet of all incoming values is taken and assigned to the node. This is defined as the ‘maximal fixed point’ (MFP) solution for the dataflow problem. In theory, it would be preferable to ‘meet over all paths’ (MOP) in the program, however this is undecidable in the general case [11]. Therefore, the maximal fixed point (MFP) serves as a conservative over-approximation².

To provide a concrete example of dataflow analysis, and why SSA enables an efficient dataflow solver, I will describe and implement a simple constant folding

²Note that, if the transfer functions are distributive, then the MFP solution is equal to the MOP solution.

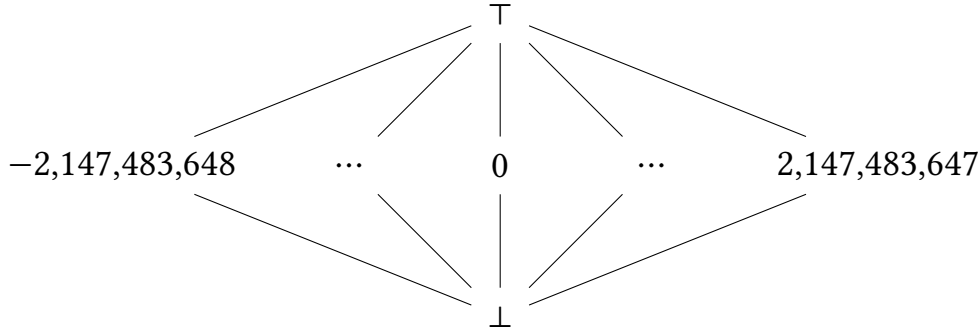


Figure 2.5: Finite meet-semilattice of 32-bit integers.

analysis. I define a meet-semilattice over the bounded set of 32-bit integers (which I assume is the size of `int` on our target machine), as shown in Figure 2.5. Transfer functions are defined for basic arithmetic expressions:

$$x \text{ op } y = \begin{cases} \perp & \text{if } x = \perp \vee y = \perp \\ \top & \text{if } (x = \top \vee y = \top) \wedge x \neq \perp \wedge y \neq \perp \\ \text{Eval}(x \text{ op } y) & \text{otherwise} \end{cases}$$

2.3.1 Dense Dataflow Analysis

To understand the benefits of SSA form for dataflow analysis, let us first apply the *dense* approach to the original program. In the dense approach, the entire control-flow graph is traversed until a fixed point is reached. Because variables may be defined multiple times in the program, the solver must track the state of *all* variables throughout program execution. This is accomplished by encoding our abstract state using a mapping from variable names to lattice points. Doing so, it is able to approximate only the *reaching definitions* at each use site.

The transfer function is applied to the arithmetic expression, whose result is

```

1 int foo(const int a) {
2   int x = 1;    // {x=⊤, y=⊤} ⇒ {x=1, y=⊤}
3   int y = 3;    // {x=1, y=⊤} ⇒ {x=1, y=3}
4   if (a == 0)
5     y = y * a;  // {x=1, y=3} ⇒ {x=1, y=⊥}
6                 // {x=1, y=⊥} ∧ {x=1, y=3} ⇒ {x=1, y=⊥}
7   y = 7;        // {x=1, y=⊥} ⇒ {x=1, y=7}
8   x = x + y;    // {x=1, y=7} ⇒ {x=8, y=7}
9   return x;
10 }

```

Figure 2.6: Solution to constant folding dataflow problem using dense analysis on the control-flow graph.

then assigned in the output mapping. This is referred to as a *forward* analysis, since information is propagated from predecessor to successor in the graph. The iterative solution to the problem, applied to Figure 2.1 is shown in Figure 2.6.

While the precision of the dense approach helps to disambiguate the multiple definitions of y in our example code, it has computational complexity $O(N \cdot E \cdot |L|)$, where N is the number of nodes in the CFG, E is the number of edges, and $|L|$ is the height of the semilattice. Because of this, the dense approach does not scale well to larger programs.

2.3.2 Sparse Dataflow Analysis

With a concrete understanding of the cost incurred by dense dataflow analysis, I will explore how SSA form enables a *sparse* approach. Sparse dataflow analysis uses def-use chains—which are sparse in comparison to control-flow edges—to propagate dataflow facts. Doing so, the solver does not need to propagate all variable state along edges, only the variable being used, since reaching definitions

```

1 int foo(const int a) { // a = ⊥
2   int x0 = 1;          // x0 = 1
3   int y0 = 3;          // y0 = 3
4   if (a == 0)
5     int y1 = y0 * a;    // y1 = 3 * ⊥ = ⊥
6   int y2 = phi y1, y0; // y2 = 1 ∧ ⊥ = ⊥
7   int y3 = 7;          // y3 = 7
8   int x1 = x0 + y3;     // x1 = 1 + 7 = 8
9   return x1;           // return 8
10 }

```

Figure 2.7: Solution to constant folding dataflow problem using sparse analysis on def-use chains.

are made explicit in SSA form. The iterative solution to the problem, applied to Figure 2.2 is shown in Figure 2.7.

Compare to the dense approach, sparse analysis has a computational complexity of $O(|L| \cdot (V + E))$, and $O(V + E)$ if the transfer functions are distributive. This makes sparse analysis scalable to large programs.

Furthermore, the sparse approach has a pull-style design, which is preferable for compiler construction. When analyzing a single use, all relevant def-use chains can be gathered as a partition for the solver. This enables dataflow facts to be computed on demand, instead of ahead-of-time using push-style dense analysis.

2.3.3 Widening and Narrowing

When operating on an infinite lattice—e.g., the set of real numbers—it must be ensured that the dataflow analysis converges to a single value, rather than oscillating between values within the lattice. Cousot and Cousot [12] accomplished this representation by a *widening* operator, which acts as a forcing function to move the

lattice result to a more conservative point instead of oscillating indefinitely. Furthermore, Cousot and Cousot [13] also showed that widening and narrowing over an infinite lattice is strictly more powerful than approximating an infinite lattice with a finite one. I frequently apply this widening approach in dataflow analyses, since dataflow facts with symbolic expressions require infinite lattices to encode them.

2.4 Static Single Information

SSA form makes the flow of data from unique variable definition to uses explicit. With this, information about the definition itself are propagated from the definition to all use sites. For sparse analysis, this guarantees that program points other than the definition need not be analyzed to understand how the value being used was produced.

However, some analyses rely on language constructs other than definitions to propagate dataflow facts. For example, consider the `if (a == 0)` statement in Figure 2.2. The then block of this statement will only execute if `a` is equal to `0`. Because of this, we know that `a` *must* be equal to `0` in the then block. This additional dataflow fact can be leveraged by analysis for further precision.

For dense dataflow analysis, this can be done by making the analysis path-sensitive. However, this drastically increases the memory footprint of the analysis, since dataflow facts are qualified by their path.

For sparse analysis, we rely on def-use chains to propagate dataflow facts, so we must encode this information in unique definitions. This is done by constructing a static single information (SSI) form of the program, where assertions that narrow the set of valid states are given unique variable names. The most simple

instantiation of SSI is *extended SSA form* (E-SSA), which adds unique definitions for values after they have passed a check or assertion. For example, the E-SSA form of the example program is shown in Figure 2.8

```

1 int foo(const int a) {
2     int x0 = 1;
3     int y0 = 3;
4     if (a == 0) {
5         int a0 = a;           // a0 = 0
6         int y1 = y0 * a;      // y1 = 3 * 0 = 0
7     }                         // y1 fact is more precise!
8     int y2 = phi y1, y0;
9     int y3 = 7;
10    int x1 = x0 + y3;
11    return x1;
12 }

```

Figure 2.8: Example C program in Extended SSA form.

With this additional information, our sparse dataflow analysis can incorporate the additional check information to further optimize the program. This achieves a path-sensitive dataflow result with a sparse representation of path conditions on values.

2.5 Dominance and Strict SSA Form

While SSA form guarantees that each variable usage refers to a single definition, it does not guarantee that the variable will have been defined before that use. To remedy this, compilers utilize *strict SSA*. A program is in strict SSA form if, in addition to the single-assignment property, every use of a variable is dominated by its definition. That is, every path from the program's entry to a use of a variable

x must pass through the definition of x . This guarantees that a variable's definition is always available at any point it may be used, without requiring runtime checks or default values. Strict SSA form simplifies compiler construction, preventing the compiler from introducing 'use before initialization' errors.

Because the safety guarantees of strict SSA form are so useful, most discussion about SSA form implies strict SSA form. While recent work has shown that SSA without dominance is more suitable for higher-order programs [14], this dissertation focuses on first-order programs. As such, I will use the term SSA form instead of strict SSA form.

CHAPTER 3

Representing Data Collections in an SSA Form

While prior work on compiler intermediate representations has added first-class support for highly structured aggregate data, such as arrays and tensors, these representations lack more complex data collection types. Today, these data collections are prematurely lowered to a low-level representation, losing valuable information that could enable optimizations. As a result, the compiler is left to glean this information from the remnants. Furthermore, the complex implementation of these data collections lead to undecidable analysis problems that frequently block the compiler. To remedy this, data collections must be preserved in the compiler.

In this chapter, I lay out the design for MEMOIR: a compiler intermediate representation that preserves data collections as first-class citizens. I design MEMOIR with the goal of enabling compiler optimization and static analysis. Doing so, I seek to unlock new degrees of freedom for novel optimizations on the organization of program memory.

This design is grounded by integrating MEMOIR in the LLVM compiler. To accomplish this integration, MEMOIR must enable efficient compiler analysis and transformation by being in a static single assignment (SSA) form. Additionally,

This chapter is adapted from “Representing Data Collections in an SSA Form” in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024.

MEMOIR must be general-purpose, that is, agnostic to the source language and target architecture.

Beyond these conventional notions of general-purpose compilers, MEMOIR is designed to abstract away the underlying memory implementation of the collection. This enables the compiler to re-use core infrastructure across a vast set of collections used in practice. Furthermore, it enables the compiler to automatically optimize the implementation of data collections.

The remainder of this chapter is organized as follows. Section 3.1 motivates the design of MEMOIR through an empirical study of data collection usage in real programs. Section 3.2 introduces MEMOIR by walking through the translation from a C++ program. Section 3.3 presents the MEMOIR syntax, types and operations. Section 3.4 demonstrates element-level sparse dataflow analysis on MEMOIR programs. Section 3.5 discusses how MEMOIR decouples logical organization from memory layout, enabling layout-agnostic analysis and layout optimization. Sections 3.6 and 3.7 describe how MEMOIR programs are constructed from imperative source programs and destructured back to imperative code for execution. Section 3.8 introduces extensions to the core MEMOIR IR.

3.1 Data Collections

In order to capture general purpose data collections, we must understand which data collections are commonly used. Using a data driven approach, I investigated the memory usage of benchmarks in the industry-standard SPEC CPU 2017 Integer suite. Running the benchmarks with the `refspeed` input under the Valgrind DHAT tool [15], I gather information about memory utilization on the heap¹. This

¹These benchmarks use a negligible amount of stack memory.

provides statistics about memory allocations, reads and writes for each allocation site, reported as a call stack. These allocation sites were manually tagged with the high-level data structure class being allocated.

- *Sequential* collections include arrays, lists and tensors.
- *Associative* collections include tables, sets, and maps.
- *Objects* are individual data records.
- *Trees* and *Graphs* are recursive data structures.
- *Unstructured* memory includes files and other memory blobs.

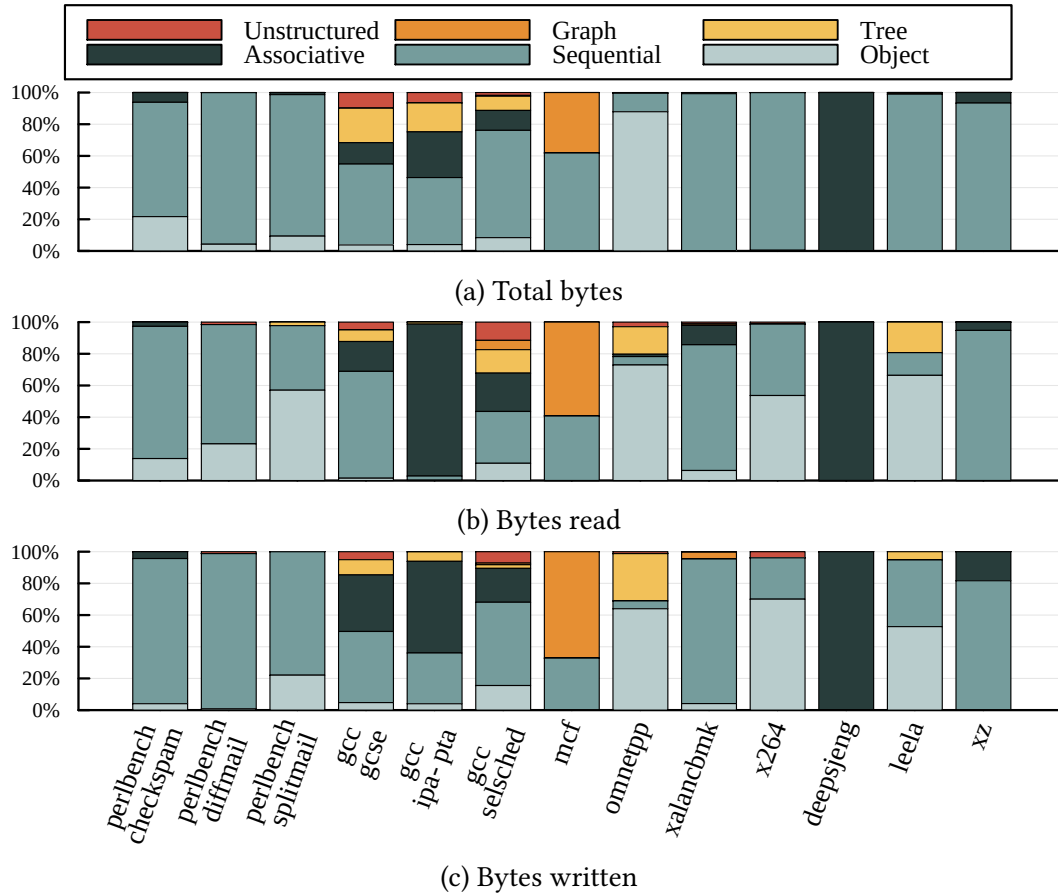


Figure 3.1: Memory allocations and accesses in SPEC CPU 2017 Integer suite, classified by high-level data structure.

This data was aggregated by data structure class and the results are shown in Figure 3.1. From this data, I find that sequential data collections are common across the majority of benchmarks. Additionally, associative collections and objects are the primary type used in individual applications, such as `omnetpp` and `deepsjeng`. Graphs and trees see some use, with `mcf` making heavy use of graphs. Furthermore, I find that the vast majority of memory allocated and used in this program is highly structured. This implies that, even in heavily optimized programs like SPEC, data collections are still used as abstractions over memory. From this data, I design the core collection types for MEMOIR.

3.2 MEMOIR by Example

MEMOIR provides a first-class representation for data collections in an SSA form, but what does that look like in practice? In this section, I provide an informal introduction to MEMOIR by example.

Consider the C++ program in Listing 1, which implements a basic breadth-first search over a graph using two frontiers. I will construct the MEMOIR form of this program, one collection at a time.

```

1 using namespace std;
2 map<int, vector<int>> graph;
3 deque<int> frontier = { 0 }; // initialize with root
4 set<int> visited = {};
5
6 while (not frontier.empty()) {
7     int node = frontier.front(); // read node
8     frontier.pop_front();        // pop node
9
10    for (int neigh : graph[node]) {
11        if (not visited.contains(neigh)) {
12            frontier.push_back(neigh); // enqueue neighbor
13            visited.insert(neigh);     // mark as visited
14        }
15    }
16 }

```

Listing 1: Breadth-first traversal over a graph using a frontier, implemented in C++.

The frontier is an ordered sequence of items, where each item is stored at a position in the list. In this case, the developer employed a deque for efficient insertion to the front and back. However, at its core, all the developer needs is a mapping from position to item. MEMOIR captures this pattern with the *sequence* type (**Seq**<V>), which has a contiguous index space $[0, n)$ with values (V) of the same type.

To create the frontier, the program allocates a fresh sequence containing one element (in this case, 32-bit integers):

```
frontier = alloc Seq<i32> 1
```

The **alloc** $T \bar{x}$ operator creates a new, empty collection variable of type T . It takes an optional list of initialization parameters, $\bar{x}$, which are used to specify the initial size (as above). Once allocated, the program can inspect or update elements using two core operators.

When initializing the frontier, the first element in the list is initialized:

write 0 to frontier[0]

The **write** operator yields a *fresh* collection with the element at the given index updated to hold the new value. All other elements are left unchanged, and the original collection frontier still refers to the sequence with the unchanged element. This reflects a central property of MEMOIR: collection types are immutable, meaning the values held by their elements are fixed once assigned. Modifications are instead represented with operations, like **write**, that perform a functional update, producing a new collection value rather than mutating an existing one.

To check if the frontier is empty on line 6, its size is queried:

size frontier

which yields the number of elements present in the collection. On line 7, the first item in the list is read:

read frontier[0]

which yields the value held in frontier at index 0. During MEMOIR construction, these uses of the collection will be wired to use the definition of frontier that reaches this program point.

On line 8, the first node in the frontier is popped off the front of the list:

```
remove 0 from frontier
```

The **remove** operator yields a fresh collection with the element at index 0 removed. Since the sequence must be contiguous, the following elements shift down one position to fill in the empty space, moving the element at index 1 to index 0, etc.

On line 13, discovered nodes are enqueued by adding an element to the end of the frontier:

```
insert end into frontier
```

Similarly to the **write** operator, this yields a fresh collection. However, **insert** does not modify any of the values held by elements. Instead, it solely adds a new element to the list. The **end** keyword is used to reference one-past-the-last element in the sequence. Insertion at the end does not cause other elements to shift to new positions, but inserting at an arbitrary index may. I refer to operations like **insert** and **remove** as *remapping*—since they move, add or remove elements.

The visited set is captured in MEMOIR with the *set* type (**Set**<i32>), which holds unique indices. Unlike **insert** and **remove** on a sequence, remapping operations on sets do not move existing elements. Instead, they are solely added and removed.

On line 11, the program checks if a given index is present in the set:

```
has visited[neigh]
```

which yields true if the set contains the index *neigh*, and false otherwise.

Finally, the graph adjacency list is a mapping from each node to its own collection of neighbors (a collection of collections). MEMOIR represents this directly as a nested type `Map<i32, Seq<i32>>` where each key (a node identifier) maps to a value that is itself a `Seq<i32>` of neighbor identifiers. Nesting requires no new operators. Reading an element in a node's neighbor list is an ordinary **read** on the base map, **read** graph[node][idx], which yields a i32 value. Though not shown in the example, updates to a nested collection type must be provided the *base* collection value, which is updated wholesale.

Listing 3.1 shows the full MEMOIR program translated from the C++ program. Some details in the original C++ program are not shown explicitly here. Namely, the allocations use abstract types and the inner for loop uses index-based iteration instead of a C++ iterator object. These language constructs are explicitly represented using extensions of core MEMOIR, which I define in Section 3.8.

```

1  block entry()
2    let graph = alloc Map<i32, Seq<i32>> in
3    let frontier = alloc Seq<i32> 1 in
4    let frontier0 = write 0 to frontier[0] in
5    let visited = alloc Set<i32> in
6    jmp while(frontier0, visited)
7
8  block while(frontier1: Seq<i32>, visited1: Set<i32>)
9    let n = size frontier1 in
10   if0 n exit() body()
11
12  block body()
13    let node = read frontier1[0] in # read front
14    let frontier2 = remove 0 from frontier1 in # pop front
15    jmp for(0, frontier2, visited1)
16
17  block for(idx: i32, frontier3: Seq<i32>, visited2: Set<i32>)
18    let num_neighs = size graph[node] in # number of neighbors
19    let cond = cmp eq idx num_neighs in
20    if0 cond while(frontier3, visited2) if()
21
22  block if()
23    let neigh = read graph[node][idx] in # read current neighbor
24    let contains = has visited2[neigh] # check if visited
25    if0 contains continue() then() # continue loop, or insert
26
27  block then()
28    let visited3 = insert neigh into visited2 in
29    let frontier4 = insert end into frontier3 in
30    jmp continue(frontier4, visited3)
31
32  block continue(frontier5: Seq<i32>, visited4: Set<i32>)
33    let idx' = add idx 1 in
34    jmp for(idx', frontier5, visited4)
35
36  block exit() # rest omitted

```

Listing 3.1: Breadth-first traversal over a graph using a frontier, translated from C++ into MEMOIR.

3.3 MEMOIR Syntax and Informal Semantics

In this section, I define the core MEMOIR syntax and informal semantics for the sake of completeness and documentation. For a formal definition of MEMOIR semantics, see Chapter 6.

MEMOIR represents logical data collections as an index-value mapping, $C : Key \rightarrow Val$. I refer to an individual index-value pair ($Key \times Val$) in the collection as an *element*. I refer to the set of indices present in the index-value mapping ($Domain(C)$) as the *index space* of a collection.

MEMOIR has three primary collection types:

- *Sequence* (**Seq**<V>) has a contiguous index space $[0, n)$ with values (V) of the same type.
- *Map* (**Map**<K,V>) associates keys (K) to values.
- *Set* (**Set**<K>) contains a set of unique keys.

These capture the most commonly used collections across a variety of programming languages and libraries.

In addition, MEMOIR has compound types to represent more complex data structures by combining multiple collections. The tuple type (**Tup**<V, ...>) has a contiguous, statically-known index space with values of possibly different types. These types can also be nested to create more complex data structures.

For example, graphs can be represented in MEMOIR as maps, where each node is assigned a unique identifier (Id). This identifier is then mapped over to the set of neighbor nodes:

Map<Id,**Set**<Id>>

This provides a high-level representation for graphs, however it can also be used to represent specific memory layouts as well. For example, MEMOIR can represent a graph in compressed sparse row (CSR) format:

Map<Id,Seq<Id>>

Or, it can be represented as an edge list:

Seq<(Id,Id)>

With this set of collection types, MEMOIR provides a natural representation for the most commonly used data collections, while being expressive enough to represent more specialized ones.

MEMOIR collection types are immutable. The elements, and the values they hold, are fixed upon assignment. MEMOIR uses aggregate update operations to represent modifications to the collection. These operations perform functional updates, yielding a fresh collection with the updates applied, while the input collection is left unmodified. In Section 3.9, I discuss how this design provides a representation that is easy to comprehend, and matches production compilers.

The syntax of MEMOIR is provided in Figure 3.2. MEMOIR is built on top of a typical control-flow graph (CFG) program representation for control flow with basic block arguments instead of ϕ nodes. Blocks are represented with recursive let-bindings, which execute an instruction, bind the results to fresh variable names, and then execute the continuation. Blocks are exited from via terminator expressions, which either jump to another block, or return from the containing function. Each function is composed of a list of blocks. The first block is the entry, whose basic block arguments are also the function arguments.

$x \in \text{Variables}, l \in \text{Labels}, c \in \text{Constants}$	
$f ::= \mathbf{fn} \ l \rightarrow \tau \ \bar{b}$	<i>function</i>
$b ::= \mathbf{block} \ l \ (\bar{x} : \bar{\tau}) \ e$	<i>basic block</i>
$e ::= \mathbf{let} \ \bar{v} = i \ \mathbf{in} \ e$	<i>operation</i>
$\mathbf{jmp} \ l \ (\bar{v}) \mid \mathbf{if0} \ x \ l \ (\bar{v}) \ l \ (\bar{v})$	<i>block terminators</i>
$\mathbf{ret} \ \bar{v}$	<i>function terminator</i>
$i ::= \mathbf{alloc} \ \tau \ \bar{x}$	<i>construction</i>
$\mathbf{read} \ x[v]$	<i>read from element</i>
$\mathbf{write} \ v \ \mathbf{to} \ x[v]$	<i>write to element</i>
$\mathbf{insert} \ v \ \mathbf{into} \ x$	<i>insert element</i>
$\mathbf{remove} \ v \ \mathbf{from} \ x$	<i>remove element</i>
$\mathbf{has} \ x[v]$	<i>contains element</i>
$\mathbf{size} \ x$	<i>number of elements</i>
$\mathbf{call} \ l \ (\bar{v})$	<i>function call</i>
$v ::= x \mid c$	<i>scalar operand</i>
$\tau ::= \mathbf{Seq} \langle \bar{\tau} \rangle \mid \mathbf{Map} \langle \tau, \bar{\tau} \rangle \mid \mathbf{Set} \langle \bar{\tau} \rangle$	<i>collection types</i>
$\mathbf{Tup} \langle \bar{\tau} \rangle$	<i>tuple type</i>
$i64 \mid i32 \mid \dots \mid f64 \mid f32 \mid \dots$	<i>scalar types</i>

Figure 3.2: Syntax of MEMOIR.

MEMOIR includes the collection operations described in Section 3.2, along with basic arithmetic and logic operations. The call instruction binds operands to entry block arguments, executes the function beginning at the entry block, and yields return values.

3.4 Element-Level Sparse Dataflow Analysis

To illustrate the ease of element-level analysis with MEMOIR, I will define a simple constant analysis over map variables, which resolves the motivating example in Figure 1.1. First, the example is translated into MEMOIR:

```

1  block entry(x0: Map<i32,i32>)
2    let x1 = insert 1 into x0 in
3    let x2 = write 10 to x1[1] in
4    let x3 = insert 2 into x2 in  # logical insertion is preserved
5    let x4 = write 20 to x3[2] in
6    let v = read x4[1] in
7    ret v                        #! production compilers could NOT optimize this

```

Figure 3.3: Simple MEMOIR program accessing a map.

I then define a lattice for scalar values $L_n = \perp \leq \mathbb{N} \leq \top$. I then define a lattice for finite collections $L_c = \{L_n \mapsto L_n\}$. L_c is partially ordered by subset inclusion, such that $L_c^1 \leq L_c^2$ iff $L_c^1 \subseteq L_c^2$. I write $L(v)$ to fetch the current lattice point of the value v . If the value is a scalar type $L(v) \in L_n$, if it is a collection type $L(v) \in L_c$. I extend the transfer functions of a conventional constant propagation analysis. I conservatively overapproximate the state of a collection when the index value

cannot be statically resolved to a constant by the analysis.

$$\begin{aligned}
\boxed{\text{alloc } \tau} &= \emptyset \\
\boxed{\text{write } v \text{ to } x[i]} &= \begin{cases} L(x)[L(i) \mapsto L(v)] & \text{if } L(i) \in L(x) \\ L(x) & \text{otherwise} \end{cases} \\
\boxed{\text{read } x[i]} &= \begin{cases} L(x)(L(i)) & \text{if } L(i) \in L(x) \\ \perp & \text{otherwise} \end{cases} \\
\boxed{\text{insert } i \text{ into } x} &= \begin{cases} L(x)[L(i) \mapsto \perp] & \text{if } L(i) \neq \perp \\ \perp & \text{otherwise} \end{cases} \\
\boxed{\text{remove } i \text{ from } x} &= \begin{cases} L(x) \setminus \{L(x) \mapsto \star\} & \text{if } L(i) \neq \perp \\ \perp & \text{otherwise} \end{cases} \\
\boxed{\text{clear } x} &= \emptyset \\
\boxed{\text{has } x[i]} &= \begin{cases} \perp & \text{if } L(i) = \perp \\ 1 & \text{if } L(i) \in L(x) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

The solution can be found using a sparse dataflow analysis over the dataflow graph. For our example program, the analysis is performed over the def-use chains rooted at `x0`. The results are shown in Table 3.1. The results indicate that the value `v` is always equal to the constant 10, and the compiler can propagate it to all uses of `v`, optimizing the return to `ret 10`.

x0	$\perp$
x1	$\{1 \mapsto \perp\}$
x2	$\{1 \mapsto 10\}$
x3	$\{1 \mapsto 10, 2 \mapsto \perp\}$
x4	$\{1 \mapsto 10, 2 \mapsto 20\}$
v	10

Table 3.1: Solution to constant analysis on Figure 3.3

3.5 Decoupling Logical Organization and Layout

When a data collection is prematurely lowered, information about its logical organization is replaced with a low-level implementation with a fixed memory layout. This destroys the rich semantic information in the collection abstraction by collapsing it into a low-level memory representation. The compiler is left to reason about collections through the lens of raw memory operations, which is difficult in the presence of pointer aliasing.

MEMOIR resolves this by preserving data collections in the compiler IR, making the decoupling between logical organization and memory layout explicit. With this separation, the compiler can reason about logical operations independently of how the collection is laid out and accessed in memory. Conversely, the compiler can optimize the memory layout without disturbing the logical semantics, a degree of freedom that is not seen in modern general-purpose compilers.

3.5.1 Layout-Agnostic Analysis

By preserving data collections in the compiler, analysis can reason about logical operations instead of raw memory operations. For example, the sparse dataflow analysis described in Section 3.4 can model collections as a mapping rather than a contiguous block of memory with pointer semantics. This makes the analysis more precise, since it is not built upon conventional alias analysis, which frequently blocks traditional compiler analysis with complex memory operations, as described in Chapter 1.

Furthermore, layout-agnostic analysis enables reuse of compiler infrastructure. If an analysis is not concerned with the memory implementation of a collection, it can be applied uniformly to multiple implementations. For example, analyses over sequential collections can be applied to an array, linked list, or ring buffer with no consideration for the underlying differences between them. Such layout-agnostic infrastructure drastically reduces the cost of compiler development and verification.

3.5.2 Layout Optimization

Since the semantics of data collections in MEMOIR are not implicitly tied to a given memory implementation, the compiler is free to optimize that memory implementation. For example, the canonical array-of-structs to struct-of-arrays optimization helps prevent scatters and gathers in vectorized code. An example from the Intel technical manual [16] is shown in Figure 3.4. This optimization is intractable for nontrivial imperative programs because of pointer aliasing. As a result, it must be performed manually. However, it can be implemented with MEMOIR thanks to the logical representation of collections.

```

struct Node {
    int x;
    int y;
    int z;
}; // scalar fields
Node NODES[1024];

float dist[1024]
for (int i = 0; i < 1024;
      i += 16) {
    float x[16];
    float y[16];
    float z[16];
    float d[16];
    // element->field
    x[:] = NODES[i:16].x;
    y[:] = NODES[i:16].y;
    z[:] = NODES[i:16].z;
    d[:] = sqrtf(x[:] * x[:]
                + y[:] * y[:]
                + z[:] * z[:]);
    dist[i:16] = d[:];
}

```

(a) Array of Structs

```

struct Nodes {
    int x[1024];
    int y[1024];
    int z[1024];
}; // array fields
Nodes NODES;

float dist[1024]
for (int i = 0; i < 1024;
      i += 16) {
    float x[16];
    float y[16];
    float z[16];
    float d[16];
    // field->element
    x[:] = NODES.x[i:16];
    y[:] = NODES.y[i:16];
    z[:] = NODES.z[i:16];
    d[:] = sqrtf(x[:] * x[:]
                + y[:] * y[:]
                + z[:] * z[:]);
    dist[i:16] = d[:];
}

```

(b) Struct of Arrays

Figure 3.4: C++ program to compute the sum of squares with array-of-structs (AoS) to struct-of-arrays (SoA) optimization.

MEMOIR enables other data memory layout optimizations beyond AoS-to-SoA, such as field reordering and array interleaving. I implement such data structure splicing optimizations [17] and novel ones enabled by MEMOIR later in Chapter 4.

3.6 Construction

While functional languages may already closely correspond with MEMOIR, since they both use functional updates for data collections, imperative programs do not. In this section, I describe how MEMOIR is constructed from an imperative program using high-level data collections.

Construction of a MEMOIR program from an imperative source code proceeds in two phases. First, structured control flow is lowered to a CFG using conventional techniques, which I do not address here. Second, imperative collection operations are converted to their functional MEMOIR counterparts via SSA construction, which is the focus of this section.

I define a set of imperative collection operations, which destructively update the collection in-place. These directly map to the MEMOIR collection operations. They are denoted with a **!** suffix. For example, **write!** is the imperative operation corresponding to **write**. Imperative update operations yield a reference to the input object following the update. Because of this, MEMOIR and the input language have the same let-binding structure. An example imperative program is shown in Figure 3.5. I use **let _ = ...** to denote an imperative update whose result is not used, since the in-place update modifies the collection through the existing reference **x**.

```

block entry(x: Map<i32,i32>)
  let c = has x[1] in
  if0 c then() else()
  block then()
    let _ = insert! 1 into x in
    let _ = write! 0 to x[1] in
    jmp cont()
  block else()
    let _ = write! 1 to x[1] in
    jmp cont()
  block cont()
    let v = read x[1] in
    ret v

```

Figure 3.5: Imperative source program.

Before converting imperative updates to functional updates, collection variables are renamed to follow SSA naming conventions using standard SSA construction techniques introduced by Cytron et al. [4]. Block arguments are inserted at the dominance frontier of collection variables, merging reaching versions at control flow join points. Uses of collection variables are updated to use the reaching definition of the original collection variable. Figure 3.6 shows how the example imperative program is renamed.

```

block entry(x: Map<i32,i32>)
  let c = has x[1] in
  if0 c then() else()
  block then()
    let x1 = insert! 1 into x in
    let x2 = write! 0 to x1[1] in # replace with reaching definition
    jmp cont(x2)
  block else()
    let x3 = write! 1 to x[1] in
    jmp cont(x3)
  block cont(x4: Map<i32,i32>) # insert block argument
    let v = read x4[1] in # replace with reaching definition
    ret v

```

Figure 3.6: Renamed imperative program.

Finally, each imperative update is converted to a functional update, yielding a fresh collection variable. Since instructions have already been updated to use the reaching definition, this conversion yields an equivalent program. I formally prove this semantic equivalence result in Section 7.3, and provide the full proof in Section A.12.3. This final step of construction yields a MEMOIR program, as shown in Figure 3.7.

```

block entry(x: Map<i32,i32>)
  let c = has x[1] in
  if0 c then() else()
  block then()
    let x1 = insert 1 into x in # convert to functional operation
    let x2 = write 0 to x1[1] in # convert to functional operation
    jmp cont(x2)
  block else()
    let x3 = write 1 to x[1] in # convert to functional operation
    jmp cont(x3)
  block cont(x4: Map<i32,i32>)
    let v = read x4[1] in
    ret v

```

Figure 3.7: Constructed MEMOIR program.

3.7 Destruction

Once optimizations have been applied to a MEMOIR program, the functional collection operations must be lowered back to imperative code for efficient execution. This process is called *SSA destruction*. I perform SSA destruction using the one-shot bufferization algorithm [3], the standard algorithm for destructing aggregate SSA forms in MLIR, used by dialects such as `tensor`.

One-shot bufferization inspects the def-use chain of each update to determine if it can be performed in-place. A functional update `let y = write v to x[i]` can be lowered to an in-place `write!` if `x` is not used after the update, i.e., it is not live after the instruction. In this case, the update can safely reuse the memory of `x`, since

no subsequent operation will observe the original value. Continuing our example, the MEMOIR program from Figure 3.7 is destructured as shown in Figure 3.8.

```

block entry(x: Map<i32,i32>)
  let c = has x[1] in
  if0 c then() else()
  block then()
    let x1 = insert! 1 into x in # x dead after, in-place
    let x2 = write! 0 to x1[1] in # x1 dead after, in-place
    jmp cont(x2)
  block else()
    let x3 = write! 1 to x[1] in # x dead after, in-place
    jmp cont(x3)
  block cont(x4: Map<i32,i32>)
    let v = read x4[1] in
    ret v

```

Figure 3.8: Destructured MEMOIR program.

However, if x is live after the update, the original value must be preserved. In this case, one-shot bufferization introduces a copy of x before performing the update in-place, as shown in the destruction of Figure 3.9 to Figure 3.10.

```

block foo(y: Map<i32,i32>)
  let y1 = insert 1 into y in #! y live after
  let u = read y[1] in #! y is used here
  jmp next(y1, u)

```

Figure 3.9: MEMOIR program where a variable is live after an update.

```

block foo(y: Map<i32,i32>)
  let y0 = copy y in #! spurious copy introduced
  let y1 = insert 1 into y0 in
  let u = read y[1] in
  jmp next(y1, u)

```

Figure 3.10: Destructed MEMOIR program introduces a spurious copy.

A copy introduced in this way was not present in the original imperative program, and was solely introduced to maintain correctness during destruction. I refer to this as a *spurious copy*. For large collections, spurious copies are costly, turning an $O(1)$ update into an $O(n)$ operation. Guaranteeing the absence of spurious copies is therefore critical for performance.

The necessary condition for copy-free destruction follows directly from the one-shot bufferization algorithm: a functional update can be lowered to an in-place update if and only if the updated object is dead afterwards. I formalize this as *consumption-safety*. A consumption-safe update is the final consumer of the object and can take ownership without requiring a copy. Therefore, if all updates in a program are consumption-safe, then the program can be destructed without introducing any spurious copies.

Chapter 7 establishes consumption-safety as the necessary and sufficient property for guaranteed copy-free destruction. Concretely, I prove that SSA construction from imperative programs always yields a consumption-safe program and that transformations can be made to preserve consumption-safety throughout the pipeline. This guarantee ensures that the round trip from imperative code through MEMOIR and back introduces no overhead beyond what was present in the original program.

3.8 MEMOIR Extensions

The core MEMOIR IR suffices for expressing collection operations in SSA form. However, two classes of extensions are needed to capture the full range of programs and optimizations that MEMOIR targets. First, iterators and references provide traversal and element access patterns beyond by-index access. Second, implementation annotations allow the compiler and developer to constrain the physical layout of collections. Both are introduced informally here and treated formally in later chapters.

3.8.1 Bookmarks

By-index access requires the caller to supply an index at each access site. However, programs frequently need to identify a specific element once and access it repeatedly without repeating the lookup. Take the following code for example, which performs a single lookup in a `std::map` before accessing it multiple times:

```
1 auto &ref = map[key];    // lookup once
2 ref.visited = true;      // update field 1
3 ref.total += value;      // update field 2
```

Thanks to by-reference access, we only need to find the key once. Without by-reference access, the program would need to lookup the key in the map each time:

```
1 // two lookups!
2 map[key].visited = true; // lookup, and update field 1
3 map[key].total += value; // lookup, and update field 2
```

This results in an additional lookup into the `std::map`, an $O(\log n)$ operation. If this code is in the hot loop, it could be a major performance bug.

To avoid prematurely lowering by-reference accesses to by-index accesses in this way, I present *bookmarks* as an extension of MEMOIR. A bookmark is an immutable SSA value that references a single element within a collection. Unlike C++ references, which are used independently from their owning object, bookmarks are accessed in the context of a specific collection version. Concretely, this is done using the syntax `b of x`, read as “access the element referenced by b in the context of collection x.” This syntax can be used anywhere that the core by-index access `x[i]` is used. I provide a formal definition of bookmarks in Chapter 6 and provide an informal description by example here.

The above C++ code can be written using MEMOIR bookmarks:

```

1 let ref = mark map[key] in # bookmark key
2 let map1 = write 1 to ref[1] of map in # update field 1
3 let total = read ref[2] of map1 in # read field 2
4 let total' = add total value in
5 let map2 = write total' to ref[2] of map1 in # update field 2

```

This representation preserves SSA form, since elements are identified by bookmarks and accessed in the context of the collection that owns them. All updates must take the updated collection as an explicit operand rather than mutating through references directly. In Chapter 6, I formalize the bookmark semantics and construct safety analyses that detect invalid accesses caused by logic and memory bugs.

3.8.2 Iterators

Iterators are ubiquitous in collection-oriented programming, providing a general mechanism for logically traversing elements. Beyond traversal, iterators are nec-

essary to represent collections that have no index-based access, such as bags or `std::hive`. For these collections, iteration is the only means of element access.

To capture these semantics, I extend MEMOIR with iterators. An iterator extends a bookmark with the ability to traverse through a collection without exposing the underlying mechanism. Iterators can be used to access collections in the same manner as bookmarks, e.g., `read it of x`. Iterators are constructed to reference the first element of a collection `begin of x` or the ‘one past the end’ sentinel element `end of x`. They can be advanced, e.g., `next it of x`, which yields a fresh SSA iterator value. Finally, iterators can be compared against each other, e.g., `eq it1 it2 of x`, to check if they refer to the same element. I formally define iterators in Chapter 6 and provide an informal description by example here.

Take the following C++ code, for example, which iterates over a map and performs some computation on the key-value pair:

```
1 const std::map<int, int> &items = ...;
2 for (const auto &[key, val] : items)
3     compute(key, val);
```

With iterators, this program can be represented in MEMOIR:

```

1  block preheader(items: Map<i32,i32>)
2    let first = begin of items in # construct iterator to first element
3    jmp header(first)
4
5  block header(it: Iter<i32,i32>)
6    let last = end of items in # get sentinel iterator for comparison
7    let cond = eq it last in items in # did we traverse to the end?
8    if0 cond exit() body()
9
10  block body()
11    let key = getkey it of items in # get key from iterator
12    let val = read it of items in # read value from iterator
13    let _ = call foo(key, val) in
14    let it' = next it of items in # traverse to the next element
15    jmp header(it')

```

The loop follows the conventional C++ iterator pattern, where an iterator is advanced from **begin** to **end**, accessing each element via the current iterator value.

3.8.3 Implementation Annotations

While MEMOIR is designed to be layout-agnostic, there are cases where the implementation matters. For example, the iterator and reference extensions introduced above raise a subtle question: when a collection is updated, do existing references and iterators remain valid? The answer depends on the concrete memory implementation. A `std::vector` may reallocate on insert, invalidating all pointers, while `std::list` guarantees pointer stability.

$$\begin{aligned}\tau &::= \dots \mid \tau \text{ prefer } \textit{impl} \mid \tau \text{ as } \textit{impl} \\ \textit{impl} &::= \text{std::vector} \mid \text{std::list} \mid \dots\end{aligned}$$

Figure 3.11: MEMOIR syntax extensions with implementation annotations.

Beyond safety, some optimizations require a specific implementation to apply. For example, automatic data enumeration, described in Chapter 5, transforms the program to use specialized implementations. Additionally, a developer may have selected a specific implementation in the source language and does not want the compiler to change it. To support these cases, MEMOIR allows collection types to be annotated with an implementation specification:

A collection type annotated with *prefer* expresses a preference, which the compiler may override for optimization. A collection type annotated with *as* expresses a hard requirement, which the compiler must respect. Collection types with no annotation leave the compiler free to select any valid implementation. For example, `Seq<i32> prefer std::vector` is a sequence of 32-bit integers that the developer would prefer to be implemented using `std::vector` from the C++ STL, while `Seq<i32> as std::vector` *must* be implemented as a `std::vector`. This allows developers to express assumptions about the best implementation for a program, or make explicit selections when external performance analysis has shown that one is always best.

The MEMOIR syntax extension is shown in Figure 3.11. The set of valid implementations *impl* is open-ended by design. I describe how this enables the compiler to be extended with third-party and custom implementations in Chapter 6.

3.9 Related Work

Prior work on representing data in SSA form falls into two categories. Virtual SSA forms encode analysis results as SSA-like structures layered on top of the IR. Aggregate SSA forms, which represent data objects as first-class SSA values. MEMOIR builds on the latter, generalizing aggregate SSA forms to encompass general-purpose data collections.

Additionally, prior work has studied how to preserve high-level type information through compilation more broadly. These approaches use typed intermediate and assembly languages to carry source-level structure—including aggregate types—into the back end. MEMOIR shares this goal of preserving structured data through compilation, but explores a different point in the design space: a single representation for a broad class of collection types and implementations, built for integration into a production compiler.

3.9.1 Virtual SSA

Virtual SSA forms are used to represent the results of a static analysis in a compact, easy to analyze manner. Doing so, compilers can perform further analysis and queries on static analysis results as if they were instructions in the program themselves.

Hashed SSA (HSSA) [18] was the first technique for representing aliases and indirect memory operations in an SSA form. HSSA extends a conventional SSA form with virtual variables, which represent memory locations that may be aliased or indirectly accessed. By assigning hash-based names to memory locations, loads and stores through pointers can be optimized similarly to scalar variables. HSSA is limited by the accuracy and precision of the alias analysis used to construct it,

and did not see extensive use in practice.

Memory SSA [19] builds upon HSSA by specifically focusing on memory operations. Memory SSA partitions memory locations into virtual variables based on alias analysis. It models loads, stores and control-flow as `MemoryUse`, `MemoryDef` and `MemoryPhi` nodes layered on top of the IR, representing the memory state at each program point using virtual variables. This lets passes cheaply query def-use chains for aliasing relationships, without building a fully precise partition of memory into separate variables.

Unlike HSSA, Memory SSA is used in production compilers like GCC and LLVM. However, its utilization is limited. For example, while Memory SSA was introduced to LLVM in 2016 as a part of the 3.9 release [20], it was not fully integrated into the GVN pass until LLVM 23 in 2026 [21].

The primary cause of this is that virtual SSA forms cannot be directly transformed. Instead, compiler developers must query the virtual SSA form, relate the results back to the core IR, and express transformations there. This undermines one of the primary benefits of SSA: the ability to express transformations as rewrite rules directly on the representation. MEMOIR avoids this limitation by being a true program representation, where operations on collections are first-class IR instructions that can be analyzed and transformed directly.

3.9.2 Aggregate SSA

Aggregate SSA forms takes a different approach, representing data objects as immutable SSA values updated functionally. This maintains referential transparency for *all* elements of the aggregate.

A straightforward approach for representing arrays in an SSA form was introduced by Cytron et al. [4]. In this SSA form, arrays are updated wholesale using

<pre> v = A[i]; A[j] = v; w = A[j]; </pre>	<pre> v = access(A, i); update(A, j, v); w = access(A, j); </pre>	<pre> v = access(A0, i); A1 = update(A0, j, v); w = access(A1, j); </pre>
(a) Original	(b) Aggregate Operators	(c) SSA

Figure 3.12: Array program rewritten to use aggregate operators, and then put into SSA form.

update and access instructions that treat the array as an immutable value. For example, the imperative code in Figure 3.12a is rewritten to use aggregate operators, which destructively update arrays in-place (Figure 3.12b) before being put into SSA form (Figure 3.12c). This puts the aggregate array itself in an SSA form, where any update to the state produces a fresh variable instead of updating an existing one. Cytron et al. [4] identified that there is an issue with this representation, where unrelated element updates are sequentialized by aggregate update operators. In response, they proposed a ‘hidden update’ operator, which uses dependence analysis to identify locations where indices access disjoint regions of memory. However, this solution is ill-defined in their work and the informal semantics lead to a flimsy representation that is easy to break with transformations.

Later work by Knobe and Sarkar [22] introduced Array SSA form, which introduced an SSA form for array elements. This solved the issue that necessitated the ‘hidden update’ operator in Cytron et al. [4]. By using new ϕ nodes for ‘non-killing’ array definition (ϕ_{def}) and access (ϕ_{use}), they perform optimizations—such as dead store elimination and sparse conditional constant propagation—on array elements. Array SSA form was introduced in the Jalapeño compiler [23] for Java, which provided the type-level guarantees needed to safely transform into Array SSA form. Array SSA form extended many scalar optimizations to operate on array elements, along with new transformations such as array bounds check elimina-

tion [24]. Thanks to Java’s static, strong-typing, they later implemented Extended Array SSA Form [25] to represent fields of objects for further optimization.

To enable explicit analysis and optimization for SIMD execution, most modern compilers have extended their IR with vector types. These vectors are statically sized, or scalable size for SIMD targets with scalable vector width. With the advent of Compilers for machine learning applications, vector types have been generalized to tensor types, providing more expressiveness. While originally compilers only supported fixed-rank tensors with statically-sized dimensions, modern compilers include support for dynamic sizes [26], shape operators [27] and sparsity [28].

While these aggregate SSA forms demonstrate that preserving data in the IR is both practical and beneficial, they are each tailored to a specific collection type with a fixed memory layout. MEMOIR generalizes this line of work to arbitrary data collections and decoupling logical organization from physical layout. Where prior aggregate SSA forms expand the optimization space for a fixed set of types, MEMOIR provides the general-purpose foundation that the field has lacked.

The single-assignment property for elements in Array SSA form [22] is nice for analysis, however it is difficult to optimize because of the verbose representation. Furthermore, the ϕ_{def} nodes assume that the shape of arrays does not change, which preclude it from representing shape operators, such as removing an element. Instead, MEMOIR builds upon the aggregate operation approach proposed by Cytron et al. [4], which is also used for the tensor dialect in MLIR. This provides a more easy to comprehend representation that follows state-of-the-art compiler design.

3.9.3 Relations

While collective operations on dense tensors—where the vast majority of values are non-zero—were properly handled by such opaque operations, many workloads operate on sparse tensors—where the vast majority of values are zero. Storing and iterating over these zeros wastes both memory and compute, so sparse tensors are more efficiently represented in memory using a compact layout, such as compressed sparse fiber (CSF) [29], [30], that stores only the non-zero values and enough structure to locate them. This compact layout comes at a cost during code generation. An expression like $c_i = a_i \times b_i$ can no longer be lowered to a single loop over i , since a and b may store non-zeros at different positions, and the generated code must walk the structure of both operands together to discover (at each step) whether a value exists in one, the other, or both. This process of interleaving traversal over the stored structure of multiple operands, so the compiler can specialize code to the cases that actually arise, is known as co-iteration.

The concept of co-iteration was originally introduced by Kjolstad et al. [28] in the TACO compiler as *merge lattices*, which represents multiplication and addition as a conjunctive merge (both must be present) or disjunctive merge (either must be present). While TACO represented co-iteration as a fixed set of operators, Ahrens et al. [31] made co-iteration a first-class, extensible language with looplets, which decouple how a format is walked from how operands merge. This allows co-iteration to compose across arbitrary structures, rather than a fixed set of level formats. Looplets were later used in the Finch compiler Ahrens et al. [32] to support sparse and structured tensor programming with flexible control flow. More recent work has pushed the same mechanism into new domains, such as real-valued indices [33] and symmetric tensors [34]. A separate line of work ar-

rives at the same need from relational algebra rather than tensor algebra. Indexed streams formalize fused execution as intersection- and union-like operations over streams, showing that relational joins and tensor co-iteration are instances of the same underlying operators [35].

Co-iteration is, in effect, a recurring answer to the same underlying need: representing how the elements of two collections relate, whether one-to-one, one-to-many, or some other correspondence. The relation between operands lives in the merge strategy—whether a conjunction, a disjunction, or a symmetry— or looplet nest, rather than explicitly in the IR itself. By representing relations as edges or annotations between collection values in the IR—rather than left implicit in the operations and structure that produce them—the compiler can effectively analyze and transform them.

3.9.4 Locality

Locality across collections is likewise left largely implicit. TACO and the MLIR sparse dialect coordinate traversal order, temporaries, and layout across multiple tensors only as a byproduct of code generation [28], [36]. The Finch compiler [32] co-optimizes control flow and data structure to improve this locally, by specializing loops to structure so that clustered non-zeros and repeated values are exploited automatically, but layout decisions across separate collections remain outside the abstraction. Galley [37] introduces a cost-based planner that reasons about join order and materialization across multiple tensors, which is the closest existing analogue to cross-collection locality reasoning, but this reasoning is a query-optimization heuristic layered on top of the IR rather than a locality property represented within it. Separately, mechanized verification of structured sparse data operations shows that correctness properties of these formats can be proven compositionally, sug-

gesting locality and layout guarantees could eventually be specified and checked at the same level, but this work targets proof obligations rather than compiler-facing representation [38]. As a result, decisions about whether two collections should be laid out contiguously, interleaved, or independently are still made during lowering or planning, not represented in a form the compiler can analyze or transform earlier in the pipeline.

Relations and locality are two sides of the same coin. Today’s tensor IRs implicitly encode how collections correspond and how they should sit in memory as artifacts of how a computation is lowered or planned, rather than as properties the IR represents directly. While this thesis does not resolve that gap, it provides a general-purpose abstraction that could serve as a vehicle toward it.

3.9.5 Typed Intermediate Languages

Typed intermediate languages [39] preserve type information from the source language throughout compilation. This type-driven compilation approach [40] was primarily driven by the desire for improved back-end code optimization [39], [41], [42], which was achieved by preserving high-level data types from the source language, such as Java and C# objects [43], [44]. However, Shao et al. [45] demonstrated that these typed intermediate languages are difficult to design, especially when representing and manipulating large aggregate types. In response, most production-quality implementations of this approach have been limited to languages with smaller types, such as Microsoft’s Common Intermediate Language (CIL) [46] and more recent work on domain-specific languages [3], [47]. While typed intermediate languages are a well-trodden area of research, MEMOIR explores the design of a general-purpose intermediate representation that preserves abstract data collection types, and how it can be integrated into the production-

quality LLVM compiler.

3.10 Discussion

This chapter introduced MEMOIR, a general-purpose compiler intermediate representation for data collections in an SSA form. By decoupling a collection’s logical organization from its low-level memory implementation, the compiler can reason about logical operations independently of memory layout. This enables element-level analyses that are intractable on lowered representations. Thanks to its SSA form, these analyses can be made efficient by leveraging its sparse data flow representation.

With this foundation, the following chapters explore what becomes possible with MEMOIR. Chapter 4 demonstrates how the logical representation enables novel analysis and transformation that are agnostic to the low-level collection implementation by operating directly on logical collections. Chapter 5 shows how decoupling logical organization from implementation enables the compiler to manufacture data properties, expanding the space of available collection implementations.

The definition of bookmarks in Section 3.8.1 illustrates how MEMOIR can represent by-reference access, but also exposed a core tension with SSA form. When a collection is updated during traversal, the semantics of iterators and references become non-trivial. Chapter 6 resolves this tension by introducing bookmark semantics, which provide a natural representation for these language constructs with concise semantics. Furthermore, bookmarks provide a framework for extending MEMOIR with custom collections through implementation specifications.

Finally, preserving data collections in the IR is only practical if they can be low-

ered back to efficient imperative code. Chapter 7 provides formal guarantees that this round-trip from imperative code to MEMOIR and back can be done without introducing any spurious copies. Furthermore, it provides a framework for compiler developers to reason about how their transformations preserve or violate this guarantee.

CHAPTER 4

Optimizing Logical Organization

In Section 3.5, I described how MEMOIR enables analysis and optimization on the logical organization of data collections. In this chapter, I provide concrete examples for how this enables novel compiler optimizations.

Section 4.1 illustrates how a logical representation enables conventional analysis and transformation to be generalized from scalars to collection elements. I show this concretely by generalizing live variable analysis to *live element analysis*, which analyzes the elements of a collection that are actually used. Using the results of this analysis, I generalize dead instruction elimination to *dead element elimination*, which eliminates updates to elements that are not actually used.

I then show how the layout of data collections can be optimized. Section 4.1.1 introduces *dead field elimination*, which extends dead element elimination by removing unused fields from tuples when safe to do so, reducing memory usage and improving cache behavior. Section 4.2 introduces *Field elision*, which converts fields that are rarely used into a map, whose elements are initialized on demand. Section 4.3 introduces *Redundant indirection elimination*, which eliminates indirect memory accesses by transforming collection types, eliminating pointer chasing patterns.

This chapter is adapted from “Representing Data Collections in an SSA Form” in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024.

These optimizations are then applied and evaluated on the `mcf` and `deepsjeng` benchmarks from the industry-standard SPEC CPU 2017, showing their impact in practice. Finally, I close the chapter with a discussion of the broader impact of these optimizations, and their relation to prior works.

4.1 Live Element Analysis and Dead Element Elimination

Live element analysis generalizes live variable analysis to operate on logical elements of a data collection. Fundamentally, it determines which indices within a collection *may* be accessed in future execution. At its core, this analysis inspects access operations and propagates information about the index up the def-use chain. Take the following code for example, which writes to the second element, reads from the first, and then returns from the function (discarding the sequence):

```

1 block entry(x: Seq<i32>)
2   let x' = write 7 to x[1] in # write to second element
3   let v = read x'[0] in # read first element
4   ret v # sequence is discarded

```

Live element analysis determines that only the element at index 0 of `x'` is used. This information is propagated up the def-use chains to `x'` and `x`. With this dataflow fact, `write 7 to x[1]` need not execute since the element it writes to is never used. Dead element elimination (DEE) takes this live element information, and optimizes the code by eliminating the write:

```

1 block entry(x: Seq<i32>)
2   # eliminate write to dead element
3   let v = read x[0] in # directly use x
4   ret v

```

Listing 4.1: Dead element elimination eliminates write when written element is dead.

4.1.1 Dead Field Elimination

Handling simple cases like these allow us to specialize dead element elimination for tuple fields, which are always accessed using a statically known field index. I refer to this specialization specifically as *dead field elimination* (DFE). In addition to eliminating unused updates, DFE further optimizes the program by eliminating unused fields from tuples. This leads to both computation and memory savings.

For example, take the following code, which allocates a tuple, initializes its fields, and then returns the value contained in the first field:

```

1 block entry()
2   let t = alloc Tup<i32,f32> in # create fresh tuple
3   let t' = write 3 to t[0] in # write to first field
4   let t'' = write 7.77 to t[1] in # write to second field
5   let v = read t'[0] in # read first field
6   ret v

```

Live element analysis indicates that only the first field is live. With this information, both the write to the second field and the second field itself can be eliminated. This results in the following optimized program:

```

1  block entry()
2      let t = alloc Tup<i32,f32> in  # eliminate second field from type
3      let t' = write 3 to t[0] in
4      # eliminate write to second field
5      let v = read t'[0] in
6      ret v

```

4.1.2 Symbolic Range Analysis

While the above code example is the best case for live element analysis, since all reads have statically known indices, that is rarely the case in real code. Rather, the compiler is typically only able to glean information about the range of values a given index may be using a *symbolic range analysis* [48], [49].

This analysis uses a constrained arithmetic over the set of integers, $+\infty$, $-\infty$ and variable symbols with addition, subtraction, multiplication, division, minimum and maximum operations. These expressions are partially ordered such that $e_1 \subseteq e_2$ iff e_1 contains e_2 as a subexpression. This analysis yields a range for each integer variable in the program as a point on the *range lattice* $R = \perp \leq [\ell : u] \leq \top$, where $[\ell_i : u_i] \leq [\ell_j : u_j]$ iff $\ell_i \subseteq \ell_j$ and $u_i \subseteq u_j$. Using this information, the compiler can optimize code in the face of bounded uncertainty. For example, if the write in the above example uses an index that cannot be resolved statically, dead element elimination will insert a condition to skip the write if the element is not live:

Range analysis is typically made more precise using extended SSA form [50], [51], as described in Section 2.4, which refines analysis with path conditions. For example, the following code reads from the array in a loop with a symbolic trip

```

1 block entry(x: Seq<i32>, i: i64)
2   if0 i then() cont(x)
3   block then()
4     let x' = write 7 to x[i] in # conditionally write to index when 0
5     jmp cont(x')
6   block cont(x": Seq<i32>)
7     let v = read x"[0] in
8     ret v

```

Listing 4.2: Dead element elimination transforms the program to conditionally write only on single live element.

```

1 block entry(x: Seq<i32>, i: i64)
2   let x' = write 7 to x[i] in # write to an element
3   jmp loop(10, 0)
4   block loop(j: i64, sum: i32)
5     let v = read x'[j] in # read elements [0:10]
6     let sum' = add sum v in # increment sum
7     let j' = sub j 1 in
8     if0 j exit(sum') loop(j', sum')
9   block exit(sum: i32)
10    ret sum

```

Listing 4.3: Dead element elimination transforms the program to conditionally write when in live element range.

count: To optimize it, dead element elimination inserts a range-check to conditionally perform the write when $0 \leq i \leq 10$.

4.1.3 Lineage Functions

While the example code above solely used **read** and **write** operations—where the live ranges are simply merged and propagated up the def-use chain. MEMOIR includes more complex data collections that can insert and remove elements. When performed on a sequence, these cause the element to move to a new index. Take the following code for example, where the read from index 0 uses the write to

index 1:

```

1 block entry(x: Seq<i32>)
2   let x' = write 7 to x[1] in # write second element
3   let x'' = remove 0 from x' in # remove first element, x''[0] = x'[1]
4   let v = read x''[0] in # read first element
5   ret v

```

To provide a sound live element analysis, without being overly conservative, such cases must be handled. To resolve this, I introduce the *lineage function*, which describes how operations insert, remove and shift elements in the collection.

A lineage function takes an index as argument, and yields the new index where that element is stored. For example, the lineage function to push an element to the front of a sequence is:

$$\text{insert}(0) = \lambda x. x + 1$$

For cases where an element is not defined in the resulting collection, the lineage function yields $\perp$. This can be seen in the lineage function to remove an element from the front of a sequence:

$$\text{remove}(0) = \lambda x. \begin{cases} \perp & \text{if } x = 1 \\ x - 1 & \text{otherwise} \end{cases}$$

I provide a more thorough description of lineage functions in Section 6.1.1.

4.1.4 Live Element Analysis, Formally

I now provide a more formal definition of live element analysis. Live element analysis operates on the symbolic range lattice $R = \perp \leq [\ell : u] \leq \top$. Similarly

to prior work on range analysis, live element analysis is solved using a *constraints* graph.

A constraints graph is constructed from the def-use chains, where each edge $E = \overrightarrow{n_{\text{def}} n_{\text{use}}}$ is constructed from a def-use pair. Edges are annotated with a constraint L , which is constructed from the lineage function of the defining operation.

To construct a constraint from a lineage function, each branch is converted to a distinct edge. This is done by applying the branch result to the input lattice point. For example, the operation $y = \mathbf{insert\ 0\ into\ } x$ is converted into a single constraint:

$$R(y) - 1 \leq R(x)$$

In more complex cases, the lineage function may have multiple branches with various conditions. This is handled by taking the intersection $\wedge$ of the branch result with a range lattice approximating the branch condition. Each branch constraint is added to the graph. For example, $y = \mathbf{insert\ } i \mathbf{\ into\ } x$, where some elements are shifted and some are not, is converted into the constraints:

$$R(y) \wedge [-\infty : i] \leq R(x)$$

$$(R(y) \wedge [i : \infty]) - 1 \leq R(x)$$

For branches whose result is $\perp$, no constraint edge is added. For example, $y = \mathbf{remove\ } i \mathbf{\ from\ } x$ is converted into the constraints:

$$R(y) \wedge [-\infty : i] \leq R(x)$$

$$(R(y) \wedge [i : \infty]) + 1 \leq R(x)$$

The *intersection operator* $\wedge$, used above, intersects the ranges of two lattice points, defined as:

$$R_i \wedge R_j = [\ell_i : u_i] \wedge [\ell_j : u_j] \triangleq [\max(\ell_i, \ell_j) : \min(u_i, u_j)]$$

For example:

$$[0, 7] \wedge [3, 10] = [\max(0, 3) : \min(7, 10)] = [3, 7]$$

Later, I use the *union operator* $\vee$, which unions the ranges of two lattice points, defined as:

$$R_i \vee R_j = [\ell_i : u_i] \vee [\ell_j : u_j] \triangleq [\min(\ell_i, \ell_j) : \max(u_i, u_j)]$$

For example:

$$[0, 3] \vee [7, 10] = [\min(0, 7) : \max(3, 10)] = [0, 10]$$

These constraint edges are combined to create the constraints graph. The constraints graph is made context-sensitive by marking context-sensitive edges with their call site. The formal definition of the constraints graph is provided in definition 4.1.1.

Definition 4.1.1. A constraints graph, $G = (N, E, L, C)$, where: N is a set of vertices where n_a is a vertex; E is a set of edges where $\overrightarrow{n_a n_b}$ is an edge; $L : E \rightarrow S$ is a function from edges to constraints; $C : E \rightarrow c$ is a partial function from edges to call site c , representing context-sensitive edges.

A constraint graph is solved in a similar manner to prior work on range anal-

ysis, as proposed by Gawlitza et al. [48]. The algorithm for solving a given constraints graph is detailed in Algorithm 1.

As a result of live element analysis, each collection variable in the program is assigned a range lattice point. This represents the range of indices that contain live elements.

4.1.5 Dead Element Elimination, Formally

Given the result of live element analysis, the compiler proceeds by transforming the program to eliminate writes to elements that are provably unused. The transformation proceeds as follows.

For each $y = \mathbf{write} \ x[i]$ operation in the program, fetch the live element result $R(y) = [\ell : u]$. If $[0, \infty] \leq R(y)$ (i.e., all elements may be live) do not proceed.

After confirming that the collection has dead elements that can be eliminated, the algorithm checks if the index being written to may overlap with the live elements. This is done by taking the intersection of the index range and the live element result $R(i) \wedge R(y)$. If the result of this is an empty range, the **write** can be trivially eliminated.

For cases where the written elements cannot be proven dead in all cases, DEE inserts a conditional branch, such that the write is only performed if the index is within the live range. An example of this was shown in Listings 4.2 and 4.3.

In some cases, the range of live elements does not simplify to statically known constants. Instead, it may include program variables as symbols and arithmetic expressions that cannot be statically computed. To perform DEE in the face of these, the materialization function is used.

The *materialization function* $M(e, p) \rightarrow v$ is a partial function, which analyzes

ALGORITHM 1: Live element analysis by solving a constraints graph.

in: S a set of constraints

in: V a set of *sequence* variables.

in: $R(i) = [\ell : u]$ a mapping from *index* variable i to *range*
out: $R(v, c) = [\ell : u]$ a mapping from variable v and call site c to a *range*
let $G = (N, E, L, C)$
A constraints graph from definition 4.1.1
foreach $v \in V$ **do** create a vertex $n_v \in N$
foreach $i \in I$ **do** create a vertex $n_i \in N$, **let** $\rho(i) = R(i)$
foreach $s \in S$ s.t. $s = f(x) \sqsubseteq y$ **do**

 create an edge $\overrightarrow{n_x n_y} \in E$
let $L(\overrightarrow{n_x n_y}) = f(x) \vee \rho(x) \sqsubseteq y$
foreach $s \in S$ s.t. $s = [\ell : u] \sqsubseteq x$ **do**

 create a vertex $n_{[\ell : u]} \in N$
let $L(\overrightarrow{n_{[\ell : u]} n_x}) = [\ell : u] \sqsubseteq x$
foreach $s \in S$ s.t. $s = [a : b]$ *alias* x **do**

 create an edge $\overrightarrow{n_x n_{[a : b]}} \in E$
let c be the call site

let $C(\overrightarrow{n_x n_{[a : b]}}) = c$
let $L(\overrightarrow{n_x n_{[a : b]}}) = \rho([a : b], c) = \rho(x)$
call $\text{resolve_cycle}(G)$
function $\text{resolve_cycle}(G)$:

foreach $n_v \in G$ in topological order **do**
if n_v is a trivial SCC **then**
 $\rho(v) = \vee \{L(\overrightarrow{n_u n_v}) \mid \overrightarrow{n_u n_v} \in E\}.$
else
let SCC be the SCC containing n_v
H is the context-insensitive subgraph of SCC
let $H = \{\overrightarrow{n_x n_y} \in \text{SCC} \mid C(\overrightarrow{n_x n_y}) \text{ is undef.}\}$
call $\text{resolve_cycle}(n_w)$
foreach $n_v \in \text{SCC}$ s.t. $C(\overrightarrow{n_u n_v})$ is defined **do**
 $\rho(v, C(\overrightarrow{n_u n_v})) = \rho(u)$
foreach $n_v \in \text{SCC}$ s.t. $\rho(v)$ is undefined **do**
 $\rho(v, \text{undefined}) = [0 : \text{end}]$

an expression tree e at a given program point p , constructing the necessary operations and returning the resultant value v , which is either a variable or a constant. $M(e, p) = e$ iff e is a constant, a parameter of the function containing p , or a variable dominating p . $M(e, p) = g$ iff $\exists$ variable g which dominates p and has the same global value number [52] as e . $M(e, p) = op(M(e_1, p), \dots, M(e_n, p))$ iff e is an expression tree such that $e_i \subseteq e$ for all $i = 1, \dots, n$, op is an operation with no side effects and $M(e_i, p)$ are defined for all i . Otherwise, $M(e, p)$ is undefined.

When constructing the conditional, if either side of the range cannot be materialized, then the transformation does not apply.

4.2 Field Elision

Data structure splicing transformations optimize the memory layout of objects by reordering fields, splitting objects, and migrating fields between objects. In this section, I focus specifically on field migration.

Field migration takes multiple fields of an object and moves them into a separate object, replacing them with a pointer to the migrated field object. This is most beneficial when these fields are considerably colder than the others, or if they have a low *affinity* [53] with other hot fields in the object. By migrating them to their own object, the remaining fields with more highly correlated access are more spatially local—since the cache -since unrelated, adjacent fields are not loaded into the cache.

While field migration enables cache locality, it requires an additional pointer field to reference the migrated fields. This increases the size of the object and may lead to suboptimal cache locality, since the pointer field may be loaded into the cache. Furthermore, in order for migration to be profitable, the size of the

migrated fields must be greater than the size of the pointer field.

To enable field migration without the overhead of an additional pointer field, I propose *field elision*: a novel data structure splicing transformation that converts a field of an object into a key-value pair stored in an auxiliary map. By storing the field in a map, rather than directly in the object, the original object need not hold a pointer to the migrated fields. Furthermore, the field can be lazily allocated in the map, reducing memory usage when a field is infrequently needed, but not provably dead. The field elision (FE) transformation proceeds as follows.

Given a field within a tuple type $T.f$, construct a global map m . For each access to the field in the program, replace it with an access to the map m . Finally, remove field a from the tuple type T . An example of this transformation is shown below:

```

1 struct Node {
2     int hot;
3     int cold;
4 };
5 void work(Node *node, int niter) {
6     int a = node->cold;
7     for (int i = 0; i < niter; ++i)
8         node->hot += 1;
9 }

```

```

1 struct Node {
2     int hot;
3     static std::unordered_map<Node *, int> cold;
4 };
5 void work(Node *node, int niter) {
6     int a = Node::cold[node]; // access field via map
7     for (int i = 0; i < niter; ++i)
8         node->hot += 1;
9 }

```

4.3 Redundant Indirection Elimination

Redundant indirection elimination (RIE) simplifies indirect access (e.g., `a[b[i]]`) to maps when the index is derived from constant data (e.g., the elements of `b` are constant). When detected, the keys of a map can be replaced with the indices of another collection. This transforms the collection's type, removing an access to the index collection `b`. An example of this transformation for a C++ program is shown below:

```

const vector<Item> items;
unordered_map<Item, Val> histogram;
for (int i = 0; i < items.size(); ++i)
    histogram[keys[i]] += 1;

```

(a) Before RIE.

```

const vector<Item> items;
vector<Val> histogram(keys.size());
for (int i = 0; i < items.size(); ++i)
    histogram[i] += 1;

```

(b) After RIE.

Figure 4.1: C++ program computing the histogram of a list of items before and after applying redundant indirection elimination (RIE).

The transformation proceeds as follows. Given a map, $m = \mathbf{alloc} \text{ Map}<T, V>$, let R be the set of variables in the def-use chain of m . If any variable in R *may*, but not *must*, reference m (e.g., control divergence), RIE is not applicable. Otherwise, the transformation checks all accesses $r[k]^1$, where $r \in R$. If all keys are of the form $k = \mathbf{read} \ c[i]$, where all instances of c *must* refer to the same collection, then:

1. If c is of type $\mathbf{Seq}<T>$, construct $c' = \mathbf{alloc} \ \mathbf{Seq}<V> \ \mathbf{size}(c)$.
2. If c is of type $\mathbf{Map}<U, T>$, construct $c' = \mathbf{alloc} \ \mathbf{Assoc}<U, V>$.

Finally, for all $r \in R$, replace accesses $r[k]$ with $c'[i]$, where $k = c[i]$.

4.4 Evaluation

In this evaluation, I answer the following research questions:

¹The notation $c[i]$ refers to any **read**, **write** or **has** operation that uses collection c at index i .

RQ1 Do MEMOIR optimizations reduce execution time and memory usage?

RQ2 Do MEMOIR optimizations have cascading benefits, as seen in conventional compiler transformations?

4.4.1 Experimental Setup

This evaluation was performed on a server with two Intel Xeon Gold 6258R (Cascade) processors running at 2.70GHz. Each processor has 28 cores with 2-way hyper-threading, 32KiB 8-way L1d\$ and 1MiB 16-way L2\$, backed by 38.5MiB 11-way LL\$, all having a 64B line size. They are supported by 512GiB of DDR4 main memory running at 2933MT/s. The OS used is Red Hat Enterprise Linux 8.7.

I evaluate against four production-quality compilers: GCC 8.5.0 (**GCC**), ICC 18.0.1 (**ICC**), LLVM 9.0.0 (**LLVM9**), and LLVM 14.0.6 (**LLVM14**). All performance results are gathered from 10 executions per configuration, all values are the median of those executions relative to **LLVM9**.

I implement the optimizations described in this chapter in the MEMOIR compiler using LLVM 9.0.0 with abstractions from NOELLE [54]. The input programs are compiled using clang 9.0.0. The O3 optimization pipeline is applied before running MEMOIR optimizations.

I evaluate on benchmarks from the SPEC CPU 2017 benchmark suite [55], the industry standard benchmark suite used to compare compiler optimizations. Only mcf_s and deepsjeng_s are evaluated, since raising high-level collections out of these highly optimized, low-level implementations in C/C++ is a difficult manual process. For reference these benchmarks run to completion in approximately 11 and 12 minutes, respectively, on our machine when compiled with **LLVM9**. At the time of writing, agent-based rewriting using large language models also failed

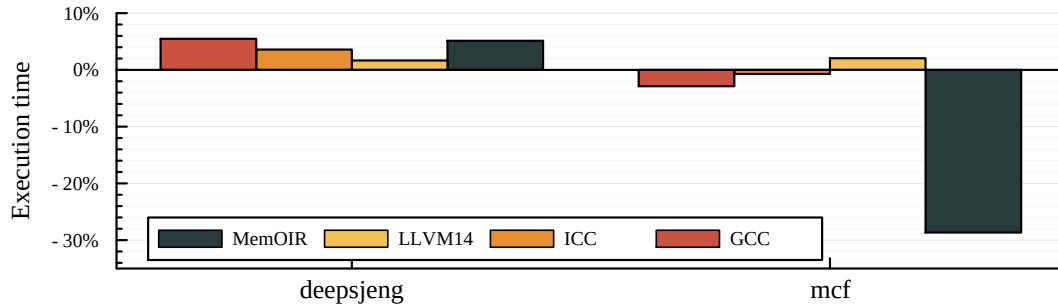


Figure 4.2: Relative execution time of ported SPECINT 2017 benchmarks

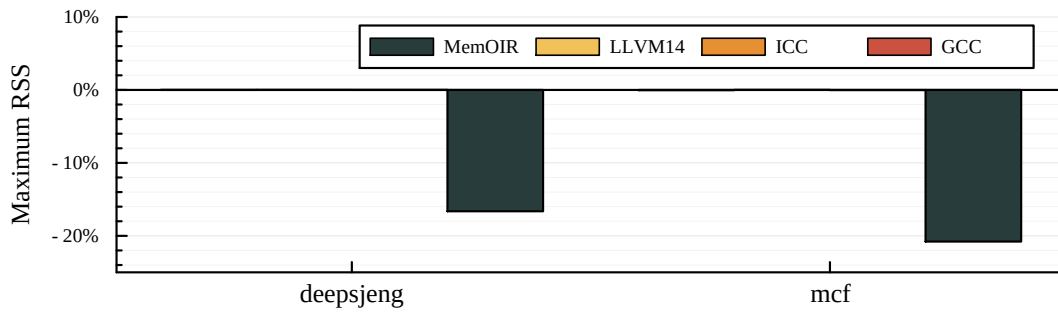


Figure 4.3: Relative memory usage of ported SPECINT 2017 benchmarks.

to correctly raise to these abstractions, however improving the ergonomics of the MEMOIR library—see Chapter 6—and steady advancements in language models could enable fully automatic rewriting.

RQ1: Do MEMOIR optimizations reduce execution time and memory usage?

I first evaluate a compilation pipeline containing DEE, DFE, FE and RIE run in sequence. The results of this experiment are shown in Figures 4.2 and 4.3.

I find that MEMOIR optimizations are able to reduce the memory usage of both mcf and deepsjeng, while only mcf sees a reduction in execution time. For deepsjeng, only field elision and key folding were applicable, eliding a 16-bit field from the hottest data structure. This allowed for better packing of the struct in memory, reducing the memory usage by 16.6% but entailing a 5.1% increase in execution time due to worse cache locality. Additional optimization using traditional data

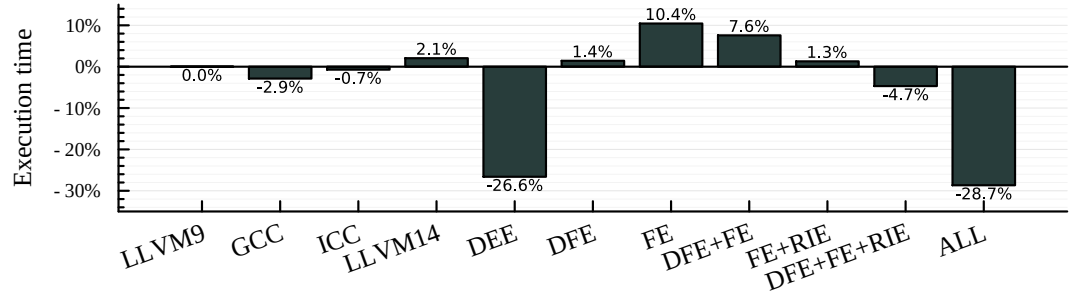


Figure 4.4: Relative execution time for breakdown of mcf optimizations.

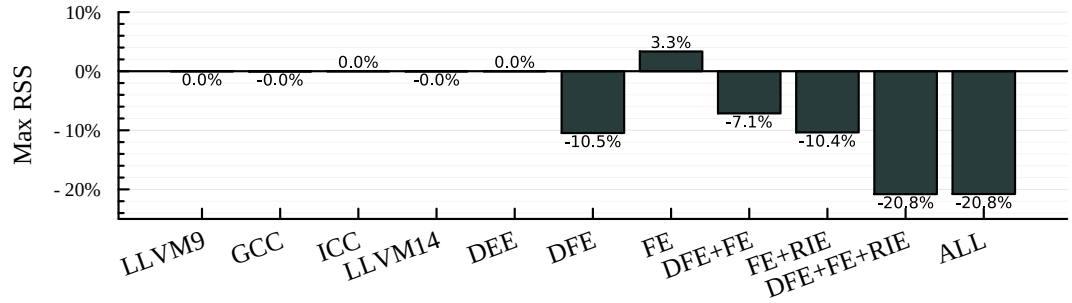


Figure 4.5: Relative memory usage for breakdown of mcf optimizations.

structure splicing transformations could be useful to improve performance with the smaller memory layout.

RQ2: Do MEMOIR optimizations have cascading benefits, as seen in conventional compiler transformations?

Having found that MEMOIR optimizations lead to considerable performance and memory usage improvements, I explore each optimization individually and in concert with others. I focus on the mcf benchmark, since all of the proposed optimizations are applicable there. To explore their individual impact and interplay, I run an ablation study over all combinations of optimizations (modulo ordering). The results of this study are shown in Figures 4.4 and 4.5.

DEE results in a 26.6% speedup over LLVM9 by reducing the computational complexity of mcf’s quick sort on a sequence of length $n = K + B$ from $O(n \log n)$ to $O(n + B \log B)$. This is effectively performed by DEE with no notion of the quick

sort algorithm, solely using the logical operations represented in MEMOIR.

FE, in the absence of other optimizations, causes a 10.4% increase in execution time and a 3.3% increase in max RSS, where a single pointer field is elided. The resulting associative array is lowered to a hash table, whose key-value store causes increases memory usage and expansion of the table causes the increased execution time. When combined with RIE, the positive impact of FE can be seen with a 10.4% decrease in max RSS with only a 1.3% increase in execution time, as the associative array is converted into a single sequence. This removes the storage of the key, only needing to store the value.

When combined with DFE the memory size of the object is shrunk to 56 bytes, allowing fields of more than one object to be stored on the same cache line. Because accesses to these objects are filtered by the value of the first field (`id`), filtered iterations (i.e., where the rest of the object is not read), can be processed along with the previous iteration, achieving a 4.7% speedup over the baseline. The resulting struct layout, and a visualization of the cache behavior after each optimization, are shown in Figure 4.6. This combination also greatly reduces the memory usage of the benchmark, reducing the max RSS by 20.8%, approximately 844MiB. When **ALL** optimizations are applied an additional 2.1% speedup is seen over solely **DEE** and max RSS reduction remains 20.8%.

4.5 Discussion

In this chapter, I showed how MEMOIR enables a variety of optimizations on both compute and memory by providing a logical representation for data collections. These optimizations are driven by efficient static analyses, which make full use of the sparse representation of dataflow in SSA form. Importantly, these optimiza-

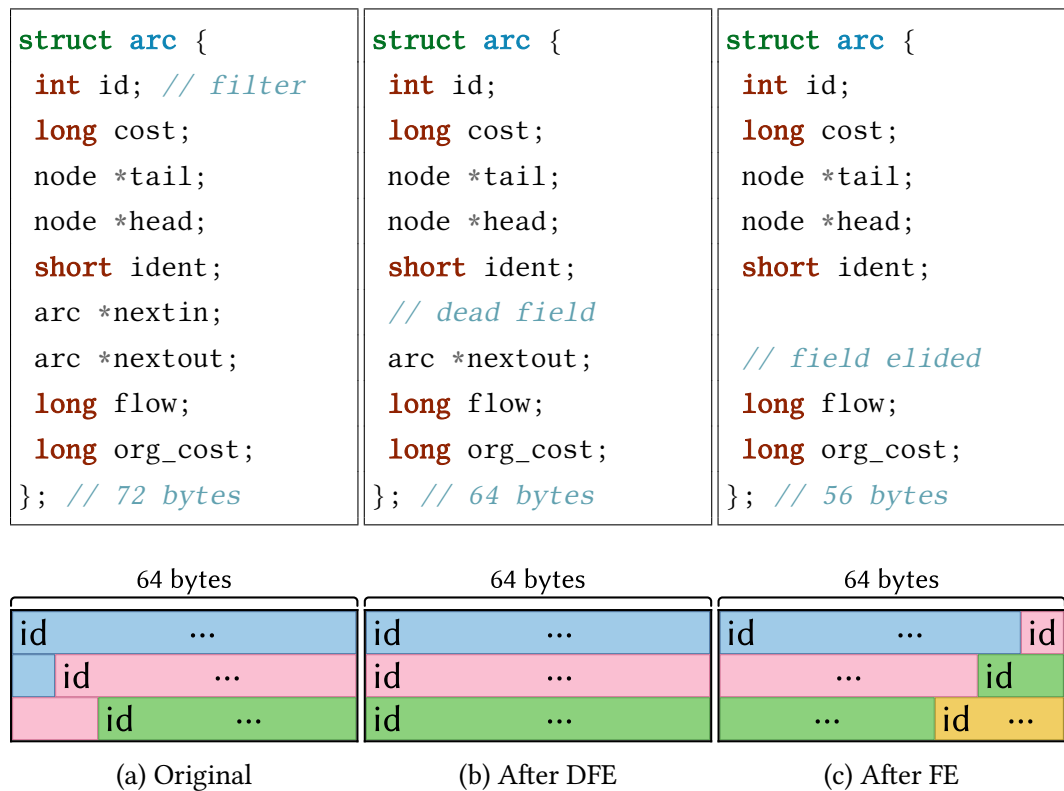


Figure 4.6: MEMOIR layout optimizations applied to hot struct in mcf.

tions are all implementation-agnostic, they apply uniformly across maps, sets, sequences and fields.

I demonstrated that these optimizations have a considerable impact on both the performance and memory usage of C/C++ programs in industry-standard benchmarks. In comparison, production C/C++ compilers used in production have negligible impact on the performance of these benchmarks, with newer versions actually seeing considerable performance degradations. Furthermore, none of these production compilers are capable of meaningfully reducing memory usage, since their low-level memory representations make anything but stack objects impossible to effectively transform. This illustrates how the collection-oriented design of MEMOIR enables both conventional data layout optimizations and novel optimizations on element-level computation and data organization.

While production compilers fall short of the optimizations made possible with MEMOIR, similar memory optimizations have been automated in research compilers.

Forma [56] provides a compiler framework for automatically detecting type-safe arrays in a C/C++ program and performing array reshaping on them. In their evaluation, Forma is only able to tackle one array in each of their four benchmarks. It is important to note that the array reshaping optimizations performed by Forma are a subset of those described in this chapter.

Automatic Pool Allocation [57] proposes a transformation framework to coalesce multiple object allocations into memory pools. These memory pools provide increased data locality and enable a limited number of data layout optimizations to be performed on objects within a given pool. MPADS [58] extends their work to perform a larger set of data layout optimizations on each of the memory pools for additional data locality.

A bevy of prior works on data structure splicing have optimized the layout of fields in memory objects [17], [53], [59], [60], [61]. Such transformations include object splitting, object merging, pointer-field inlining, field reordering, and field migration. In this chapter, I demonstrated how the collection-oriented view enables *field elision*: a novel data structure splicing optimization that migrates fields to a new collection instead of another object. Field elision goes beyond field migration, removing the need for additional pointer fields, and reducing memory usage by lazily allocating fields by need.

CHAPTER 5

Unlocking Specialized Collection Implementations

When developers use data collections (e.g., lists, sets, maps), they must choose how to implement them. While custom implementations allow hand-tuned performance and memory use, many developers rely on general-purpose implementations provided by the language or libraries. For example, a C++ developer can select either a `std::vector` (array) or a `std::list` (linked list) to optimize common operations. This process, called *collection selection*, significantly impacts both performance and memory usage [62], [63]. Consequently, tools exist to guide developers through this optimization space [63], [64], [65], [66], [67], [68], [69], [70], [71], [72].

Furthermore, collection implementations can be specialized to leverage properties of the data they store. For example, if a certain key is never used in a hashtable, it can be used as a tombstone key to mark slots that are empty or removed, eliding the need for an extra bit to mark each slot. Without such properties, selections like this are left entirely unexplored. Implicit knowledge of properties is a hallmark of domain-specific languages, naturally enabling specialized imple-

This chapter is adapted from “Automatic Data Enumeration for Fast Collections” in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2026.

mentations. However, for general-purpose applications, these properties do not exist naturally. In these cases, developers must *manufacture* the necessary property by normalizing their data if they wish to employ a specialized implementation.

A classic example of this is *string interning*, where each string is assigned a unique identifier. Interning is useful when strings serve as map keys or set items, avoiding an $O(n)$ multiplicative factor on all search operations. This pattern is commonly seen in compilers and language runtimes, with the earliest documented usage in Common Lisp [73] and Scheme [74]. It persists today in modern languages such as Java [75], Lua [76], and Clojure [77]. However, interning applies to more than strings. Java interns boxed booleans, and some integers, to reduce memory usage [78]. Additionally, Go’s `unique` package [79] lets developers intern comparable data types, exposing a `Handle` type for identifiers. More generally, the flyweight design pattern [80] is used by developers to deduplicate shared immutable data, an example being LLVM’s storage of constant values.

Data enumeration generalizes interning over arbitrary data types with an additional constraint that all unique identifiers be contiguous, i.e., in the range $[0, N)$ where N is the number of unique data items. With this property, maps and sets over the data can be directly converted into arrays (à la bitsets) where a key’s identifier is used as its index into the array. Converting sets and maps into arrays is the primary benefit of data enumeration.

For example, bitsets provide a compact and performant solution to store finite sets over a contiguous range of integers, e.g., $[0, N)$. This property can be manufactured with *data enumeration*, where each item in the domain of a set is assigned a unique integer identifier. These identifiers can then be used as the domain of a bitset. Consider the following code, where all unique items in an array are inserted into a set and printed:

```

1 array : List[str] = ["foo", "bar", "foo"]
2 set: Set[str] = {}
3 for v in array:
4     if v not in set:
5         set.add(v)
6         print(v)

```

This program can be transformed to use data enumeration (shown below). First, a mapping from identifiers to the original data is created (line 1). Items in the array are replaced with their identifiers (line 2). Then, the set is replaced with a bitset over the identifiers (line 3). The original data is retrieved by indexing into the mapping with the identifier (line 7).

```

1 enumn : Dict[int, str] = {0->"foo", 1->"bar"}
2 array : List[int] = [0, 1, 0]
3 set: BitSet = {}
4 for v in array:
5     if v not in set:
6         set.add(v)
7         print(enumn[v])

```

While data enumeration may seem straightforward, it can be unwieldy to perform manually. Developers must balance the cost of translating between data and identifiers, ensuring that operations are performed on the correct form. In large projects, keeping usage consistent across modules is difficult, risking subtle logic errors, while evolving features and refactoring can introduce performance bugs as boundary choices become stale.

To ameliorate these issues, I propose *automatic data enumeration (ADE)*. With ADE, the compiler creates the enumeration mapping, inserts translations between

data and identifiers, and replaces sets with bitsets. I introduce optimization techniques to eliminate redundant translations and employ static heuristics to decide when to use either identifiers or the original data. With these techniques, ADE achieves speedups of 2.16× on average (up to 8.72×) and reduces peak memory consumption by 5.6% on average (up to 50.7%) on a wide range of benchmarks. Furthermore, user directives enable performance engineering for up to 78.1× speedup and a 70.1% peak memory reduction.

This chapter proceeds as follows: Section 5.1 provides a high-level description of how ADE optimizes a program, and the optimizations that make ADE performant in practice. Section 5.2 details the implementation of ADE in the MEMOIR compiler. Section 5.3 evaluates the impact of ADE on performance and memory usage across multiple architectures.

5.1 Automatic Data Enumeration, Informally

Before going into the details of automatic data enumeration (ADE), I discuss the high-level approach.

Associative collections map keys of an arbitrary type to values. While their definition affords greater flexibility than sequential collections, accessing them requires indirection (e.g., search trees or probes). Fundamentally, this arises from mapping a sparse or noncontiguous domain into memory:

$$\mathbb{K} \xrightarrow{\text{sparse}} \mathbb{V}$$

I address this limitation by decomposing associative collections into a sparse

mapping and a dense mapping:

$$\mathbb{K} \xrightarrow{\text{sparse}} \mathbb{E} \xrightarrow{\text{dense}} \mathbb{V}, \text{ where } \mathbb{E} = [0, |\mathbb{K}|)$$

$\mathbb{K} \rightarrow \mathbb{E}$ assigns keys to a contiguous range $\mathbb{E}$, which is then mapped to values via $\mathbb{E} \rightarrow \mathbb{V}$. The contiguity of $\mathbb{E}$ enables the values of $\mathbb{E} \rightarrow \mathbb{V}$ to be efficiently stored in an array-like data structure. I refer to the mapping $\mathbb{K} \rightarrow \mathbb{E}$ as the *enumeration* of $\mathbb{K}$, and $\mathbb{E} \rightarrow \mathbb{V}$ as the *enumerated* collection.

As discussed earlier, adding this level of indirection alone doubles the number of accesses in the program. The benefits of ADE are only realized when combined with purpose-designed optimizations. The primary optimization driver is *redundant translation elimination*, where translations between raw data and identifiers can be eliminated by replacing them with an equivalent algebraic expression. While redundant translation elimination can render many benefits, it shines more when combined with transformations that induce latency redundancy for further elimination.

Propagation transforms other data collections to store identifiers instead of raw data. An example of this was shown in the introduction, where array was transformed from a list of strings to a list of identifiers. This introduced redundancy that could be exploited within the loop, since the set could directly add the elements of array without needing to encode them into the enumeration. This optimization is only applied when the other data collection *solely* stores raw data from the enumerated collection, since all stored values will be encoded and decoded uniformly.

Sharing reuses the enumeration across multiple collections by merging their keys into a single enumeration. This amortizes the cost of constructing the enumeration, and renders more translations redundant for elimination. However, this

could make an enumerated collection effectively, since the enumeration stores the union of multiple key spaces. This would lead to the array-like data structures being over-provisioned, wasting memory.

5.2 Automatic Data Enumeration, Formally

In this section I describe our implementation of automatic data enumeration in the MEMOIR compiler. Section 5.2.1 describes how to transparently transform a program with automatic data enumeration. I describe optimizations to eliminate redundancy (Section 5.2.2) and transformations that uncover latent redundancy (Sections 5.2.3 and 5.2.4). Next, I explain how ADE handles function calls (Section 5.2.5) and nested collections (Section 5.2.6). Then, Section 5.2.7 presents the collection implementations explored in our system, including details on the specialized implementations enabled by data enumeration. Finally, Section 5.2.8 introduces optimization directives to enable performance engineering with ADE.

5.2.1 Inserting enumerations and Translations

I now describe the framework for automatic data enumeration before delving into optimizations. Listing 5.1 serves as an example MEMOIR program being transformed. As a result of our example transformation, I will enumerate %hist.

```

1  block entry(input: Seq<f32>)
2      let empty = alloc Map<f32,i32> in # construct an empty histogram
3      let first = begin of input in # construct iterator to beginning of input
4      jmp header(first, empty)
5  block header(it: Iter<i64,i32>, hist: Map<f32,i32>)
6      let last = end of input in
7      let done = eq it last of input in # have we reached the end of input?
8      if0 done body() exit()
9  block body()
10     let present = has hist[val] in # check if val is already tracked
11     if0 present then() else()
12  block then()
13     let freq = read hist[val] in # already tracked, get current frequency
14     jmp cont(hist, freq)
15  block else()
16     let hist1 = insert val into hist # not tracked, insert
17     jmp cont(hist', 0)
18  block cont(hist2: Map<f32,i32>, cur_freq)
19     let new_freq = add cur_freq 1 in # increment the frequency
20     let hist3 = write new_freq to hist2[val] in # update the histogram
21     let it' = next it of input in # traverse to next input element
22     jmp header(it', hist3)

```

Listing 5.1: MEMOIR program that computes a histogram from a sequence of integers.

First, the enumeration `enumn: Enum<f32>` is created with type:

`Enum<T> = (Enc<T>, Dec<T>), where`

`Enc<T> = Map<T,i64> and Dec<T> = Seq<T>`

`Enc<T>` maps values to unique identifiers and `Dec<T>` provides the reverse mapping. The enumeration is populated on the fly as new values are encountered. I introduce three translation functions:

`fn enc(Enum<T>,T) -> i64` translates a value to its identifier. Behavior is undefined if the value is not in the enumeration.

`fn dec(Enum<T>,i64) -> T` translates an identifier to its value. Behavior is undefined if the identifier is not in the enumeration.

`fn add(Enum<T>,T) -> (Enum,i64)` adds a value to the enumeration, returning the updated enumeration and the new identifier. If the value is already in the enumeration, returns the input enumeration and the value's identifier.

To transform the program, the compiler must know where to perform translations. Algorithm 2 generates three sets—*ToEnc*, *ToDec* and *ToAdd*—with use sites that must be patched by translating the value with a call to the respective function.

After collecting the uses to patch, the program is transformed to the one shown in Listing 5.2. An empty enumeration is allocated (line 2). The type of `hist` is transformed to `Map{BitMap}<i64,i32>` (line 3). All use sites are patched by calling the helper function (lines 12, 15, 19 and 23).

A keen eye will notice that, at this point, I have not optimized the behavior of the program. Rather, I have solely added a level of indirection! In the remainder of this section, I discuss optimizations that make ADE performant.

```

1 block entry(input: Seq<f32>)
2   let empty_enumn = alloc Enum<f32> # construct enumeration
3   let empty = alloc Map<f32,i32> as BitMap in # select BitMap
4   let first = begin of input in
5   jmp header(first, empty, empty_enumn)
6 block header(it: Iter<i64,i32>, hist: Map<f32,i32>, enumn: Enum<f32>)
7   let last = end of input in
8   let done = eq it last of input in
9   if0 done body() exit()
10 block body()
11   let id = call enc(e, val) # encode raw value to ID
12   let present = has hist[id] in
13   if0 present then() else()
14 block then()
15   let old_freq = read hist[id] in
16   jmp cont(hist, old_freq, enumn, id)
17 block else()
18   let fresh_enumn, fresh_id = call add(enumn, val) in # assign ID
19   let hist1 = insert fresh_id into hist
20   jmp cont(hist', 0, fresh_enumn, fresh_id)
21 block cont(hist2: Map<f32,i32>, freq: i32, enumn': Enum<f32>, id': i64)
22   let new_freq = add freq 1 in
23   let hist3 = write new_freq to hist2[id'] in # update to use identifier
24   let it' = next it of input in
25   jmp header(it', hist3, enumn')

```

Listing 5.2: Compute histogram of sequence with ADE.

5.2.2 Eliminating Redundant Translations

Redundant translation elimination (RTE) is the primary driver for optimizing performance with ADE. Informally, RTE leverages the properties of the `enc`, `dec` and `add` functions to apply rewrite rules. The first leverages the fact that `dec` is the inverse function of `enc`, allowing the rewrite:

ALGORITHM 2: Uses to patch for enumerated collection.

in: v associative collection variable

out: $ToEnc(v)$, $ToDec(v)$, $ToAdd(v)$

foreach $r \in Redefs(v)$ **do**

foreach $u \in Uses(r)$ **do**

switch $User(u)$ **do**

case $\boxed{\text{read } r[k]}$ **do**

insert use of k **in** $ToEnc(v)$

case $\boxed{\text{has } r[k]}$ **do**

insert use of k **in** $ToEnc(v)$

case $\boxed{\text{remove } k \text{ from } r}$ **do**

insert use of k **in** $ToEnc(v)$

case $\boxed{\text{write } v \text{ to } r[k]}$ **do**

insert use of k **in** $ToEnc(v)$

case $\boxed{\text{insert } k \text{ into } r}$ **do**

insert use of k **in** $ToAdd(v)$

case $\boxed{\text{let } k = \text{getkey it of } x \text{ in } \dots}$ **do**

insert $Uses(k)$ **in** $ToDec(v)$

case $\boxed{\text{op}(r[k] \dots, \dots)}$ **do** # *Nesting, see Section 5.2.6*

insert use of k **in** $ToEnc(v)$

$$\mathbf{call\ enc}(e, \mathbf{call\ dec}(e, x)) \rightarrow x$$

Additionally, if a value needs to be decoded, then it is already in the enumeration, allowing the rewrite:

$$\mathbf{call\ add}(e, \mathbf{call\ dec}(e, x)) \rightarrow (e, x)$$

Finally, since values are never removed from the enumeration and *dec* is injective, allowing the rewrite:

$$\mathbf{eq}(\mathbf{call\ dec}(e, x), \mathbf{call\ dec}(e, y)) \rightarrow \mathbf{eq}(x, y)$$

This analysis is described with more detail in Algorithm 3, which collects redundant translations into *TrimEnc*, *TrimDec* and *TrimAdd*. These are used to recompute *ToEnc*, *ToDec* and *ToAdd*, by subtracting the corresponding *Trim* set.

I employ a static heuristic to estimate the benefit of ADE. Informally, enumeration is beneficial iff ADE can find redundant translations. To compute the heuristic, I perform a *what if* analysis using the results of the `FINDREDUNDANT` function:

$$|TrimEnc| + |TrimDec| + |TrimAdd|$$

This heuristic could be extended with profile information, however this static heuristic has sufficed for our use cases.

5.2.3 Sharing enumerations Between Collections

Although enumeration can benefit a single collection, it is common for multiple collections to operate on the same data. To exploit this, ADE *shares* an enumeration between multiple enumerated collections. This amortizes the overhead of

ALGORITHM 3: Identify redundant translations.

function FINDREDUNDANT(*ToEnc*, *ToDec*, *ToAdd*):
foreach $u \in ToDec$ **do**
 | **if** $u \in ToEnc$ **then** # *Encoding a decoded value*
 | | **insert** u **in** *TrimDec*
 | | **insert** u **in** *TrimEnc*
 | **else if** $u \in ToAdd$ **then** # *Already enumerated*
 | | **insert** u **in** *TrimDec*
 | | **insert** u **in** *TrimAdd*
 | **else if** $User(u)$ **is** $eq(x, y) \wedge x = UsedBy(u)$ **then**
 | | **let** $w = \text{use of } y \text{ in } User(u)$
 | | **if** $w \in ToDec$ **then** # *Comparing enumerated values*
 | | | **insert** u **in** *TrimDec*
 | | | **insert** w **in** *TrimDec*
return *TrimEnc*, *TrimDec*, *TrimAdd*

constructing the enumeration and uncovers redundant translations to eliminate. In this section, I present algorithms and heuristics to identify cases where sharing is beneficial.

I use the term *candidate* to refer to a group of collections that share an enumeration. Informally, I identify a candidate as the maximal set of collections that improve the benefit heuristic. For a collection to be in a candidate, its inclusion must make the benefit of the candidate greater than the sum of its parts. The key insight being that, if two collections are sharing data, they are likely to be constructed from one another or have their values compared. If this is the case, redundant translations will surface when computing the benefit. Algorithm 4 shows the analysis in more detail.

ALGORITHM 4: Find candidates for enumeration sharing.

in: A set of collection allocations in the function

out : *Candidates* set of collections to share an enumeration

Track the allocations that already belong to a candidate.

let *Used* = $\emptyset$

Create a candidate for each collection in the function.

foreach $a \in A$ **where** $a \notin \textit{Used} \wedge a$ **isa** $\text{Assoc}\langle T, U \rangle$ **do**

let $C = \{a\}$

insert a **in** *Used*

Find other collections that would benefit the candidate.

foreach $b \in A$ **where** $b \notin \textit{Used}$ **do**

if $\text{CANSHARE}(b, T) \vee \text{CANPROPAGATE}(b, T)$ **then**

let $B_{\Sigma} = \text{BENEFIT}(C) + \text{BENEFIT}(\{b\})$

let $B_{\cup} = \text{BENEFIT}(C \cup \{b\})$

Benefit must be greater than the sum of its parts.

if $B_{\cup} > B_{\Sigma}$ **then insert** b **in** C ; **insert** b **in** *Used*

Only emit candidates with positive benefit.

if $\text{BENEFIT}(C) > 0$ **then insert** C **in** *Candidates*

Benefit of sharing an enumeration between collections.

function $\text{BENEFIT}(C)$:

let $\text{TrimEnc}, \text{TrimDec}, \text{TrimAdd} = \text{TRIM}\left(\bigcup_{c \in C} \text{ToEnc}(c), \bigcup_{c \in C} \text{ToDec}(c), \bigcup_{c \in C} \text{ToAdd}(c)\right)$

return $|\text{TrimEnc}| + |\text{TrimDec}| + |\text{TrimAdd}|$

Section 5.2.3: Key type of collection must match to share.

function $\text{CANSHARE}(x, T)$:

return x **isa** $\text{Assoc}\langle T, V \rangle$

Section 5.2.4: Element type of collection must match to propagate.

function $\text{CANPROPAGATE}(x, T)$:

return x **isa** $\text{Assoc}\langle K, T \rangle \vee x$ **isa** $\text{Seq}\langle T \rangle$

5.2.4 Propagating Identifiers through Collection Elements

In addition to sharing an enumeration between the keys of sets and maps, any collection can be transformed to store identifiers in its elements using *propagation*. Consider Listing 5.3, which iteratively searches the %uf map using the value read in the last step as the key in the next step.

```

1  block entry(uf: Map<i64,i32>, v: i32, e: Enum<i32>)
2      let id_v = call enc(e, v) in
3      jmp loop(v, id_v, e)
4
5  block loop(current: i32, current_id: i64, e': Enum<i32>)
6      let parent = read uf[current_id] in
7      let e", parent_id = call add(e', parent) in
8      let done = cmp eq parent_id current_id in
9      if0 done loop(parent, parent_id, e") exit(parent)
10
11 block exit(found: i32)
12     ret found

```

Listing 5.3: Find parent in union-find *without* propagation.

If ADE is naïvely applied this code, the key is translated on each iteration, adding costly accesses to a potentially hot loop. Propagation addresses this by mutating %uf to store identifiers for both its keys *and* values. Listing 5.4 shows the same program with propagation applied to the elements of %uf. As a result, there is no translation in the loop. It has been replaced by a single decode outside

the loop.

```

1  block entry(uf: Map<i64,i64>, v: i32, e: Enum<i32>)
2      let id_v = call enc(e, v) in
3      jmp loop(v, id_v, e)
4
5  block loop(current: i32, current_id: i64, e': Enum<i32>)
6      let parent_id = read uf[current_id] in
7      let done = cmp eq parent_id current_id in
8      if0 done loop(parent, parent_id, e") exit(parent)
9
10 block exit(found: i32)
11 ret found

```

Listing 5.4: Find parent in union-find *with* propagation.

I refer to a collection storing identifiers in its elements as a *propagator*. Algorithm 5 details how ADE identify the uses to patch for propagators. Since ADE cannot determine that a given value is an identifier or not at runtime, propagators must store identifiers in *all* elements. As such, all values written to the collection must be added to the enumeration. Algorithm 4 shows how ADE applies the benefit heuristic to identify beneficial propagators for a candidate. I employ the same heuristic for propagation and sharing, as the key insight holds for both.

ALGORITHM 5: Uses to patch for propagator.

```

in:  $v$  a collection variable
out:  $ToEnc(v)$ ,  $ToDec(v)$ ,  $ToAdd(v)$  uses to patch.
foreach  $r \in Redefs(v)$  do
  foreach  $u \in Uses(r)$  do
    switch  $User(u)$  do
      case  $v := read(r, k)$  do
        insert  $Uses(v)$  in  $ToDec(v)$ 
      case  $write(v, r, k)$  do
        insert use of  $k$  in  $ToAdd(v)$ 
      case for  $[k, v]$  in  $r$ : ... do
        insert  $Uses(v)$  in  $ToDec(v)$ 

```

5.2.5 Interprocedural Transformation

To handle enumerated collections passed as arguments, the compiler determine which enumeration to use for translation in the callee, detailed in Algorithm 6. This generates a set of equivalence classes, where items in a class share the same enumeration. Each class is given a global variable to store the enumeration. When patching a use, the enumeration is loaded from this global variable for the translation.

For recursive functions, ADE reuses the enumeration at each invocation to avoid construction overhead. Without reuse, constructing an enumeration at each invocation led to timeouts and significant memory overhead for our benchmarks.

For functions that are externally visible, or have parameters that are only enumerated for some callers, the function being transformed is cloned. ADE does not apply to collections passed into calls with indirect or externally defined callees, nor collections that escape into unknown memory locations. This ensures correct translation for all uses and functionally prevents transformation of collections shared among parallel threads, but permits correct handling of sequential

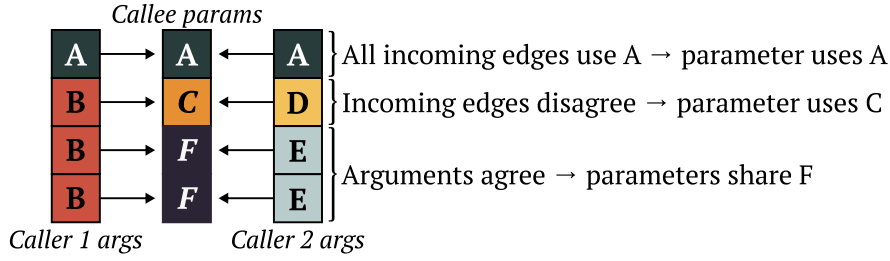


Figure 5.1: Visualization of unifying parameters in Algorithm 6.

programs and thread-local collections.

ALGORITHM 6: Unify collections that share an enumeration.

in: V the set of all candidates

in: F the set of functions with enumerated parameters

out: *EnumClass*: equivalence classes for each enumeration

Unify redefinitions.

let U be a union-find data structure

foreach $r \in \text{Redefs}(v)$ **where** $v \in V$ **do**

└ **unify** v **with** r **in** U

Unify parameters, see Figure 5.1 for a visual aide.

foreach $p \in \text{Params}(f)$ **where** $f \in F$ **do**

└ *# Unify parameters where all arguments agree.*

foreach $a \in \text{Incoming}(p)$ **where** $U.\text{find}(a) = b$ **do**

 └ **unify** p **with** b **in** U

Unify parameters whose arguments are unified at each call.

if $\exists q \in \text{Params}(f)$ **where**

$\forall c \in \text{Callers}(f). U.\text{find}(c.\text{arg}(p)) = U.\text{find}(c.\text{arg}(q))$ **then**

 └ **unify** p **with** q **in** U

Compute the equivalence classes from the union-find.

let *EnumClass* = **classes**(U)

5.2.6 Enumerating Nested Collections

If a nested collection is enumerated, all collections at that nesting level share an enumeration. For example, given `%x:Seq<Set<f32>>`, `%x[0]` and `%x[1]` use

the same enumeration. To ensure that all keys are properly translated, nested operands are patched indices as shown in the last **case** statement of Algorithm 2.

5.2.7 Collection Selection

ADE makes specialized implementations possible by manufacturing the necessary properties, expanding the selection space. Table 5.1 outlines the implementations available to each type, where the **Enum’d?** column marks specialized implementations for enumerated collections. The rest of this section describes those specialized implementations in more detail.

`BitSet` is implemented with a `boost::dynamic_bitset`, a contiguous array of bits that supports dynamic resizing. Dynamic resizing is necessary for our application since enumerations are constructed on the fly. `BitMap` is implemented similarly, with a contiguous array to hold the values.

`SparseBitSet` is implemented with Roaring [81], a popular implementation used by Apache Lucene [82], Youtube [83], and academia [84]. Roaring uses a hybrid compression technique, combining uncompressed bitsets, packed arrays and run-length encoding to achieve a performant yet compact implementation.

5.2.8 Performance Engineering with Optimization Directives

Modern compilers are becoming open boxes [85], [86], [87], where performance engineers are given the ability to dictate optimization plans to the compiler. In this vein, I provide directives to specify candidates and constraints that override the benefit heuristic. This allows performance engineers to fine-tune ADE, while still benefiting from fully automatic transformation.

Examples of these directive are shown in Listing 5.5. Users can explicitly en-

Table 5.1: Collection selection candidates and the average case complexity of basic operations, where n is the number of elements in the collection k is the largest integer key in the collection.

Type	Enum'd?	Selection	Description	Rd/Wr	Insert	Remove	Storage
Seq<T>	✗	Array	Resizable array	1	$O(n)$	$O(n)$	$n \cdot \mathbf{bits}(T)$
Set<T>	✗	HashSet	Hash table	—	$O(1)$	$O(1)$	$O(n \cdot \mathbf{bits}(T))$
⋮	✗	FlatSet	Sorted array	—	$\log(n)$	$\log(n)$	$n \cdot \mathbf{bits}(T)$
	✗	SwissSet	Swiss table	—	$O(1)$	$O(1)$	$O(n \cdot (1 + \mathbf{bits}(T)))$
	✓	BitSet	See Section 5.2.7	—	1	1	k
	✓	SparseBitSet	See Section 5.2.7	—	$O(1)$	$O(k)$	$O(k)$
Map<K,T>	✗	HashMap	Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n \cdot (\mathbf{bits}(K) + \mathbf{bits}(T)))$
	✗	SwissMap	Swiss table	$O(1)$	$O(1)$	$O(1)$	$O(n \cdot (1 + \mathbf{bits}(K) + \mathbf{bits}(T)))$
	✓	BitMap	See Section 5.2.7	1	1	1	$k \cdot (1 + \mathbf{bits}(T))$

```

1  #pragma ade enumerate
2  a = alloc Set<i32>
3  #pragma ade noenumerate
4  b = alloc Map<i32, i32>
5  #pragma ade noshare(a)
6  c = alloc Set<i32>
7  #pragma ade share group("d+e group")
8  d = alloc Set<i32>
9  #pragma ade share group("d+e group")
10 e = alloc Set<i32>

```

Listing 5.5: Directives allow users to control and guide ADE.

able (line 1) or disable (line 3) data enumeration for a collection. Additionally, they can specify that a collection should not share an enumeration with another (line 5). Alternatively, they can specify a group of collections to share an enumeration between (lines 7 and 9).

5.3 Evaluation

In this evaluation, I answer the following research questions:

RQ1 What is the impact of ADE on performance and memory usage, and are there differences across architectures?

RQ2 What proportion of sparse accesses are eliminated?

RQ3 What is the impact of our techniques to uncover and eliminate redundant translations?

RQ4 Can user directives enable further benefits?

RQ5 How does ADE compare to state-of-the-art collection implementations from industry?

5.3.1 Experimental Setup

I perform our evaluation on two machines:

Intel-x64: 176 core Intel Xeon Gold 6238L CPU at 2.10GHz with 2.8MiB 8-way L1d\$, 88MiB 16-way L2\$, 121MiB 11-way LL\$, and 376GiB of DDR4 at 3200MT/s.

AArch64: 128 core ARM Neoverse N1 at 3GHz with 8MiB 4-way L1d\$, 8MiB 4-way L2\$, 128MiB 8-way LL\$, and 256GiB of DDR4 at 3200MT/s.

Both machines run Ubuntu 22.04.1 LTS. Our compiler is built upon the MEMOIR compiler [88] using LLVM 18.1.8 [2], with abstractions from NOELLE [54].

I implement 15 of the 17 Lonestar 'Analytics' benchmarks [89] and freqmine from PARSEC [90], listed in Table 5.2. These benchmarks were selected for their use of high-level collection types. While they are used in the compiler community as benchmarks for parallel programs, their use of high-level collection types makes their sequential implementation interesting for data-oriented optimizations. These benchmarks are implemented in C++ using abstract MEMOIR data collection types instead of low-level implementations, representing code written by developers before heavy manual optimization. Figure 5.2 shows a breakdown of collection operations executed in each, and a hierarchical clustering of benchmarks based on the breakdown. I find that these benchmarks stress a diverse set of collection types and operations. Suite inputs are used where provided, otherwise an input from SNAP [91], [92] is used. All benchmarks are compiled with full LTO and -O3. Results shown are the median of 10 runs, with the geometric mean (GEO) reported in each plot. The baseline execution times are shown in Table 5.2 to contextualize reported speedup.

RQ1: What is the impact of ADE on performance and memory usage, and are there differences across architectures?

Table 5.2: List of benchmarks used to evaluate ADE, and their baseline execution time.

Abbr.	Benchmark	Execution Time (seconds)
PTA	LLVM Points-To Analysis	2877.58
FIM	Frequent Itemset Mining	152.56
MST	Minimum Spanning Tree	103.62
BP	Bipartion	125.17
CC	Connected Components	29.51
PP	Preflow-Push	585.84
CD	Community Detection	1481.67
PR	PageRank	166.60
BFS	Breadth-first Search	497.70
KC	k -Core	16.57
KT	k -Truss	7.46
TC	Triangle Counting	869.69
IS	Independent Set	78.45
MCBM	Max Card. Bipartite Matching	538.42
SSSP	Single-Source Shortest Path	860.84
BC	Betweenness Centrality	16.08

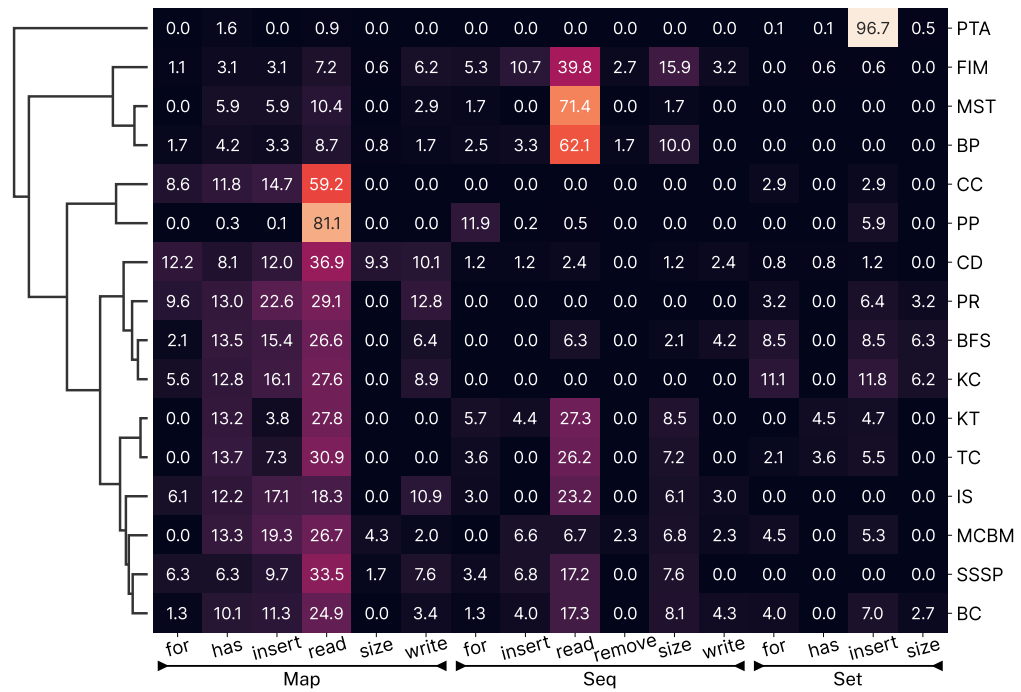


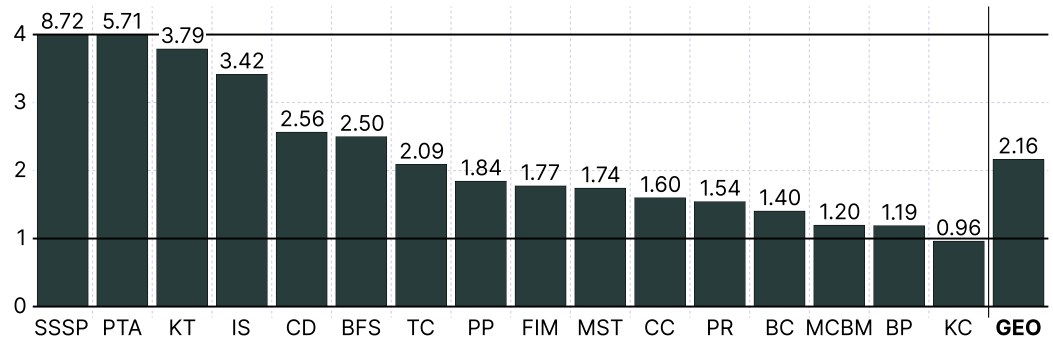
Figure 5.2: Hierarchical clustering of benchmarks based on the breakdown of dynamic collection operations executed.

I compare ADE against the MEMOIR compiler on Intel-x64. I present speedup for the whole-program (Figure 5.3a) and region of interest, where initialization is factored out (Figure 5.3b).

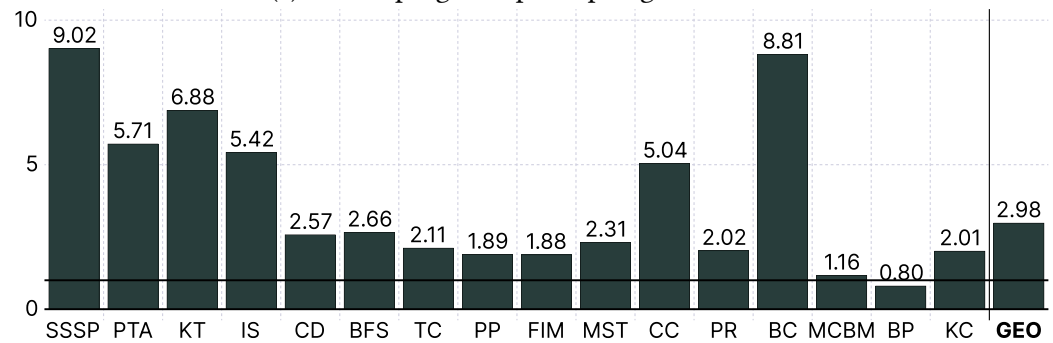
I find that ADE provides substantial whole-program speedups of $2.12\times$ on average, up to $8.72\times$, with a sole regression on KC of $0.94\times$. KC is a case where the cost of constructing the enumeration ($1.16\times$ initialization slowdown) is not amortized by the ROI speedup ($2.01\times$) since $> 90\%$ of execution time is initialization. Focusing solely on the region of interest, I see speedups of $2.98\times$ on average, up to $9.02\times$. These results show that ADE optimizations and specialized implementations overcome the overhead of enumeration construction and translation for significant performance gains.

I evaluate memory usage using maximum resident size and present our findings in Figure 5.3c. I find that ADE reduces maximum resident size by 5.6% on average, up to 50.7% . However, ADE sees no impact on the majority of benchmarks. I see one regression on FIM with a 27.3% increase. The regression is caused by our static benefit heuristic, which results in the compiler enumerating cold collections that are only used during verbose output. Since verbose output is disabled for the PARSEC input, the additional mappings are allocated and never used. While ADE does not consistently reduce memory usage, it achieves the aforementioned performance results without bloated memory usage.

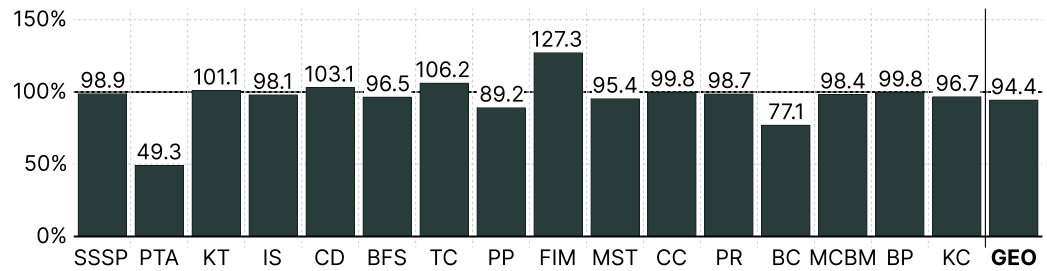
With recent shifts in server and consumer computing markets from x86-64 towards AArch64, it is important to examine the impact of compiler optimizations across both architectures. Figure 5.4 shows the performance results of ADE on AArch64. I found no significant difference in relative memory usage between Intel-x64 and AArch64. Bars are shaded green if the results are better on AArch64 than Intel-x64, and red if worse.



(a) Whole-program speedup, higher is better.

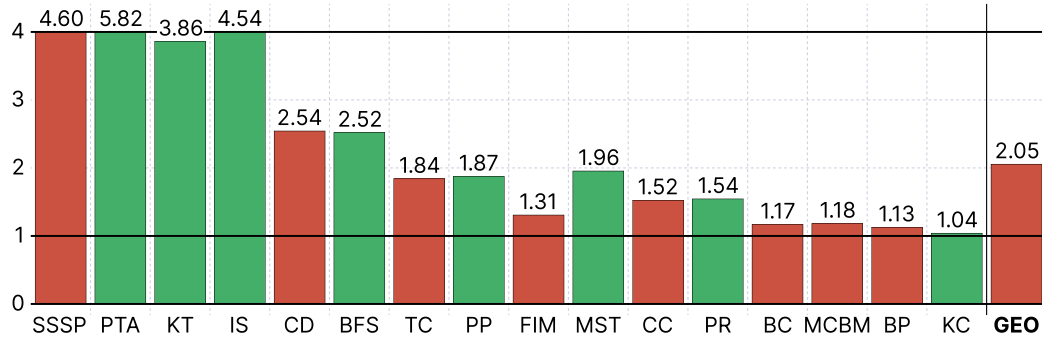


(b) ROI speedup, higher is better.

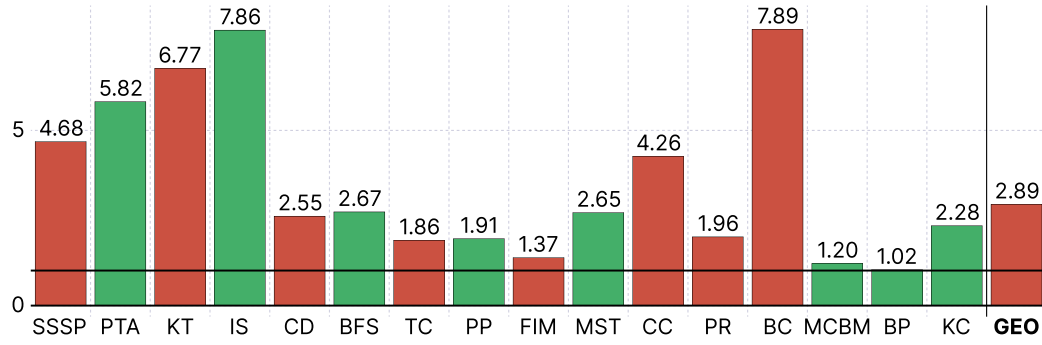


(c) Maximum resident set size, lower is better.

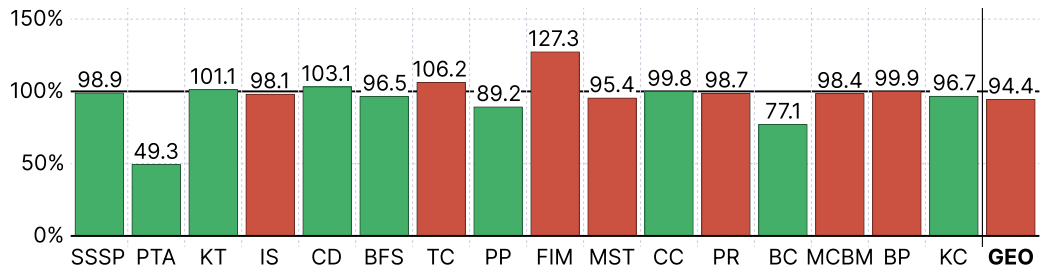
Figure 5.3: Performance and memory usage of ADE compared to MEMOIR running on Intel-x64.



(a) Whole-program speedup, higher is better.



(b) ROI speedup, higher is better.



(c) Memory usage, lower is better.

Figure 5.4: Performance and memory usage of ADE relative to MEMOIR on AArch64.

I find that ADE speedups are robust across architectures, with whole-program speedups of $2.03\times$ on average, up to $11.35\times$, and ROI speedups of $2.91\times$ on average, up to $7.89\times$. I find some benchmarks with significant regression on AArch64, such as SSSP, where whole-program speedup falls from $8.72\times$ to $4.60\times$. This is caused by operation-level performance differences across architectures (Table 5.4). Specifically, writes and insertions to a BitMap, hot operations for SSSP, see $1.56\times$ and $1.47\times$ slowdown on AArch64, respectively.

RQ2: What proportion of sparse accesses are eliminated?

In Section 5.1 I described the high-level goal of ADE: replacing sparse accesses with dense accesses. I evaluate the composition of these accesses in our benchmarks, shown in Table 5.3, and find that ADE effectively replaces sparse accesses with dense accesses. I find that ADE may increase the total number of dynamic accesses, however performance results suggest that this is a beneficial tradeoff.

To understand this tradeoff, I measure relative performance of different implementations across machines, shown in Table 5.4. I find that all operations, aside from set iteration, are significantly more performant on a Bit{Set, Map} compared to both Hash{Set, Map} and Swiss{Set, Map}.

RQ3: What is the impact of our techniques to uncover and eliminate redundant translations?

ADE employs a battery of optimizations to uncover and eliminate redundant translations. I perform an ablation study to understand the impact of each. The results are shown in Figures 5.5 and 5.6 relative to ADE with all optimizations.

First, I evaluate the impact of redundant translation elimination (RTE), as described in Section 5.2.2. The results of disabling RTE are seen in Figure 5.5a. I find that RTE is beneficial across the board, with an average $2.63\times$ slowdown when disabled, up to $16.7\times$. I also find that RTE has no impact on memory usage, which

Table 5.3: Sparse/dense access counts relative to MEMOIR.

<i>Bench</i>	MEMOIR		ADE		Difference		
	<i>Sparse</i>	<i>Dense</i>	<i>Sparse</i>	<i>Dense</i>	<i>Sparse</i>	<i>Dense</i>	<i>Total</i>
BC	48.8	51.2	40.1	110.6	-8.7	+59.4	+50.7
BFS	100.0	0.0	3.2	98.2	-96.8	+98.2	+1.4
BP	6.3	93.7	6.2	93.5	-0.1	-0.2	-0.3
CC	100.0	0.0	32.0	101.0	-68.0	+101.0	+33.0
CD	100.0	0.0	39.8	60.3	-60.2	+60.3	+0.0
FIM	33.0	67.0	5.6	102.2	-27.4	+35.2	+7.7
IS	64.9	35.1	5.7	117.0	-59.2	+81.9	+22.7
KC	100.0	0.0	151.0	82.1	+51.0	+82.1	+133.0
KT	53.0	47.0	5.3	102.2	-47.7	+55.2	+7.4
MCBM	74.1	25.9	63.7	115.6	-10.4	+89.7	+79.3
MST	5.0	95.0	2.7	100.2	-2.3	+5.2	+2.9
PP	99.3	0.7	8.7	100.2	-90.5	+99.4	+8.9
PR	100.0	0.0	24.2	95.7	-75.8	+95.7	+19.9
PTA	100.0	0.0	30.4	107.6	-69.5	+107.5	+38.0
SSSP	50.0	50.0	0.1	100.0	-50.0	+50.0	+0.1
TC	19.7	80.3	0.1	380.3	-19.6	+300.0	+280.4

Table 5.4: Per-operation speedup relative to Hash{Set, Map}.

Impl.		Read	Write	Insert	Remove	Iterate	Union
Intel-x64	BitSet	—	—	9.08	1.24	0.19	5817.38
	SparseBitSet	—	—	1.54	1.07	0.27	3700.50
	SwissSet	—	—	1.61	0.40	0.27	1.71
	FlatSet	—	—	0.19	0.10	5.59	25.31
	BitMap	10.63	15.94	13.10	1.32	2.65	—
	SwissMap	0.69	1.46	2.58	0.41	3.65	—
AArch64	BitSet	—	—	12.53	2.63	0.22	6944.48
	SparseBitSet	—	—	2.81	2.21	0.29	4702.13
	SwissSet	—	—	1.46	0.52	0.28	3.28
	FlatSet	—	—	0.28	0.22	3.15	50.37
	BitMap	18.65	10.20	8.91	2.60	6.41	—
	SwissMap	0.64	0.65	1.18	0.51	7.16	—

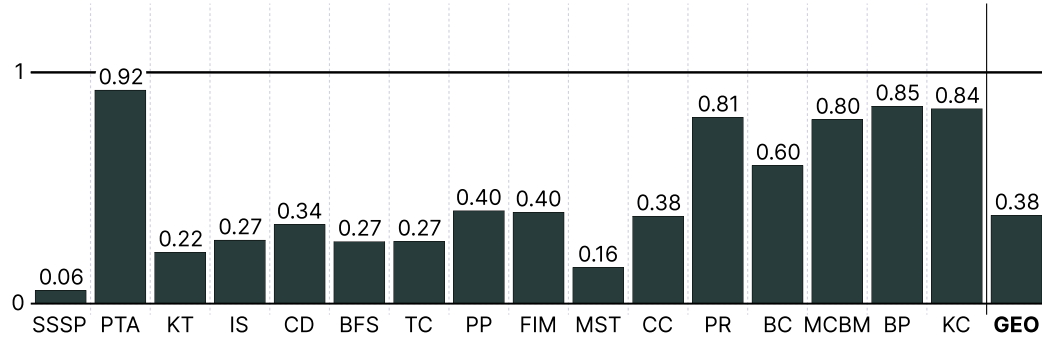
is expected because it solely removes redundant access operations.

Next, I evaluate propagation (Section 5.2.4), shown in Figure 5.5b. These results show the primary benefit of propagation: uncovering latent redundancy, where elements are used to primarily transfer values from one enumerated collection to another.. This is seen in the correlation between RTE and propagation ablation speedups. In cases, such as SSSP, this is especially prevalent as I see approximately equivalent slowdowns when either RTE or propagation are disabled. This indicates that, in such cases, the compiler is able to uncover little to no redundancy without propagation. I find that propagation has no impact on memory usage for the majority of benchmarks, with the exception of FIM where I see memory usage balloon to 1694.3% This is due to propagation being necessary to enable sharing, without which I introduce many enumeration mappings. This is supported by the memory usage seen with sharing disabled, which similarly grows to 1696.1%

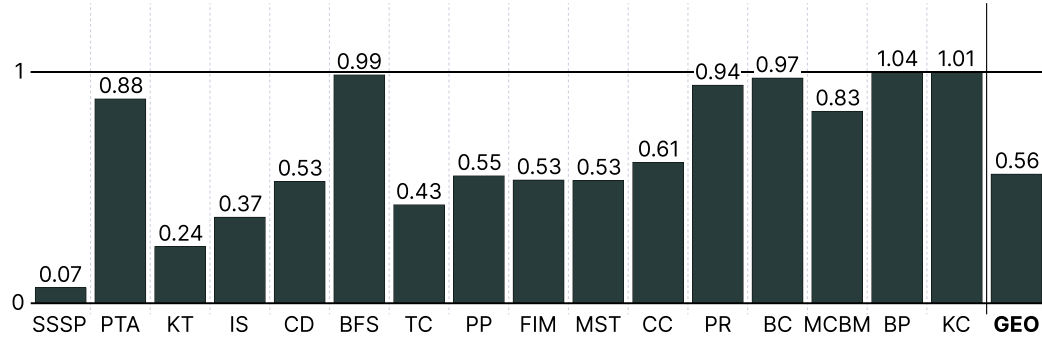
Finally, I evaluate sharing (Section 5.2.3), shown in Figures 5.5c and 5.6. Of note, no sharing also entails no propagation, as the benefit computation will never introduce a propagator unless it can be shared with an enumerated collection. Similarly to propagation, sharing uncovers latent redundancy, which is illustrated by a trend towards the slowdowns seen with RTE disabled. Unlike the prior optimizations, sharing has a significant impact on memory usage, rising by an average of 20% when disabled. However, there is one outlier in PTA, which achieves significant speedup and memory usage reduction with sharing disabled. This is caused by our benefit heuristic being overly eager to share an enumeration between collections, leading to `Bit{Set,Map}` that are sparse over the shared enumeration. I use this outlier as a case study for our user-guided optimization directives in RQ4.

RQ4: Can user directives enable further benefits?

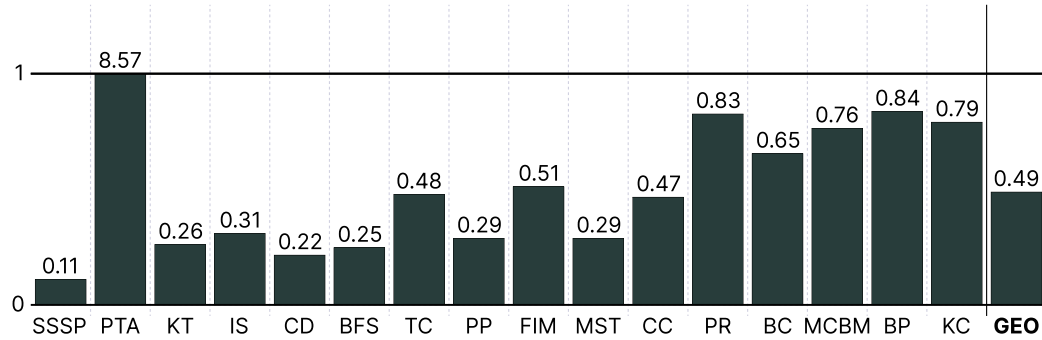
In RQ3, I found that ADE heuristics were too optimistic for sharing enumerations



(a) ADE with redundant translation elimination disabled



(b) ADE with propagation disabled



(c) ADE with sharing disabled

Figure 5.5: Ablation study, whole-program speedup with optimization techniques disabled on Intel-x64 relative to ADE with all optimizations enabled.

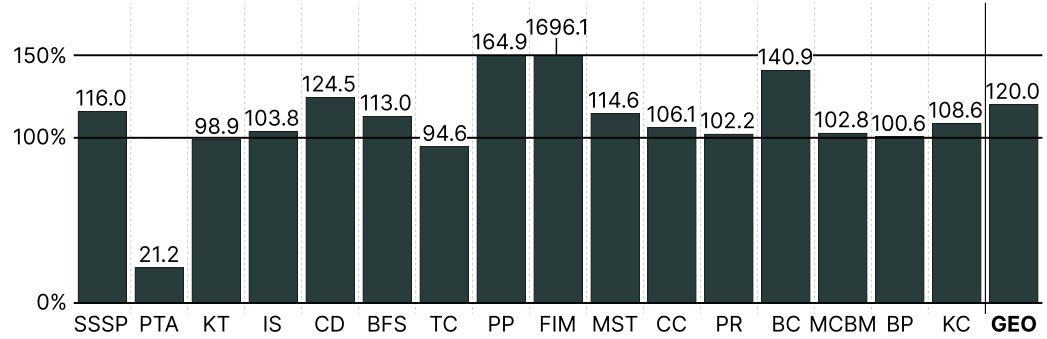


Figure 5.6: Ablation study, memory usage with sharing disabled on Intel-x64 relative to ADE with all optimizations enabled. RTE had no impact on memory usage, while propagation only impacted FIM with similar results to sharing.

in PTA. I use PTA as a performance engineering case study, using the directives from Section 5.2.8.

I first bisect the problem by disabling sharing for each collection in the program using the `noshare` directive. Doing so, I find that the regression was caused by sharing with the points-to set (`Map<ptr, Set<ptr>> %pts`), particularly the inner sets `%pts[*]`. By disabling sharing for the inner sets, guided ADE achieves a speedup of $78.1\times$ and reduce memory usage by 71.0% relative to MEMOIR ($13.7\times$ speedup and 41.1% memory usage reduction relative to untuned ADE). The root cause of this speedup is that the inner sets range over all objects, while the original enumeration ranges over all pointers. There are approximately 1.8×10^3 allocations and 2×10^7 pointers in the `sqlite3` bytecode input, resulting in the inner bitsets using only 0.009% of their bits. This led to a large performance penalty for aggregate operations, such as iterating over the bitset or computing the number of set bits.

Seeing these results, I investigated if data enumeration is even beneficial for the inner sets with the `noenumerate` directive. This resulted in a speedup of only $1.12\times$ and a memory usage reduction of 25.9% relative to MEMOIR.

Finally, I investigate alternative implementations for the inner set using the

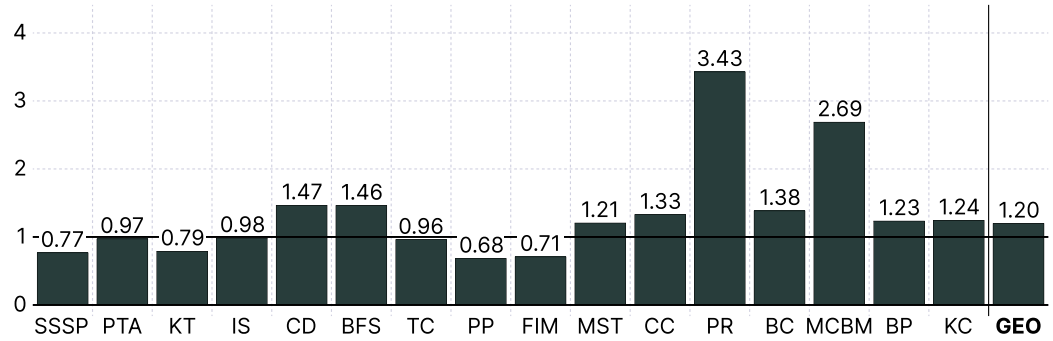
`select` directive. I first selected a `SparseBitSet` to compact the memory layout by eliminating unused bits. This resulted in a $1.57\times$ speedup and 67.1% reduction in memory usage compared to untuned `BitSet`. Next, I selected a `FlatSet` so that only items in the set are stored, with linear set union (a hot operation). This resulted in a $2.73\times$ speedup and 20.4% reduction in memory usage compared to `BitSet`. While these results showed a good improvement, they fall short of the $78.1\times$ speedup and 71.0% reduced memory usage achieved with `noshare`.

RQ5: How does ADE compare to state-of-the-art collection implementations from industry?

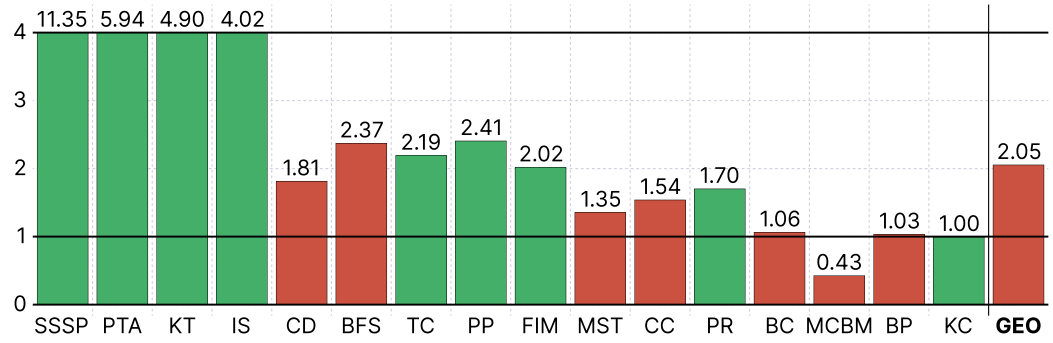
While the C++ standard template library is commonly used by developers, it has known performance limitations. Because of this, developers make use of third party libraries, such as Abseil [93]. I compare ADE against Abseil's swiss table, a highly optimized implementation of sets and maps. I present our findings in Figures 5.7 and 5.8 for Intel-x64. I did not see many significant differences on AArch64, I will discuss the differences found later in this section.

First, I directly compare `Swiss{Set,Map}` against `Hash{Set,Map}` by setting the default implementation of the MEMOIR compiler. The results are shown in Figures 5.7 and 5.8. On average, I find that `Swiss{Set,Map}` is more performant and uses less memory. However, there are major performance regressions on SSSP, KT, PP and FIM. These regressions are in stark contrast to large gains on PR and MCBM, with speedups of $3.43\times$ and $2.69\times$, respectively.

Using the direct comparison as a backdrop, I compare ADE against MEMOIR, where both use `Swiss{Set,Map}` as the default implementation. Performance results are shown in Figure 5.7b. Most benchmarks have similar speedup results as in RQ1. This is with the sole exception of MCBM, where `Swiss{Set,Map}` provide



(a) MEMOIR with Swiss{Set, Map} over Hash{Set, Map}.



(b) ADE over MEMOIR with Swiss{Set, Map}.

Figure 5.7: Whole-program speedup of MEMOIR and ADE with and against Swiss{Set, Map} on Intel-x64, higher is better.

a better solution than ADE in this use case, but tuning with directives could help.

Looking at memory usage in Figure 5.8b, ADE provides a significant improvement over Swiss{Set, Map} in PTA and TC, where memory usage is reduced by 60.0% and 70.4%, respectively, with a sole regression on FIM of 27.6%.

On AArch64, there were two deviations from the Intel-x64: SSSP saw performance regressions using Swiss{Set, Map}, while IS saw performance improvements using Swiss{Set, Map}.

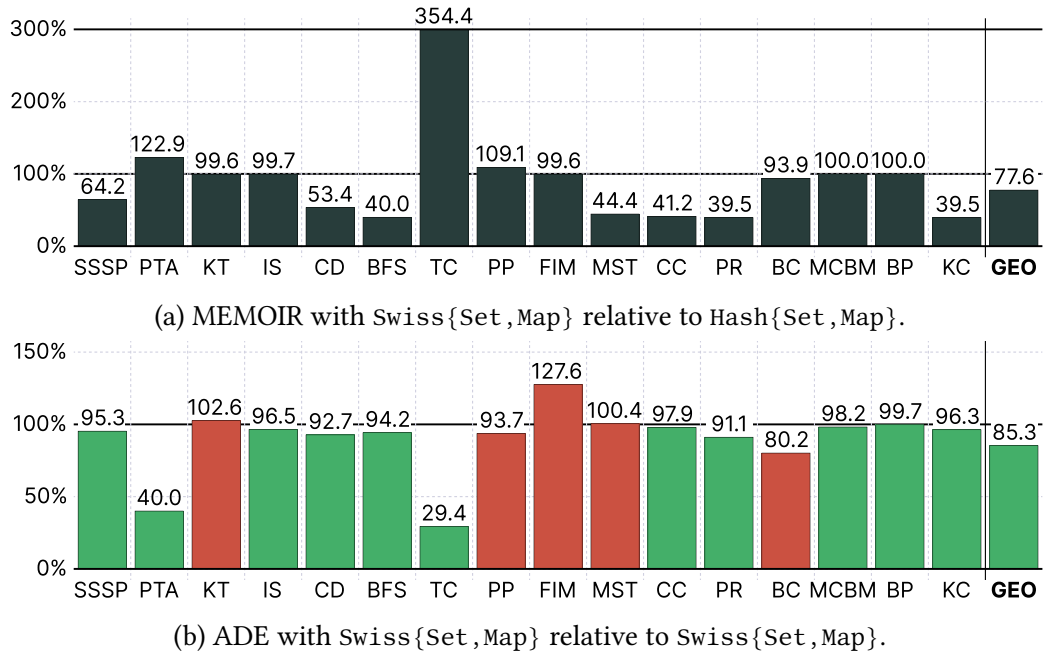


Figure 5.8: Maximum resident set size comparison with Swiss{Set, Map} on Intel-x64, lower is better.

5.4 Discussion

In this chapter, I have shown that compilers can automate data enumeration to achieve up to $78.1\times$ speedup and 71.0% reduced memory usage without manual effort. The key insight is that manufacturing a data property—contiguous integer identifiers—unlocks specialized implementations that would otherwise be inapplicable. This demonstrates that program transformation and collection selection can be co-designed to achieve more than the sum of their parts. While this work focuses on data enumeration for sets and maps, I see clear opportunities for extending these ideas to additional properties and collection types through integration with prior work.

ADE also demonstrates the power of MEMOIR [88], which make the necessary analysis and transformation possible. I believe that such infrastructure is essential

for future work in this direction, and that these results establish a new envelope for collection-centric compiler optimizations, paving the way for compilers that treat collections as first-class optimization targets.

5.4.1 Prior Works

ADE differs from prior approaches to collection selection by *enabling* specialized collections through program transformation, while prior work only *explores* the existing selection space.

Runtime selection techniques fully embrace the dynamicity of collection selection, using profiling to select the implementation. CoCo [71] uses a complexity-guided runtime selection solution for Java using ‘collection combos’, a set of implementations with a shared interface. One implementation in the combo is active at a time, while inactive collections are updated to hold abstract elements that can be used to compute the concrete value when the selection changes. Seeing the overhead of instance-level adaptation, CollectionSwitch [65] coarsens runtime selection to the allocation site. By combining runtime selection with just-in-time transformation to introduce desired properties, specialized implementations could be enabled. Additionally, a runtime system could inspect collections for such desired properties.

Search-based selection relies on runtime profiles to select implementations for hot operations. `perf1int` [63] first used this strategy to identify inefficient collection usage in C++, recommending source code changes to fix them, e.g., converting `set` to `unordered_set`. Brainy [64] extended this with a machine-learned model that is more effective across inputs. Similar ideas have since been explored in both

Java and C++ [67], [68], [69] and could be use ADE as a selection target. Collection Skeletons [66] takes a different approach, searching for an implementation that satisfies the user-specified properties of the collection, e.g., uniqueness, which could be enhanced with manufactured properties.

Data structure synthesis takes a data structure description and synthesizes an implementation. Many approaches [94], [95] use a relational model of the program’s data for synthesis, with extensions to support concurrency [96]. CRES [72] extends these techniques to use static analysis instead of runtime models, eliding profiling cost. Furthermore, Cozy [70] enables users to express ad-hoc properties via query descriptions, such as the directedness or cyclicity of a graph, which are leveraged during synthesis. To build upon this, MEMOIR collections can be described to these systems as relations, and specialized implementations can be synthesized using a toolbox of transformations that manufacture properties.

Automatic Memory Optimizations Performance engineers employ a variety of memory optimization techniques, from bespoke pointer representations [97] to hand-rolled garbage collection [98]. However, most of these techniques demand considerable manual effort with little to no automation. To reduce this burden, prior work has shown that compilers can automate several memory optimizations, such as pool allocation [57], pointer compression [99], array interleaving [56], memory defragmentation [98] and data structure splicing [58], [88], [100]. This chapter adds data enumeration to this list.

5.4.2 Limitations

ADE relies on a static benefit heuristic to decide when enumeration is profitable. As the PTA case study demonstrates, this heuristic can be overly optimistic when collections share an enumeration over incompatible key spaces, leading to sparse array-like structures that hurt rather than help. User directives mitigate this, but require manual intervention.

ADE is also restricted to sequential programs. Extending it to parallel programs would require careful handling of shared enumerations across threads to avoid data races, which I leave to future work.

Finally, the generality of ADE remains an open question. The benchmarks evaluated here represent programs that make heavy use of associative relations, e.g., graph analytics, program analysis, data mining, and network analysis. Whether ADE provides similar benefits across a broader class of programs is not yet known.

CHAPTER 6

Representing References to SSA Data Collections

Data collections—e.g., tensor, list, set, map—are a commonplace abstraction to organize data, simplifying development and maintenance. There has been growing interest in representing data collections in the compiler using static single assignment (SSA) form [88], [101], [102], [103] to enable compiler analysis and optimization. To enable local reasoning about the state of elements in the collection, these intermediate representations (IRs) use value semantics, where update operations yield a fresh collection with the updated elements and the original collection is left unchanged. While this enables sparse dataflow analysis and simplifies the applicability guards for compiler optimizations—since reaching definitions and dominance are made explicit in strict SSA form—it introduces a tension when representing common programming idioms: *element references violate SSA*.

Obtaining references to elements within a collection is a common programming idiom. References allow programmers to reuse code for objects regardless of the data structure containing it. For example, a function that increments an integer held by a reference could be applied to any integer stored within a scalar, field, or element of a collection.

Iterators extend references with the ability to traverse the container relative to

the referenced element. This enables further code reuse for functions that operate on ranges of data, such as `std::sort` and the `std::ranges` library in C++. Beyond code reuse, some collections rely solely on iterators to access them, such as a bag or the recently added `std::hive`.

References are both necessary and ubiquitous. However, they exist in opposition to SSA with value semantics, where collections and their elements can be accessed via a single reference, the collection value. Consider the following program, which pushes an element to the front of a list, and then reads through a previously obtained reference:

```

1 std::list<int> list = {10, 20, 30};
2 int &ref = list.front(); // element 10
3 list.push_front(7);      // {7, 10, 20, 30}
4 std::cout << ref;        // prints 10

```

Expressing the high-level insert operation in this program is straightforward with MEMOIR. However, there is no way to express `ref`, which identifies element 10 in `list0` and remains meaningful in the context of `list1`. Currently, MEMOIR must either disallow references entirely, or prematurely lower them to index-based accesses. Either way, the abstraction that references provide is lost. This chapter addresses this gap.

I introduce *bookmarks*, an immutable value that identifies an element within a collection and remains semantically meaningful across collection updates. With bookmarks, the above program is expressed as:

```

1 let b = mark list0[0] in # bookmark element 10
2 let list1 = insert 0 into list0 in
3 let list2 = write 7 to list0[0] in
4 let v = read b of list2 in ... # yields 10

```

The bookmark *b* identifies the element at index 0 in *list*₀. When read in the context of *list*₂, it identifies the element at index 1—following the **insert** at index 0—and yields its value 10.

The key to bookmarks is the *lineage function*, which tracks element positions between derived collection states. For example, the lineage function of **insert** 0 specifies that all elements shift to the right by one in the output. Lineage functions enable the compiler to reason about bookmarks statically, without assuming anything about the underlying memory layout.

By requiring that all operations using bookmarks supply the collection being accessed as an explicit operand, bookmarks are contextualized against any derived collection version using the lineage function—mapping the referenced element from the original collection to the accessed collection. This gives bookmarks meaningful semantics in the face of operations that modify the elements within a collection. This can be seen in the example, where it can be statically determined that the bookmark access **read** *b* **of** *list*₂ refers to the element at index 1 in *list*₂, even though it was defined on *list*₀.

Furthermore, this representation maintains SSA form by redefining the base collection whenever it is updated via a bookmark. This provides the foundation for a sparse static analysis for memory safety bugs related to reference invalidation.

These same bookmark semantics extend naturally to iterators, which generalize bookmarks with traversal operations. Applying bookmark semantics to itera-

tors enables MEMOIR to represent iterators over multiple versions of collections and element updates via iterators.

This chapter is organized as follows. Section 6.1 provides an informal description of bookmark semantics by example. Section 6.2 formally defines bookmark semantics as an extension of MEMOIR. Section 6.3 details how bookmark semantics are extended with traversal semantics to represent iterators. Section 6.5 proposes a safety analysis for bookmarks, statically identifying cases of undefined bookmark behavior. Section 6.4 specifies how bookmarks are safely lowered to imperative semantics. Section 6.6 illustrates how the expressiveness of bookmarks enables a drop-in replacement C++ library that uses logical collections, instead of raw memory, under the hood. Section 6.7 closes with a discussion about future work enabled by bookmark semantics and their limitations.

6.1 Bookmark Semantics, Informally

A *collection* is a finite container of *elements*. In SSA, collections have value semantics, where an update operation yields a new collection instead of modifying the original collection in-place. For example, given a sequence $x = [10, 20, 30]$ inserting the value 77 at index 1 yields a fresh collection $y = [10, 77, 20, 30]$. Following this operation, both x and y are valid, immutable values in the program.

An *element* is a value owned by a collection. Like collections, elements are immutable, i.e., they cannot be modified in place. An update to a collection that changes an element's value yields a new collection owning a new element. The original collection and its elements remain unchanged.

A *bookmark* is an immutable value that relates elements across collection values. A bookmark `mark x[1]` identifies the element owned by x at index 1. To

access a collection through a bookmark, the program must supply a collection as context. Consider the following code:

```

1 let x = [10, 20, 30] in # sequence of elements
2 let r = mark x[1] in # refers to the element 20 in x
3 let v = read r of x in # origin = context, yields 20

```

Here *r* is used in the context of *x*, the same collection it was created against, so it trivially refers to *x*[1]. The more interesting case arises when the context and origin differ:

```

1 let x = [10, 20, 30] in # sequence of elements
2 let r = mark x[1] in # refers to the 1th element in x
3 let y = remap x with  $\delta$  in # y is derived from x
4 let v = read r of y in # origin != context, yields ???

```

Here *r* was created against *x*, but is read in the context of *y*. The *lineage function* δ is employed to resolve such cases where a bookmark is used in a different context from the one it was created in. Each **remap** operation carries a lineage function that maps elements of its input collection to elements of its output collection. To resolve *r* in the context of *y*, the semantics apply δ to the origin index, yielding the index of the corresponding element in *y*. If $\delta(1) = 2$, then *r* refers to *y*[2] and **read r of y** yields 20.

When a bookmark is evaluated against a collection separated by multiple updates, the semantics compose the lineage functions of every **remap** along the chain. Consider:

```

1  let x = [10, 20, 30] in
2  let r = mark x[1] in # r refers to x[1]
3  let y = remap x with  $\delta_1$  in # r refers to x[ $\delta_1(1)$ ]
4  let z = remap x with  $\delta_2$  in # r refers to x[ $\delta_2(\delta_1(1))$ ]
5  let v = read r of z in # lineage composes, yields 20

```

To resolve r in the context of z , the semantics first apply δ_1 to the origin index, then apply δ_2 to the result. The composition $\delta_2 \circ \delta_1$ gives the final index in z .

If any update along the derivation chain destroys the bookmarked element, the lineage function at that step has no valid output and the bookmark is considered *dead*. Accessing a dead bookmark is undefined. For example:

```

1  let x = [10, 20, 30] in
2  let r = mark x[1] in # r refers to x[1]
3  let y = remap x with erase(1) in # y = [10, 30]
4  let v = read r of y in # r is dead  $\implies$  undefined

```

The lineage function `erase(1)` has no valid output for index 1. The element was destroyed, not relocated. Any subsequent access through r is undefined, regardless of further updates to the collection.

6.1.1 Lineage Functions by Example

I quickly introduced lineage functions in Section 4.1.3. In this section, I provide a more detailed definition by defining the set of lineage functions for sequences as an example. Each lineage function is parameterized by one or more operands, evaluated against the variable environment E via `val` before application.

The lineage function for $\text{insert}(k)$ models the structural effect of inserting a new element at index k . Elements at indices $\geq k$ are shifted up by one, and all other elements remain in place:

$$\text{insert}(k)(x) = \begin{cases} x + 1 & \text{if } x \geq k \\ x & \text{otherwise} \end{cases}$$

The lineage function for $\text{erase}(k)$ models the structural effect of removing the element at index k . Elements at indices $> k$ are shifted down by one, the element at k is destroyed, and all other elements remain in place:

$$\text{erase}(k)(x) = \begin{cases} \perp & \text{if } x = k \\ x - 1 & \text{if } x > k \\ x & \text{otherwise} \end{cases}$$

6.2 Bookmark Semantics, Formally

Bookmarks extend the MEMOIR syntax and type system. Bookmarks are of type $\text{Mark}<T>$, pointing to an element of type T within a collection. In this chapter, I distill the collection operations in MEMOIR down to **read**, **write** and **remap**. The **remap** operation eliminates redundancy in the formal definition when describing operations that update the index space of the collection.

For simplicity, I only define bookmark semantics with the only scalar type being the set of natural numbers $\text{Nat} \in \mathbb{N}$. Because of this, I use shorthand for all types $\text{Seq} = \text{Seq}<\text{Nat}>$, $\text{Map} = \text{Map}<\text{Nat}, \text{Nat}>$, $\text{Set} = \text{Set}<\text{Nat}>$, $\text{Mark} = \text{Mark}<\text{Nat}>$ and $\text{Iter} = \text{Iter}<\text{Nat}, \text{Nat}>$. The semantic domain is defined in Fig-

$x \in \text{Variables}, l \in \text{Labels}, n \in \mathbb{N}$		
$i ::=$	mark $x[v]$	<i>obtain bookmark</i>
	remap x with op	<i>remap elements</i>
$\tau ::=$	$\dots$ Mark $\langle \tau \rangle$	<i>types</i>

Figure 6.1: MEMOIR syntax extensions to support bookmarks.

ure 6.2.

Informally, bookmark semantics operate on three kinds of values w : natural numbers n , memory locations ℓ , and bookmarks r . The index domain ι extends natural numbers with two sentinel values: *end*, representing the one-past-the-end position, and *poison*, representing an invalid index¹. Rules that pattern match on (ℓ, n) implicitly require $n \in \mathbb{N}$, so rules do not apply when the index is either of these sentinel values, which is undefined behavior. References are a tuple (ℓ, ι) of the base memory location of the collection being bookmarked ℓ and the index into that collection ι . The variable environment E maps variables to values. The store S maps locations to collections—a mapping from indices to values. The lineage environment Δ maps a pair of locations (ℓ, ℓ') to a lineage function δ , representing the translation of element indices from origin location ℓ to context location ℓ' .

I define $\text{extend}(\Delta, \ell, \ell', \delta_i)$ to update the lineage environment Δ by composing δ onto all existing entries whose context is ℓ , mapping them to the new context ℓ' . The identity entry $(\ell', \ell') \mapsto \text{id}$ is also added to Δ . When $\ell = \ell'$, existing entries whose context is ℓ are composed in-place.

Instruction evaluation $(\rightarrow)$ is defined in Figure 6.3 as a relation $\langle E; S; i \rangle \rightarrow \langle S'; v \rangle$, which yields an updated store, updated lineage environment, and a resulting value. I write *fresh* ℓ to mean that ℓ is a fresh location not present in S . I define $\text{val}(v, E)$ to evaluate an operand v in the environment E . I define $\delta \cdot c$ to produce a fresh

¹These are used by iterator operations, discussed in Section 6.3.

$x \in Var$		variable
$n \in \mathbb{N}$		scalar value
$\ell \in Loc$		location
$\iota \in Idx ::= n \mid \text{end} \mid \perp$		index
$r \in Ref ::= (\ell, \iota)$		bookmark value
$w \in Val ::= n \mid \ell \mid r$		value
$c \in Col ::= \mathbb{N} \rightarrow Val$		collection
$\delta ::= \mathbb{N} \rightarrow \mathbb{N}$		lineage function
$E = Var \rightarrow Val$		variable environment
$S = Loc \rightarrow Col$		store
$\Delta = (Loc \times Loc) \rightarrow \delta$		lineage environment

Figure 6.2: Semantic domain of CORE.

collection by applying δ to remap the indices of all existing elements in c . When evaluating a **remap** instruction, the operation op is evaluated to obtain a concrete lineage function δ , where E , Δ and ℓ provide the context necessary to resolve operands to concrete indices.

I define a standard small-step operational semantics for CORE in Figure 6.4. For let-bindings, this evaluates the instruction, updates the environment with the new binding, and then steps to the continuation with the updated environment and store. For jump and **if0** terminators, this determines the successor block patches the environment to use the basic block parameters, and then continues execution in the block expression. **return** terminators mark the end of evaluation.

6.3 Extending Bookmarks to Represent Iterators

Iterators extend bookmarks with the ability to traverse a collection. While a bookmark identifies a single element, an iterator additionally supports movement to adjacent elements, enabling idioms like sequential scans, in-place filtering, and range-based algorithms. Like bookmarks, iterators are immutable SSA values, i.e., an iterator identifies an element at the point it was created and remains mean-

$$\begin{array}{c}
\text{fresh } \ell \quad S' = S[\ell \mapsto \emptyset] \quad \Delta' = \Delta[(\ell, \ell) \mapsto \text{id}] \\
\hline
\langle E; S; \Delta; \boxed{\text{alloc}} \rangle \rightarrow \langle E; S'; \Delta'; \ell \rangle \quad \text{E-ALLOC} \\
\\
\text{fresh } \ell \quad c = [i \mapsto \text{val}(v_i, E)] \\
S' = S[\ell \mapsto c] \quad \Delta' = \Delta[(\ell, \ell) \mapsto \text{id}] \\
\hline
\langle E; S; \Delta; \boxed{[v_i]} \rangle \rightarrow \langle E; S'; \Delta'; \ell \rangle \quad \text{E-INIT} \\
\\
n = \text{val}(E, v) \quad E(x) = \ell \\
\hline
\langle E; S; \Delta; \boxed{\text{mark } x[v]} \rangle \rightarrow \langle E; S; \Delta; (\ell, n) \rangle \quad \text{E-MARK} \\
\\
E(x_r) = (\ell_r, n_r) \quad E(x_c) = \ell \quad \delta = \Delta(\ell_r, \ell) \\
n'_r = \delta(n_r) \quad S(\ell) = c \quad w = c(n'_r) \\
\hline
\langle E; S; \Delta; \boxed{\text{read } x_r \text{ of } x_c} \rangle \rightarrow \langle E; S; \Delta; w \rangle \quad \text{E-READ} \\
\\
E(x_r) = (\ell_r, n_r) \quad E(x_c) = \ell \quad \delta = \Delta(\ell_r, \ell) \\
n'_r = \delta(n_r) \quad w = \text{val}(E, v) \quad S(\ell) = c \\
c' = c[n'_r \mapsto w] \quad \text{fresh } \ell' \quad S' = S[\ell' \mapsto c'] \\
\hline
\langle E; S; \Delta; \boxed{\text{write } v \text{ to } x_r \text{ of } x_c} \rangle \rightarrow \langle E; S'; \Delta; \ell' \rangle \quad \text{E-WRITE} \\
\\
E(x) = \ell \quad S(\ell) = c \quad \text{fresh } \ell' \\
\langle E; \Delta; \ell; \text{op} \rangle \rightarrow \langle \delta \rangle \quad c' = \delta \cdot c \quad S' = S[\ell' \mapsto c'] \\
\Delta' = \text{extend}(\Delta, \ell, \ell', \delta) \\
\hline
\langle E; S; \Delta; \boxed{\text{remap } x \text{ with op}} \rangle \rightarrow \langle E; S'; \Delta'; \ell' \rangle \quad \text{E-REMAP}
\end{array}$$

Figure 6.3: Instruction Evaluation Rules of CORE.

$$\begin{array}{c}
\frac{\langle E; S; i \rangle \rightarrow \langle S'; \bar{v} \rangle}{\langle E; S; \text{let } \bar{x} = i \text{ in } e \rangle \rightarrow \langle E[\bar{x} \mapsto \bar{v}]; S'; e \rangle} \text{E-LET} \\
\\
\frac{p(l) = \text{block } l(\bar{x}_p) e}{\langle E; S; \text{jump } l(\bar{x}_a) \rangle \rightarrow \langle E[\bar{x}_p / \bar{x}_a]; S; e \rangle} \text{E-JMP} \\
\\
\frac{E(x) = 0 \quad p(l_0) = \text{block } l_0(\bar{x}_p : \tau_p) e}{\langle E; S; \text{if0 } x \text{ } l_0(\bar{x}_0) \text{ } l_1(\bar{x}_1) \rangle \rightarrow \langle E[\bar{x}_p / \bar{x}_a]; S; e \rangle} \text{E-IF0-THEN} \\
\\
\frac{E(x) \neq 0 \quad p(l_0) = \text{block } l_1(\bar{x}_p) e}{\langle E; S; \text{if0 } x \text{ } l_0(\bar{x}_0) \text{ } l_1(\bar{x}_1) \rangle \rightarrow \langle E[\bar{x}_p / \bar{x}_a]; S; e \rangle} \text{E-IF0-ELSE}
\end{array}$$

Figure 6.4: Small-step operational semantics of SSA.

ingful across collection updates via the lineage environment. I define the syntax extension for iterators in Figure 6.5.

$$\begin{array}{ll}
i ::= \dots \mid \text{begin of } x \mid \text{end of } x & \text{obtain iterator} \\
\mid \text{next } x \text{ of } x \mid \text{prev } x \text{ of } x & \text{traversal} \\
\mid \text{dist } x \text{ to } x \text{ of } x & \text{distance} \\
\tau ::= \dots \mid \text{Iter} & \text{iterator type}
\end{array}$$

Figure 6.5: Syntax extensions for SSA to support iterators.

6.3.1 Iterator Operations

An iterator is obtained from a collection using **begin** and **end**, which yield iterators to the first element and one-past-the-end of the collection, respectively. Given an iterator, **next** and **prev** yield fresh iterators to the adjacent elements, and **dist** computes the distance between two iterators as a natural number. Like **mark**,

all traversal operations require a collection as context. This collection determines which version of the sequence the iterator is being evaluated against.

6.3.2 Iterators as Bookmarks

Fundamentally, an iterator is a bookmark. An iterator value is a tuple (ℓ, n) of an origin location and an index, identical in structure to a bookmark. The distinction between iterators and bookmarks is not in their representation but in their use. Iterators support traversal operations that bookmarks do not, and iterator arguments to **remap** operations yield a result iterator following the update.

By unifying the two, the lineage machinery developed for bookmarks applies directly to iterators. The validity of an iterator at a use site is determined by exactly the same definedness condition as a bookmark, i.e., whether $\Delta(\ell, \ell')(n)$ is defined at that point. The validity analysis, described later in Section 6.5.2, therefore handles iterators and bookmarks uniformly, with no special cases required.

6.3.3 Updates using Iterators

When a collection is updated via a **remap** operation, the behavior of existing iterators is governed by the operation's lineage function — the same function that governs bookmarks. An iterator that survives the update has its index remapped by the lineage function; an iterator whose element is removed becomes dead and any subsequent use is undefined.

When a **remap** operation takes an iterator as an argument, it will yield both a fresh collection and a fresh iterator to use following the update. For example, erasing the element at an iterator position yields a fresh iterator to the element that now occupies that position:

```
1 let list1, it1 = remap list0 with erase(it0) in  
2 let v = read it1 of list1 in
```

Here it_0 identifies the element to erase, and it_1 is the fresh iterator pointing to the element that took its place in $list_1$.

The result iterator yielded by a **remap** operation is specified by the operation's specification using a small language for data structure developers to describe how their operations affect element positions, physical stability, and iterator results. I discuss the specification language in Section 6.4.2.

For the purposes of this work, I define iterator traversal concretely over natural number indices. **next** and **prev** traverse to the next or previous index present in the collection, respectively, and **dist** computes the difference between two indices. In practice, a collection implementation may define its own traversal order via the implementation specification (Section 6.4.2), wrapping these operations to reflect the underlying data structure's notion of adjacency. The formal definition of iterator semantics can be found in Figure 6.6.

$$\begin{array}{c}
\frac{E(x) = \ell \quad c = S(\ell) \quad \iota = \text{begin}(c)}{\langle E; S; \Delta; \boxed{\text{begin of } x} \rangle \rightarrow \langle E; S; \Delta; (\ell, \iota) \rangle} \text{E-BEGIN} \\
\\
\frac{E(x) = \ell \quad c = S(\ell)}{\langle E; S; \Delta; \boxed{\text{end of } x} \rangle \rightarrow \langle E; S; \Delta; (\ell, \text{end}) \rangle} \text{E-END} \\
\\
\frac{E(x_i) = (\ell_i, \iota) \quad E(x_c) = \ell \quad \delta = \Delta(\ell_i, \ell) \quad \iota' = \delta(\iota) \quad \iota' \in \mathbb{N} \quad \iota'' = \text{next}(S(\ell), \delta(\iota))}{\langle E; S; \Delta; \boxed{\text{next } x_i \text{ of } x_c} \rangle \rightarrow \langle E; S; \Delta; (\ell, \iota'') \rangle} \text{E-NEXT} \\
\\
\frac{E(x_i) = (\ell_i, \iota) \quad E(x_c) = \ell \quad \delta = \Delta(\ell_i, \ell) \quad \iota' = \delta(\iota) \quad \iota' \in \mathbb{N} \quad \iota'' = \text{prev}(S(\ell), \iota')}{\langle E; S; \Delta; \boxed{\text{prev } x_i \text{ of } x_c} \rangle \rightarrow \langle E; S; \Delta; (\ell, \iota'') \rangle} \text{E-PREV} \\
\\
\frac{E(x_c) = \ell \quad E(x_i) = (\ell_i, \iota_i) \quad E(x_j) = (\ell_j, \iota_j) \quad \delta_i = \Delta(\ell_i, \ell) \quad \delta_j = \Delta(\ell_j, \ell) \quad \iota'_i = \delta_i(\iota_i) \quad \iota'_j = \delta_j(\iota_j) \quad \iota'_i \neq \perp \quad \iota'_j \neq \perp \quad n = \text{dist}(S(\ell), \iota'_i, \iota'_j)}{\langle E; S; \Delta; \boxed{\text{dist } x_i \text{ to } x_j \text{ of } x_c} \rangle \rightarrow \langle E; S; \Delta; n \rangle} \text{E-DIST} \\
\\
\frac{E(x) = \ell \quad S(\ell) = c \quad \text{fresh } \ell' \quad \langle E; \Delta; \ell; \text{op} \rangle \rightarrow \langle \delta, r \rangle \quad c' = \delta \cdot c \quad S' = S[\ell' \mapsto c'] \quad \Delta' = \text{extend}(\Delta, \ell, \ell', \delta)}{\langle E; S; \Delta; \boxed{\text{remap } x \text{ with op}} \rangle \rightarrow \langle E; S'; \Delta'; \ell', r \rangle} \text{E-REMAP-IT}
\end{array}$$

Figure 6.6: Evaluation of iterator instructions.

6.4 Lowering to Destructive Operations

So far, bookmark semantics use value semantics for collections, where every update operation allocates a fresh memory location and yields a new collection value. While this is a clean foundation for reasoning about programs, it is not how pro-

grams are efficiently executed in practice. In this section, I present a destructive lowering of SSA programs, where update operations are performed in-place rather than allocating fresh locations.

I introduce destructive forms of the update instructions: **write!** and **remap!**. These instructions have the same syntax as their value-semantics counterparts, but their evaluation differs in three key ways. First, the store is updated at the input location ℓ in place, rather than allocating a fresh location. Second, rather than extending the lineage environment Δ with a new location pair, all existing lineage functions in Δ whose context is ℓ are updated by composing δ in place, i.e., $\text{extend}(\Delta, \ell, \ell, \delta)$. The extended evaluation semantics for the destructive forms are provided in Figure 6.7

$$\begin{array}{c}
 \frac{
 \begin{array}{l}
 E(x_r) = (\ell_r, n_r) \quad E(x_c) = \ell \quad c = S(\ell) \\
 w = \text{val}(E, v) \quad \delta = \Delta(\ell_r, \ell) \quad n'_r = \delta(n_r) \\
 c' = c[n'_r \mapsto w] \quad S' = S[\ell \mapsto c']
 \end{array}
 }{
 \langle E; S; \Delta; \boxed{\text{write! } v \text{ to } x_r \text{ of } x_c} \rangle \rightarrow \langle S'; \Delta; \ell \rangle
 } \text{E-WRITE!} \\
 \\
 \frac{
 \begin{array}{l}
 E(x) = \ell \quad c = S(\ell) \quad c' = \delta \cdot c \\
 S' = S[\ell \mapsto c'] \quad \Delta' = \text{extend}(\Delta, \ell, \ell, \delta)
 \end{array}
 }{
 \langle E; S; \Delta; \boxed{\text{remap! } x \text{ with op}} \rangle \rightarrow \langle S'; \Delta'; \ell \rangle
 } \text{E-REMAP!} \\
 \\
 \frac{
 \begin{array}{l}
 E(x) = \ell \quad S(\ell) = c \quad \langle E; \Delta; \ell; op \rangle \rightarrow \langle \delta, r \rangle \\
 c' = \delta \cdot c \quad S' = S[\ell \mapsto c'] \\
 \Delta' = \text{extend}(\Delta, \ell, \ell, \delta)
 \end{array}
 }{
 \langle E; S; \Delta; \boxed{\text{remap } x \text{ with op}} \rangle \rightarrow \langle S'; \Delta'; \ell, r \rangle
 } \text{E-REMAP!-IT}
 \end{array}$$

Figure 6.7: Instruction Evaluation Rules for Destructive Updates.

Following from Section 3.7, I use the one-shot bufferization algorithm. One-shot bufferization operates on each functional update (**write** and **remap**) and con-

verts them to destructive updates (**write!** and **remap!**) if the context collection is dead immediately following the instruction.

Given a consumption-safe program, all such update operations can be lowered to their destructive forms. However, with bookmarks, the specific implementation being lowered to matters.

The fundamental issue arises from how bookmarks are encoded in the lowered program. To achieve zero performance overhead compared to traditional element references, these must be lowered to raw pointers. However, since bookmarks are immutable values in our program, pointers must be valid for their full lifetime.

This is traditionally accomplished by using *pointer stable* implementations. For example, the virtual machines for both Java and Python assume that elements are allocated separately from their collection. This enables the compiler and runtime to reference and manage these objects independently of what happens to the collection that holds them. In C++, this is accomplished by using a specific set of pointer stable classes—namely `std::map`, `std::set` and `std::list`—which store elements as separate nodes rather than inline—as in `std::vector`. Therefore, MEMOIR collections that are accessed via bookmarks can always be safely lowered to pointer stable implementations, while preserving the logical bookmark semantics.

However, pointer stable implementations are notoriously slow. They require at least one level of indirection for all accesses, and increase the number of calls to the allocator. To enable lowering to pointer unstable implementations, the compiler must explicitly reason about the stability of operations.

This is accomplished by *stability analysis*. Informally, stability analysis analyzes the use of bookmarks in a program and identifies cases where a given implementation would lead to undefined behavior. By running this analysis, the com-

piller can check if a given pointer unstable implementation can be safely lowered to for a collection. This can then be used to either select a different implementation, or warn the user about potential undefined behavior in the generated code—since stability analysis can generate false positives.

I provide a detailed definition of stability analysis later in Section 6.5.3.

6.4.1 Pointer Stability

When lowering collections from an abstract data type to a concrete implementation, lowering must be wary of undefined behavior introduced by elements moving to different locations in memory. This is expressed by the *stability predicate*, a function `stable : Nat → Bool` that yields true if the index is stable following the operation and false otherwise.

Following from Section 6.1.1, I provide the stability predicates for a resize-able array (`std::vector`) as an example. The stability function for `insert(k)` checks collection metadata—*length* and *capacity*—to determine if the underlying memory may be reallocated. Additionally, it checks that the element will not be moved to a new memory location to make room for the element:

$$\text{insert}(k)(x) = \text{length} < \text{capacity} \wedge x < k$$

The stability function for `erase(k)` checks that the element is not the one being erased, and that it will not be moved to fill the space left by the erased element:

$$\text{erase}(k)(x) = x < k$$

```

1 impl std :: vector
2   meta length: Nat
3   meta capacity: Nat
4   op insert(k: Nat, v: Nat)
5     lineage(x) x if x < k else x + 1
6     stable(x) length < capacity and x < k
7   op push_back(v: Nat)
8     lineage(x) x
9     stable(x) length < capacity
10  op erase(k: Nat)
11    lineage(x)  $\perp$  if x = k else x - 1 if x > k else x
12    stable(x) x < k
13  op erase(it: Iter as k: Nat)
14    lineage(x)  $\perp$  if x = k else x - 1 if x > k else x
15    stable(x) x < k
16    result ( $\ell$ , k)

```

Listing 6.1: Specification of `std::vector` operations.

6.4.2 Implementation Specification

To express implementation information to the compiler, I define a specification language. An example specification for `std::vector` is provided in Listing 6.1. The language defines operators **op** that can be used in **remap** instructions. Within these operators, the lineage function and stability predicate are defined. Optionally, an additional result may be output—typically used to yield fresh iterators or references. Additionally, data structure developers can express metadata—e.g., length and capacity—which can be used within the specification.

6.5 Bookmark Safety

In this section, I define two safety analyses to detect undefined behavior in both the functional (Section 6.5.2) and imperative semantics (Section 6.5.3) of bookmarks. These analyses are based on a shared *lineage graph*, described in Section 6.5.1.

6.5.1 Lineage Graph

Both analyses in this section reason about the collection operations that occur between a bookmark's definition and its use. I unify these analyses using the *lineage graph*, which represents the derived versions of a collection and the update operations that connect them. Informally, a lineage graph is constructed by taking the union of all def-use chains and labeling edges with the update operation. An example program and the lineage graph for collection value x is provided in Figure 6.9.

The lineage graph G_x is constructed by traversing the def-use chain of x . For each `$x = \text{remap } x \text{ with } op$` instruction, an edge is added from x to x' with the label op and recurse on x' . The same is done for **write** instructions and basic block arguments, but labeled with the identity id since all elements remain in the same position.

Lineage graphs are evaluated using abstract interpretation [104] over the abstract domain $L = \perp \leq ? \leq \mathbb{N}, Var \leq \top$, visualized in Figure 6.8. $n \in \mathbb{N}$ and $x \in Var$ represent concrete and symbolic elements, respectively. $?$ represents a live element at an unknown index, and $\perp$ represents a potentially invalid element. For a bookmark $b = \text{mark } x[n]$, G_x is interpreted as a sparse forward dataflow analysis, initializing the root node x to n and all other nodes to $\top$.

Since bookmarks are SSA values, every bookmark either has a unique definition site with a fixed origin index, or is a basic block argument that merges a finite set of bookmarks each with their own fixed origin index. In either case, the set of possible origin indices is statically known and finite, and the analysis evaluates the lineage graph for each possible origin independently, taking the meet $\wedge$ over all results. When the origin index of a bookmark cannot be statically determined—

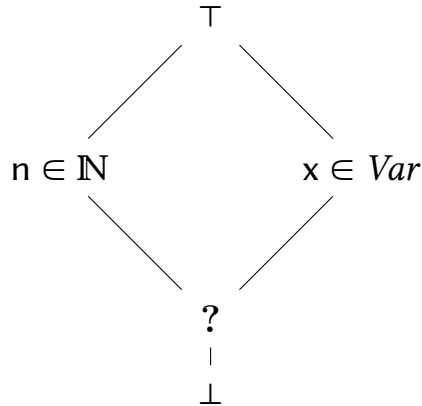


Figure 6.8: Hasse diagram of the lattice used in abstract interpretation of a lineage graph.

for example, when it flows through an opaque function call—it is conservatively initialized to $?$. This design decision is discussed further in Section 6.5.4.

At each edge, the label function is applied to the current lattice value. For each node, the meet over all incoming values is taken. For cycles, the analysis iterates until a fixed point is reached. While this analysis operates over an infinite lattice, convergence is guaranteed by widening to the $?$ value and never between concrete or symbolic indices indefinitely [13], [105].

6.5.2 Validity Analysis

A bookmark is *valid* at a use site if the element it identifies survives every collection update along every path from its definition to that use. The lineage graph is instantiated using the lineage function δ of each edge label. For concrete indices, $\delta(n)$ is evaluated directly. For symbolic indices, $\delta(x)$ is evaluated symbolically, as shown in Figure 6.9. For $?$, the analysis conservatively meets over all possible outputs, as shown in Figure 6.10c. An example is shown in Figure 6.10, where the second iteration widens $2 \wedge 1 = ?$ and conservatively approximates that

```

1 block entry (x: Seq, n: Nat, cond: Nat)
2   let b = mark x[n] in
3   if0 cond then() else()
4   block then ()
5     let x0 = remap x with erase(1) in # may erase x[n]
6     jmp join (x0)
7   block else ()
8     let x1 = remap x with insert(1) in # may move x[n]
9     jmp join (x1)
10  block join (x2: Seq)
11  let v = read b of x2 in ret v # maybe invalid

```

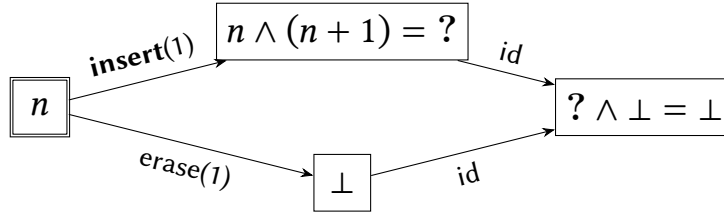


Figure 6.9: Program with an if-then-else pattern and its lineage graph, which widens symbolic arithmetic to $?$ and over-approximates meet $? \wedge \perp = \perp$.

$\text{erase}(1)(?) = \perp$.

Use sites with lattice value $\perp$ are reported as potential invalidation bugs. Use sites with concrete or symbolic indices enable further optimizations, such as dead element elimination (Section 4.1.5).

6.5.3 Stability Analysis

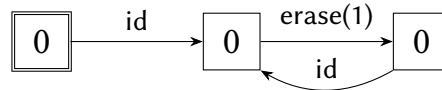
While validity analysis checks for logical invalidations, the compiler must also check for *memory invalidation* that may be caused by lowering. Memory invalidation is caused by destructive updates that invalidate existing pointers and iterators. I frame this form of memory invalidation as *stability*.

A bookmark is *stable* at a use site if the element it identifies remains physically valid in memory across every collection update along every path from its

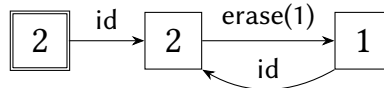
```

1  block entry (x: Seq, n: Nat, cond: Nat)
2    let b0 = mark x[0] in
3    let b2 = mark x[2] in
4    jmp body (x, 1)
5  block body (x0: Seq, i: Nat)
6    let x1 = remap x0 with erase(1) in # erases x[1:2]
7    if0 i exit (x1) body (x1, 0)
8  block join (x2: Seq)
9    let v = read b0 of x2 in # valid
10   let w = read b2 of x2 in # invalid
11   ret v,w

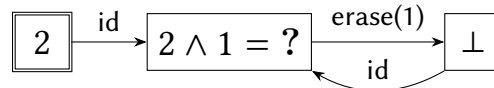
```



(a) Index 0 as input, fixpoint.



(b) Index 2 as input, after first iteration.

(c) Index 2 as input, fixpoint. The cycle results in a meet $2 \wedge 1 = ?$, which may be erased by **insert**(1).Figure 6.10: Program with a loop and its cyclic lineage graph, evaluated for bookmarks b_0 and b_2 .

```

1 std::vector<int> x = {10, 20, 30};
2 int &first = x[0]; // reference to first element
3 int &third = x[2]; // reference to third element
4 x.erase(x.begin() + 1); // erase element at x[1]
5 int u = first; // read from reference, stable => 10
6 int v = third; // read from reference, unstable => UNDEFINED

```

Listing 2: C++ program where pointer unstable operations cause undefined behavior.

definition to this use. Classic examples of this can be seen in C++ STL collection implementations, such as `std::vector`, where operations may cause elements to move to new positions. This can be seen in Listing 2, where the `erase` causes all elements after erased one to shift to the left.

As a result of the `erase` operation, the reference to the third element no longer points to element containing `30`. In general, this may result in a reference to a different element, or worse, a reference to uninitialized memory. To statically identify cases where unstable operations invalidate previously obtained bookmarks, I introduce *stability analysis*.

Stability analysis provides static guarantees that can be used during the lowering process, where the compiler reasons about how logical operations can be mapped to destructive operations on raw memory. The compiler can be configured to provide different behavior based on developer needs. If the developer wants to ensure the lowered code is safe, the compiler can emit an error when it cannot lower to the given implementation without introducing possible undefined behavior. To avoid false positives, the compiler can emit a warning, which can be checked by the developer to ensure that they are indeed safe. In other cases, the compiler may try to optimize the implementation of a given collection. In order

to safely use implementations that have unstable operations, without introducing spurious copies, the compiler takes the results of stability analysis into consideration.

Stability analysis is defined as a simple extension of validity analysis, operating over the same lattice. This is achieved by modifying the transfer function to compute the stability function on the result of validity analysis, and then meet the two results:

$$\mathbf{lineage}(L) \wedge \mathbf{stable}(\mathbf{lineage}(L)) \text{ where } L \text{ is the input lattice}$$

Worked examples from the above C++ code snippet are shown in Figure 6.11.

6.5.4 Interprocedural Analysis

So far, I have only considered intraprocedural safety analysis. I now consider how the analysis can be extended to analyze multiple functions.

Assuming that the analysis has the complete program call graph [106], safety analysis can be extended to an interprocedural analysis. This is done by merging the lineage graphs of all functions in the program, and adding id edges for each edge in the call graph.

In some cases, a complete representation of the program is not possible since the programs may be incredibly large and infeasible to compile in a single step. These use cases frequently employ techniques like ThinLTO [107], which perform whole program analysis using summaries at translation unit boundaries. Summaries can be applied to safety analysis by running an intraprocedural analysis first, sharing the results as summaries during the thin link step, and then rerunning the analysis with bookmark argument initialized to their summary. For ar-

```

block entry (x: Seq as std::vector, n: Nat, cond: Nat)
  let b0 = mark x[0] in
  let b2 = mark x[2] in
  let x1 = remap x with erase(1) in # erases x[1]
  let u = read b0 of x1 in # stable
  let v = read b2 of x1 in # unstable
  ...

```

(a) Input MEMOIR program.

```

impl std::vector
  ...
  op erase(k: Nat)
    lineage(x)  $\perp$  if x = k else x - 1 if x > k else x
    stable(x) x < k

```

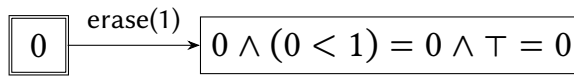
(b) Implementation specification of std::vector::erase.

```

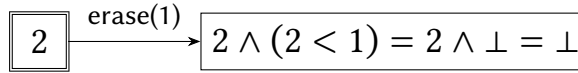
lineage(x)  $\perp$  if x = 1 else x - 1 if x > 1 else x
stable(x) x < 1

```

(c) Lineage and stability function for erase(1)



(d) Index 0 as input, fixpoint.



(e) Index 2 as input, fixpoint.

Figure 6.11: Stability analysis over a program where bookmark b_0 is stable across the erase operation, and b_2 is not, which would result in undefined behavior.

guments with multiple incoming summaries, the meet of all incoming summaries are taken.

To support partial compilation—where functions may be called from unknown functions defined outside of the current translation unit—safety analysis must consider bookmarks passed as an argument to the function. Given a bookmark argument b , safety analysis assumes its origin collection is any collection argument of compatible type, and union their lineage graphs. If no such compatible collection is found then it is evaluated against all lineage graphs in the program. The lattice point of b is initialized to either $?$ or $\perp$, depending on the desired behavior:

- $\perp$ conservatively assumes that unsafe bookmarks may be passed into the function. This causes any access to such bookmarks to be flagged as unsafe.
- $?$ optimistically assumes that no unsafe bookmarks are passed into the function. This is a valid assumption if the caller function will also be checked with the same safety analysis, since the analysis composes across functions.

6.6 C++ Library

With bookmarks, MEMOIR is expressive enough to support drop-in replacements for C++ STL collections.

A C++ snippet showing how access to a drop-in replacement for `std::vector` can be achieved with operator overloading is presented in Listing 3. The vector class holds a field for the base MEMOIR collection, which is being wrapped. This field is directly updated with new immutable collection values upon update. References and iterators hold a field with the bookmark object, and another for a pointer to the base collection. When performing an update or access via a bookmark, the

current version of the collection is obtained by dereferencing this pointer. For update operations, the result is assigned to the base collection field. This treatment ensures that all operations use the most up-to-date version of the collection, mirroring the imperative semantics of `std::vector` while obeying the SSA form of MEMOIR.

This approach provides a drop-in replacement for STL collection classes. The only caveat is that operations that rely on direct access to the backing memory are disallowed. This is done by not providing a `.data()` method nor a dereference operator for the reference class. Either of these operations can be used to obtain a raw pointer, which would enable developers to write code that violates MEMOIR semantics. When porting code to use the MEMOIR library instead of the STL, code that violates this constraint will result in a compiler error rather than a silent error.

6.6.1 Towards Zero Overhead Abstraction

Using conventional compiler optimizations, such as `mem2reg`, code using the drop-in replacement is canonicalized to simple def-use chains. This eliminates the additional fields along with the load and store instructions used to access them. In practice, I find that these optimizations are sufficient for common coding patterns.

In Section 6.4, I described how bookmarks lower to a raw pointer into memory. This provides zero-overhead access to elements via references without the need to go through the base collection. With this direct access, the only overhead left to eliminate is the additional storage for the base collection pointer. This can be accomplished with dead field elimination (Section 4.1.1), though I leave a comprehensive evaluation to future work.

```

1  template <class T>
2  struct vector {
3      Coll coll;
4
5      struct reference {
6          Mark mark; // bookmark
7          Coll *base; // pointer to base collection
8          // assignment operator, example: x[i] = value;
9          reference &operator=(const T &value) {
10             // write value to mark of *base
11             *base = memoir::write(value, mark, *base);
12             return *this;
13         }
14         // implicit conversion operator, example: int result =
15         //     x[i];
16         operator T() const {
17             // read mark of *base
18             return memoir::read(mark, *base);
19         }
20     };
21     reference operator[](size_t i) {
22         return reference(memoir::mark(coll, i), &coll);
23     }
24
25     struct iterator {
26         Iter iter; // iterator
27         Coll *base; // pointer to base collection
28         // pre-increment, example: ++it
29         iterator &operator++() {
30             // next iter of *base
31             iter = memoir::next(iter, *base);
32             return *this;
33         }
34         /* access functions same as vector::reference */
35     };
36     iterator begin() {
37         return iterator(memoir::begin(coll), &coll);
38     }
39 };

```

Listing 3: Snippet of drop-in replacement for C++ `std::vector` using MEMOIR.

6.7 Discussion

In this chapter, I introduced bookmark semantics, which resolve the fundamental tension between SSA value semantics and element references in MEMOIR. By representing references and iterators as immutable SSA values contextualized against collection versions, bookmarks can capture common C++ programming idioms while maintaining the sparse dataflow properties of SSA form. Under a copy-free lowering, bookmarks reduce to raw pointers with no runtime overhead, making them a true zero-cost abstraction.

6.7.1 Expressiveness

Bookmarks expand the set of programs that MEMOIR can express. Collections that rely solely on iterators for access—such as `bags` or `std::hive`—cannot be represented without iterator support, since they provide no index-based access interface. With bookmarks and iterators, MEMOIR can represent these collections naturally.

6.7.2 Optimizations Enabled by Bookmarks

Bookmarks enable reference reuse. By-index access requires a probe or lookup on each access, incurring the cost of the collection’s search operation. A bookmark obtained once can be reused for repeated access to the same element without repeating the lookup, reducing redundant computation.

By expressing references and iterators in MEMOIR, the compiler gains the information necessary to reason about them during lowering. In the conservative case, any collection accessed via a bookmark is lowered to a pointer-stable imple-

mentation to guarantee safety after zero-copy lowering. Using the stability analysis, introduced in Section 6.5.3, the compiler can identify collections that do not need pointer-stable implementations. For example, code that uses a `std::list` can be transformed to use a `std::vector` or `std::deque` if all bookmarks are stable under those implementations. This extends the optimization space available to ADE (Chapter 5) and other layout optimizations.

6.7.3 Limitations

The specification language requires data structure developers to manually specify lineage functions and stability predicates for each operation. While the language is concise, it is unclear how natural this is in practice. Writing correct specifications for complex operations may be error-prone, and I have not yet evaluated this on a broad set of implementations.

The zero-overhead lowering of bookmarks to raw pointers is not guaranteed in all cases. While dead field elimination can eliminate the base collection pointer when it is provably dead, this depends on the optimization pipeline and program structure. Further evaluation is needed to characterize when zero overhead is reliably achieved.

Finally, the stability analysis is conservative. It may report false positives for implementations that are pointer-stable in practice but cannot be proven so statically. In such cases, the developer must either annotate their code or accept that the compiler will conservatively restrict implementation selection.

CHAPTER 7

SSA Destruction with No Spurious Copies

MEMOIR represents data collections in an SSA form, relying on SSA destruction to lower functional updates on collections to imperative code that directly manipulates memory with destructive updates. In Section 3.7, I illustrated how this destruction step may introduce *spurious copies*, which were not present in the original imperative program. In this chapter, I explore solutions that guarantee the compilation pipeline will introduce no spurious copies.

While this question is motivated by MEMOIR, it applies to any compiler that lifts mutable resources into an SSA form. This includes data collections like arrays and tensors [22], [25], [88] along with exclusive resources, such as qubits [108], [109]. Despite the breadth of compilers that rely on this pattern, no prior work has characterized which program properties guarantee copy-free SSA destruction, nor which compiler transformations preserve them. This chapter provides that foundation.

For scalar values, SSA destruction is handled by register allocation, which assigns values to registers and spills them to the stack when registers are exhausted. Spilling a scalar is cheap—a few bytes copied to and from the stack—and the compiler tolerates it as a minor performance cost. For mutable resources like data collections, the stakes are entirely different.

When SSA destruction cannot lower a functional update to an in-place mu-

tation, it must introduce a copy of a potentially large, heap-allocated object. A single spurious copy turns an $O(1)$ operation into an $O(n)$ operation, destroying the performance of the original program. For a 1 GiB database, this is not a minor inconvenience, but a performance catastrophe!

For exclusive resources, such as qubits, the situation is worse still. A spurious copy is not merely a performance regression but a fundamental correctness violation, since these resources cannot be copied at all. As modern compilers increasingly represent large data objects and exclusive resources in SSA form, the absence of spurious copies becomes a hard requirement.

Therefore, the compiler must guarantee that every functional update can be lowered to an in-place mutation without introducing a copy. The state-of-the-art response is *one-shot bufferization*, used in MEMOIR and the MLIR tensor dialect, which analyze SSA def-use chains to maximize in-place reuse. Informally, a functional update can be destructed to an in-place mutation if the original object is never used again after the update. I refer to such updates as consumption-safe, since they are the final ‘consumer’ of the original object and can therefore take ownership. When all functional updates in a program are consumption-safe, one-shot bufferization can destruct the program to an imperative form without introducing any spurious copies. In practice, consumption-safety reduces to a simple liveness check: a functional update **let** $y = \text{update } x$ **n** is consumption-safe if x is dead after the update.

One-shot bufferization works well in practice, but relies on greedy heuristics with no formal guarantees. Crucially, it is applied *after* an arbitrary optimization pipeline, which may have already obfuscated the ownership structure of the program. Consider the following MLIR program, where global value numbering (GVN) replaces the tensor being read:

```

1  # Before GVN: no copy required
2  let t1 = write v1 to t[0] in # update index 0
3  # %t dead after, can be performed in-place
4  let x = read t1[1] in # read index 1 from t1
5  let t2 = write v2 into t1[1] in # use of t1

```

Listing 7.1: Program before GVN requires no copies.

GVN observes that `read t1[1]` is equivalent to `read t[1]` and replaces t1 with t in the **read**:

```

1  # After GVN: copy required
2  let t1 = write v1 into t[0] in
3  #! t live after update, must copy!
4  let x = read t[1] in # replaced t1 with t
5  let t2 = write v2 into t1[1] in

```

Listing 7.2: GVN transforms code to require a copy.

The original program required no copies, so the update could be performed in place. After GVN, t0 is live following the update and one-shot bufferization must conservatively introduce a copy that the transformation made necessary:

```

1  # After bufferization: spurious copy introduced
2  let t1 = copy t in #! spurious copy
3  let _ = write! v1 to t1[0] in
4  let x = read t[1] in
5  let t2 = write! v2 to t1[1] in

```

Listing 7.3: Spurious copy introduced during bufferization.

This is not a pathological case. Standard optimization passes routinely introduce spurious copies in this way, and there is currently no principled framework for compiler developers to reason about or prevent them. In our evaluation, standard optimization passes and pipelines in LLVM introduced spurious copies in 21.96% of test cases. I distill the common cases for consumption-safety violations into three general transformations and the intuition behind them:

- *Injection* may introduce a new instruction that uses a consumed variable, or inject an update at a location where its object operand is live.
- *Code motion* may hoist an update above a use of the consumed variable, leaving it live in the continuation.
- *Replacement* may substitute a consumed variable back into a use site, as seen in the GVN example above.

While some transformations may introduce spurious copies, consumption-safety gives compiler developers a principled framework to reason about and respond to them. In this chapter, I present two approaches to preserve consumption-safety.

Tightening adds lightweight applicability guards to transformations, blocking them from introducing a violation. These guards are minimal—they only block what is necessary to preserve consumption-safety in the transformed code—and our evaluation shows they have negligible impact on compilation time. However, tightening is conservative, since it cannot anticipate future transformations that may restore consumption-safety. For example, a tightened GVN may block a replacement that would have been made safe by a subsequent dead code elimination pass. This can cascade, blocking high-value optimizations later in the pipeline.

Repair takes an optimistic approach, where transformations run freely and repair passes restore consumption-safety after the fact. This is preferable when tightening would block high-value optimizations. In our evaluation, tightened LICM blocked downstream SLP vectorization on some benchmarks, while repair avoided this entirely. Our repair passes eliminated all spurious copies introduced by standard LLVM optimization passes and pipelines in our evaluation.

In some cases, a spurious copy may be acceptable when the benefit of the optimization outweighs the cost, such as copying a small array to enable vectorization. This work ensures this is an explicit, informed decision rather than an accidental consequence.

The remainder of this chapter is organized as follows: Section 7.1 distills MEM-OIR into formal components that I use to generalize our results across SSA forms. Section 7.2 formalizes one-shot bufferization and derives consumption-safety as the condition for no spurious copies. Section 7.3 proves that standard SSA construction from imperative programs yields consumption-safe programs by construction. Section 7.4 decomposes common compiler transformations into their fundamental building blocks and proves that each either naturally preserves consumption safety or derives additional applicability guards that make it do so. As an

alternative, Section 7.5 presents algorithms that attempt to restore consumption-safety after arbitrary transformations are applied. Finally, Section 7.6 I implement and evaluate both approaches in LLVM, demonstrating 100% effectiveness with negligible compilation overhead.

7.1 Language Definition

To capture the essential semantics of modern compiler intermediate representations, I present a base CORE language. Two extensions of CORE capture the two paradigms relevant to MEMOIR and common MLIR dialects. IMP models an imperative language with reference semantics, where updates destructively update the value in-place. SSA models a functional language with value semantics, where functional updates yield fresh, immutable values.

These languages operate on opaque objects via basic allocation, read and update instructions. Doing so, the underlying data structure is abstracted away, and the results can generalize across concrete data types.

The compilation pipeline transforms programs from IMP through SSA to IMP, as shown in Figure 7.1 This mirrors the process of raising imperative buffer representations to functional SSA form, and subsequently lowering them back to efficient imperative operations.

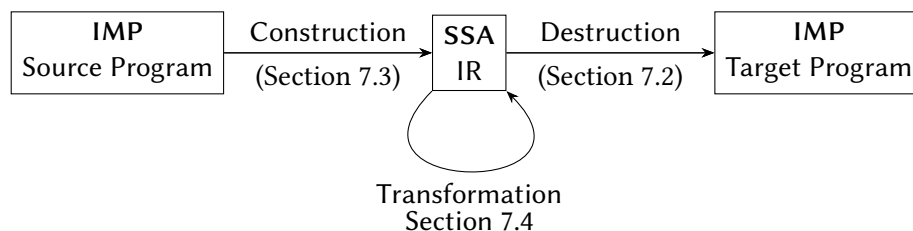


Figure 7.1: Compilation pipeline model used in this chapter.

CORE		$x \in \text{Variables}, l \in \text{Labels}, n \in \mathbb{N}$	
$f \in \text{Func}$	$::=$	$\bar{b}$	<i>function</i>
$b \in \text{Block}$	$::=$	block $l(\bar{x})\ e$	<i>basic block</i>
$e \in \text{Expr}$	$::=$	let $x = i$ in e	<i>binding</i>
		$ $ t	<i>terminators</i>
$i \in \text{Inst}$	$::=$	alloc	<i>default constructor</i>
		$ $ copy x	<i>copy constructor</i>
		$ $ read x	<i>read-only use</i>
		$ $ call $x(\bar{x})$	<i>function call</i>
$t \in \text{Term}$	$::=$	if0 $v\ l(\bar{x})\ l(\bar{x})$	<i>conditional branch</i>
		$ $ jmp $l(\bar{x})$	<i>unconditional branch</i>
		$ $ ret x	<i>function exit</i>
IMP extends CORE			
$i \in \text{Inst}$	$::=$	$\dots \mid$ update! $x\ n$	
SSA extends CORE			
$i \in \text{Inst}$	$::=$	$\dots \mid$ update $x\ n$	

Figure 7.2: Syntax of CORE, IMP and SSA

For simplicity, the read and update operations defined in this section only distill core MEMOIR. The bookmarks extension requires the compiler to reason about a separate lifetime. Instead of modeling that explicitly in this chapter, it is handled separately by the stability analysis presented in Section 6.5.3. I restrict the formalization to alias-free programs because CORE models the IR after promotable objects have been raised to SSA variables. Section 7.1.4 details how non-promotable, aliased memory is safely handled by the compiler before our framework applies, and why this separation of concerns does not compromise the approach.

7.1.1 Core Language (CORE)

CORE is a base language for the compiler, based on the control flow graph [8] using basic block arguments [3] instead of ϕ -nodes [4], [52]. The syntax is shown in Figure 7.2.

To simplify proofs later in this chapter, basic blocks are closed, such that all variables used in a block are either a block argument or bound by a local **let**. This makes liveness a syntactic property, where a variable is live iff it appears free in the continuation. Any SSA program can be converted to one with closed basic blocks by inserting basic block arguments for each variable live at the block entrance, and updating dominated uses.

CORE abstracts the underlying data structures as opaque objects with a set of distilled operations: `alloc` constructs a fresh object. `copy x` constructs a fresh object and deep copies `x` to it. `read x` is a non-destructive observation of `x`'s state.

Our semantics are defined on the semantic domain in Figure 7.3. The underlying data is represented values as a history of updates. This generalizes results across any data type with value semantics. The operational semantics of expressions in CORE are defined as small-step semantics in Figure 7.5. I write $E[\overline{x_{new}}/\overline{x_{old}}]$ for variable substitution in the environment domain. The E-LET uses the instruction evaluation rules defined in Figure 7.4.

$\ell \in Loc$	(memory location)
$n \in \mathbb{N}$	(abstract update)
$H \in History ::= \epsilon \mid H \cdot n$	(sequence of updates)
$v \in Val ::= \ell \mid n$	(value)
$E \in Env = Var \rightarrow Loc$	(environment)
$S \in Store = Loc \rightarrow History$	(store)

Figure 7.3: Semantic domain of CORE, SSA and IMP.

$$\begin{array}{c}
\frac{\ell \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{alloc}} \rangle \rightarrow \langle S[\ell \mapsto \epsilon]; \ell \rangle} \text{E-ALLOC} \\
\\
\frac{E(x) = \ell \quad S(\ell) = H \quad \ell' \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{copy } x} \rangle \rightarrow \langle S[\ell' \mapsto H]; \ell' \rangle} \text{E-COPY} \\
\\
\frac{E(x) = \ell \quad S(\ell) = H}{\langle E; S; \boxed{\text{read } x} \rangle \rightarrow \langle S; \text{observe}(H) \rangle} \text{E-READ}
\end{array}$$

Figure 7.4: Instruction Evaluation of CORE.

$$\begin{array}{c}
\frac{\langle E; S; i \rangle \rightarrow \langle S'; v \rangle}{\langle E; S; \boxed{\text{let } x = i \text{ in } e} \rangle \rightarrow \langle E[x \mapsto v]; S'; e \rangle} \text{E-LET} \\
\\
\frac{f(l) = \boxed{\text{block } l(\overline{x_p}) e}}{\langle E; S; \boxed{\text{jump } l(\overline{x_a})} \rangle \rightarrow \langle E[\overline{x_p}/\overline{x_a}]; S; e \rangle} \text{E-JMP} \\
\\
\frac{E(x) = 0 \quad f(l_0) = \boxed{\text{block } l_0(\overline{x_p}) e}}{\langle E; S; \boxed{\text{if0 } x \text{ } l_0(\overline{x_0}) \text{ } l_1(\overline{x_1})} \rangle \rightarrow \langle E[\overline{x_p}/\overline{x_a}]; S; e \rangle} \text{E-IF0-THEN} \\
\\
\frac{E(x) \neq 0 \quad f(l_1) = \boxed{\text{block } l_1(\overline{x_p}) e}}{\langle E; S; \boxed{\text{if0 } x \text{ } l_0(\overline{x_0}) \text{ } l_1(\overline{x_1})} \rangle \rightarrow \langle E[\overline{x_p}/\overline{x_a}]; S; e \rangle} \text{E-IF0-ELSE}
\end{array}$$

Figure 7.5: Small-step operation semantics of CORE.

7.1.2 IMP Language and Reference Semantics

IMP extends CORE with a destructive update operation (**update!**), which performs an in-place update and returns the same reference. In IMP, variables utilize reference semantics, where a variable holds a reference to mutable program state,

making reference passing a cheap $O(1)$ operation. For example, consider:

let $y = \mathbf{update!} \ x \ 7$ **in** e

After the **update!** executes (i.e., in the continuation e) x has been modified in-place and y refers to the same object.

The semantics for destructive updates in IMP are provided in Figure 7.6. Notice that the resulting location l is the same location as the operand, meaning the update is performed in-place. The store S is destructively updated to append the new operation n to the history H .

$$\frac{E(x) = \ell \quad S(\ell) = H}{\langle E; S; \mathbf{update!} \ x \ n \rangle \rightarrow \langle S[\ell \mapsto H \cdot n]; \ell \rangle} \text{E-UPDATE-IMP}$$

Figure 7.6: Instruction Evaluation of IMP.

7.1.3 SSA Language and Value Semantics

SSA extends CORE with a functional update operation (**update**), which yields a fresh updated object leaving the input unmodified. Variables have value semantics, where variables hold references to immutable state, giving referential transparency. For example, consider:

let $y = \mathbf{update} \ x \ 7$ **in** e

After the **update** executes (i.e., in the continuation e) both x and y are valid, and x is unmodified.

The semantics for functional updates in SSA are provided in Figure 7.7. Notice that the resulting location l' is fresh, meaning that the update yields a new value.

The store S is solely extended, with the new operation n appended to a new history.

$$\frac{E(x) = \ell \quad S(\ell) = H \quad \ell' \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{update } x \ n} \rangle \rightarrow \langle S[\ell' \mapsto H \cdot n]; \ell' \rangle} \text{E-UPDATE-SSA}$$

Figure 7.7: Instruction Evaluation of SSA.

7.1.4 Handling Memory Aliasing

CORE is restricted to alias-free programs, representing the state of the IR after memory promotion (e.g., LLVM `mem2reg`). When a compiler can establish that no aliasing pointers interact with an object, it promotes it to an SSA variable. When aliasing cannot be ruled out, the compiler falls back to a load-update-store pattern, as shown below:

```

1 // Aliased aggregate --> promotion blocked
2 // load-update-store preserved
3 %a1 = memref.load %buf[] : memref<vector<4xf32>>
4 %a2 = vector.insert %v, %a1[%i] // update
5     memref.store %a2, %buf[] : memref<vector<4xf32>>

```

Listing 7.4: Memory promotion is blocked by aliasing.

The loaded SSA variable `%a1` is consumed by the update and never used again, which is consumption-safe. When aliasing can be ruled out, `mem2reg` promotes the object directly to an SSA variable, eliminating the load-update-store pattern entirely and entering our framework. Our formalization therefore captures promotable objects, while aliased objects are handled correctly, if pessimistically, by the load-update-store fallback. This restriction also applies to parallel programs, where concurrent access via aliasing pointers results in a similarly pessimistic IR.

7.2 SSA Destruction Without Spurious Copies

I formalize SSA destruction using the one-shot bufferization algorithm, as described in Section 3.7. Interesting rules are shown in Figure 7.8. I prove that one-shot bufferization preserves semantics by forward simulation in Section A.12.2. This result is not modular, and requires that the entire program be given.

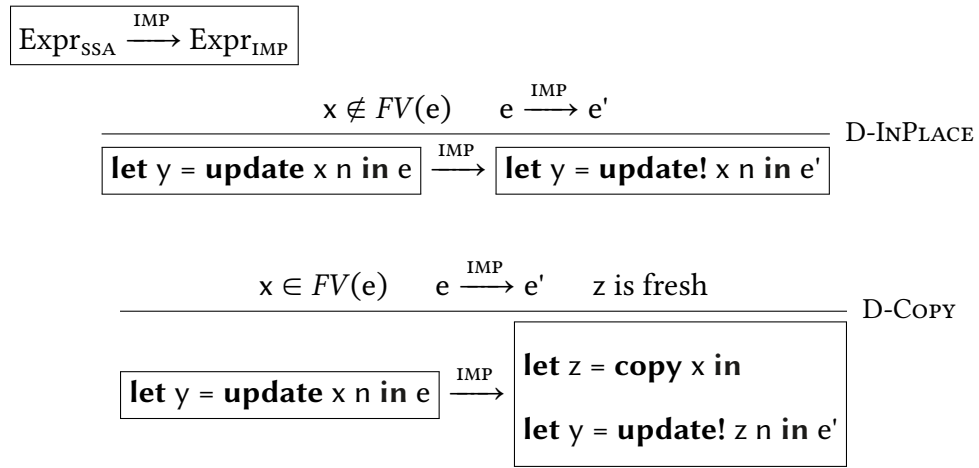


Figure 7.8: SSA Destruction relation rules of interest. D-INPLACE destructs a functional update to an in-place destructive update. D-COPY introduces a *spurious copy* that was not present in the original program.

I am specifically interested in SSA programs where all **update** operations are destructed by D-INPLACE, since D-COPY introduces a spurious copy. This requires that the object operand of every update is dead in the continuation (i.e., it does not appear free). I call this property consumption-safety, formalized as a syntactic check in Figure 7.9. Since basic blocks are closed, deadness is a syntactic check in our language: a variable is dead if it does not appear free in the continuation.

$$\boxed{\text{Expr}_{\text{SSA}} \text{ ok}}$$

$$\frac{x \notin FV(e) \quad e \text{ ok}}{\boxed{\text{let } y = \text{update } x \text{ n in } e} \text{ ok}} \text{ S-UPDATE} \qquad \frac{Op(i) \neq \text{update} \quad e \text{ ok}}{\boxed{\text{let } x = i \text{ in } e} \text{ ok}} \text{ S-LET}$$

Figure 7.9: Consumption-safety of SSA, expressing the conditions for all **update** operations in an SSA expression to be destructed into in-place updates (D-INPLACE).

I prove that a consumption-safe SSA expression can always be destructed without introducing a spurious copy in lemma 7.2.1. A sketch of the proof is provided below, with the full proof provided in Section A.10.

Lemma 7.2.1 (Consumption-safety Guarantees Copy-Free Destruction). *Given a consumption-safe SSA expression e , the destruction $e \xrightarrow{\text{IMP}} e'$ will not evaluate the D-COPY rule, resulting in an IMP expression e' containing zero spurious copies.*

Proof Sketch. This is done by showing that D-COPY cannot fire on a consumption-safe expression. Informally, consumption-safety requires that the consumed variable x_0 is dead in the continuation of every update instruction. D-COPY can only fire when this is not the case, therefore it is unreachable, and all **update** instructions are destructed by D-INPLACE, introducing no spurious copies. Copy instructions present in the original SSA expression are passed through unchanged and are not spurious, since they were not introduced by destruction. $\square$

7.3 Construction Yields Consumption-Safe Programs

The SSA construction relation in Figure 7.10 converts an IMP program into an SSA program by replacing each destructive update with a functional update, and substituting the result variable for the consumed operand in the continuation. This substitution is key. The consumed operand x is replaced everywhere in the continuation by the fresh result variable y , so x is dead after the **update**. As a result,

all functional updates in the constructed program are consumption-safe by construction. This is stated in lemma 7.3.1, which is proved by showing that each constructed update is naturally consumption-safe, since the updated value is replaced by the fresh one in the continuation.

Lemma 7.3.1 (SSA Constructed Expressions are Consumption-Safe). *Given an IMP expression e , the constructed SSA expression e' is consumption safe.*

I provide a formal proof that SSA construction always yields a consumption-safe program in Section A.11. I also prove that SSA construction preserves semantics by forward simulation in Section A.12.3.

$$\boxed{\text{Expr}_{\text{IMP}} \xrightarrow{\text{SSA}} \text{Expr}_{\text{SSA}}}$$

$$\boxed{\text{let } y = \text{update! } x \text{ n in } e} \xrightarrow[\text{SSA}]{\begin{array}{c} e \xrightarrow{\text{SSA}} e' \\ \text{C-UPDATE} \end{array}} \boxed{\text{let } y = \text{update } x \text{ n in } e'[y/x]}$$

Figure 7.10: Key rule of the SSA Construction relation.

7.4 Transformation

While SSA construction guarantees a consumption-safe program, modern compilers apply transformations that may violate this property. Rather than proving preservation on a case-by-case basis—which would be exhausting and non-exhaustive—I decompose transformations into their fundamental building blocks. I do not claim these building blocks are complete, however they represent the vast majority of compiler transformations and provide a mapping from the Allen & Cocke’s Catalog of Optimizing Transformations [110] to our building blocks in Section A.13. I prove that each building block either preserves consumption-safety¹, or derive the necessary checks to make it so.

¹A subset of these proofs restate prior results [111], [112].

<i>Building Block</i>	<i>Naturally Preserves?</i>	<i>Examples</i>
Elimination	✓	Dead code elimination
Inlining	✓	Inlining, Loop unrolling
Injection	✗	Bounds check insertion
Code Motion	✗	Sink, Hoist, LICM
Replacement	✗	Value numbering, CSE

Table 7.1: Summary of transformation building blocks, if they preserve consumption-safety, and example compiler passes.

A table summarizing the results of this section is provided in Table 7.1. Elimination naturally preserves consumption-safety, since it can only reduce the set of live variables at an update, never extend it. Inlining preserves consumption-safety, since the inlined variables were already live at the original use site, no new variables are made live at any update. Injection, code motion, and replacement do not naturally preserve consumption-safety. The intuition is straightforward:

- *Injection* may introduce a new instruction that uses a consumed variable, or inject an update at a location where its object operand is live.
- *Code motion* may hoist an update above a use of the consumed variable, leaving it live in the continuation.
- *Replacement* may substitute a consumed variable back into a use site, as seen in the GVN example (Listing 7.2).

I model building blocks as a transformation relation $e_1 \Rightarrow e_2$ in Figures 7.11 and 7.12. A transformation preserves consumption-safety if it does not extend the set of free variables in the continuation of any update instruction.

I model the program as a single function, since all analysis and transformation defined in this section are intraprocedural. I assume an input function f in all rules. I write $f(\ell) = b$ to mean that function f contains a block b with label ℓ .

I extend the expression syntax with binding contexts B , which range over sequences of let-bindings with a hole:

$$B ::= \square \mid \mathbf{let\ } x = i \mathbf{\ in\ } B \mid \mathbf{let\ } x = i \mathbf{\ in\ } t$$

I write $B[e]$ for the capture-avoiding substitution of e into the hole.

I write $Consumed(B)$ for the set of variables updated in binding context B :

$$\{x \mid \boxed{\mathbf{update\ } x\ n} \in B\}$$

I write $Defined(B)$ for the set of variables defined in binding context B :

$$\{x \mid \boxed{\mathbf{let\ } x = i \mathbf{\ in\ } e} \in B\}$$

A variable x dominates variable y ($x \mathbf{dom}\ y$) iff x must be defined before y , i.e., given a block $\boxed{\mathbf{block\ l\ } (\overline{x_{args}})\ B[\mathbf{let\ } y = i \mathbf{\ in\ } e]}$

$$x \mathbf{dom}\ y \iff x \in \{\overline{x_{args}}\} \vee y \in Defined(B)$$

I write $\{\overline{x}\} \mathbf{dom}\ y$ to mean $\forall x_i \in \{\overline{x}\}. x_i \mathbf{dom}\ y$.

7.4.1 Elimination

Elimination (T-ELIM in Figure 7.11) removes instructions from the program, commonly used for dead code elimination, dead global elimination, and unreachable

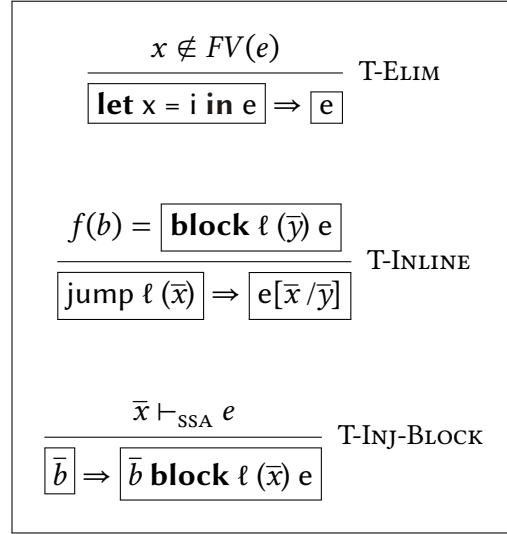


Figure 7.11: Transformations that naturally preserve consumption-safety.

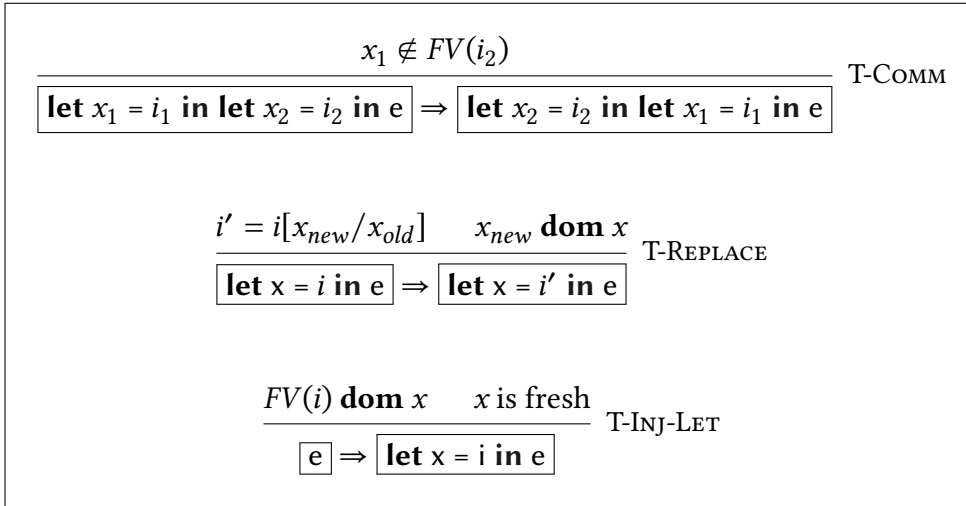


Figure 7.12: Transformations that do not preserve consumption-safety, and need to be tightened.

$$\begin{array}{c}
\frac{Op(i_2) \neq \mathbf{update} \quad x_1 \notin FV(i_2)}{\boxed{\mathbf{let } x_1 = i_1 \mathbf{ in } \mathbf{let } x_2 = i_2 \mathbf{ in } e} \Rightarrow \boxed{\mathbf{let } x_2 = i_2 \mathbf{ in } \mathbf{let } x_1 = i_1 \mathbf{ in } e}} \text{TT-COMM-LET-LET} \\
\\
\frac{i_2 = \boxed{\mathbf{update } x_0 \ n} \quad x_1 \notin FV(i_2) \quad x_0 \notin FV(i_1)}{\boxed{\mathbf{let } x_1 = i_1 \mathbf{ in } \mathbf{let } x_2 = i_2 \mathbf{ in } e} \Rightarrow \boxed{\mathbf{let } x_2 = i_2 \mathbf{ in } \mathbf{let } x_1 = i_1 \mathbf{ in } e}} \text{TT-COMM-LET-UPD} \\
\\
\frac{i' = i[x_{new}/x_{old}] \quad x_{new} \mathbf{dom } x \quad x_{new} \notin Consumed(B)}{\boxed{B[\mathbf{let } x = i \mathbf{ in } e]} \Rightarrow \boxed{B[\mathbf{let } x = i' \mathbf{ in } e]}} \text{TT-REPLACE} \\
\\
\frac{x_0 \notin Consumed(B) \quad x_0 \notin FV(e) \quad x_0 \mathbf{dom } e \quad x \text{ is fresh}}{\boxed{B[e]} \Rightarrow \boxed{B[\mathbf{let } x = \mathbf{update } x_0 \ n \mathbf{ in } e]}} \text{TT-INJ-UPD} \\
\\
\frac{Op(i) \neq \mathbf{update} \quad Consumed(B) \cap FV(i) = \emptyset \quad FV(i) \mathbf{dom } x \quad x \text{ is fresh}}{\boxed{B[e]} \Rightarrow \boxed{B[\mathbf{let } x = i \mathbf{ in } e]}} \text{TT-INJ-LET}
\end{array}$$

Figure 7.13: Tightened transformations from Figure 7.12.

code removal. I find that elimination naturally preserves consumption-safety, as stated in lemma 7.4.1.

Lemma 7.4.1 (T-ELIM Preserves Consumption-safety). *Given a consumption-safe SSA expression e the transformation $e \Rightarrow_{\text{T-ELIM}} e'$ preserves consumption-safety.*

Proof Sketch. Elimination strictly removes free variables from the continuation of **update** instructions. As a result, the transformation cannot introduce a linearity violation. More concretely, T-ELIM emits a subexpression of the original expression. Since the original expression is well-formed, the subexpression must be well-formed. The full proof provided is in Section A.4. $\square$

7.4.2 Inlining

Inlining (T-INLINE in Figure 7.11) alters the control flow graph by merging functions or basic blocks into one another. It is frequently used for control flow simplification and function inlining. Since the language only expresses programs with a single function, I do not include a rule for function inlining. However, function inlining can be achieved by injecting the blocks from another function (T-INJ-BLOCK), replacing the call with a jump into the entry block, and then applying T-INLINE. I find that inlining naturally preserves consumption-safety, as stated in lemma 7.4.2.

Lemma 7.4.2 (T-INLINE Preserves Consumption-safety). *Given a consumption-safe SSA expression e the transformation $e \Rightarrow_{\text{T-INLINE}} e'$ preserves consumption-safety.*

Proof Sketch. All free variables in the block being inlined are replaced with the block arguments passed to the jump. As a result, the free variables of the transformed expression are the same as those in the original jump expression. Therefore, no outer **update** instruction sees a change in the free variables of its continuation. The full proof is in Section A.5. $\square$

7.4.3 Injection

Injection adds new instructions and blocks to the program. Common examples include bounds check insertion, critical edge splitting, and loop unswitching. I find that injecting a basic block (T-INJ-BLOCK in Figure 7.11) naturally preserves consumption-safety. The full proof is provided in Section A.6

However, instruction injection (T-INJ-LET in Figure 7.12) does not preserve consumption safety and requires tightening. Informally, injecting a let-binding may cause a variable that was dead to become live. This may cause an unrelated **update** to become invalid. Additionally, if an **update** instruction is injected, it may consume a variable that is still live in its continuation. As a result, the injected update would be invalid. To tighten these cases, injection need two checks. First, the injected instruction's operands must not be consumed in the outer binding context. That is, $x_0 \notin Consumed(B)$ in TT-INJ-UPD and $Consumed(B) \cap FV(i) = \emptyset$ in TT-INJ-LET. Second, for injected **update** instructions, that the consumed operand is dead in the continuation ($x_0 \notin FV(e)$ in TT-INJ-UPD). The proof is provided in Section A.7

7.4.4 Code Motion

Code Motion moves instructions without changing branch targets, frequently used for sinking, loop-invariant code motion, and scheduling. I express code motion within a block as commuting **let** expressions in T-COMM (Figure 7.12).

I find that code motion preserves consumption-safety for most cases. However, this transformation does not preserve consumption-safe when hoisting an **update** above a use of the consumed variable. For this case, I derive a single tightening guard to check that the consumed variable does not appear free in the

instruction it is being hoisted past ($x_0 \notin FV(i_1)$ in TT-COMM-LET-UPD). Conversely, I find that *sinking* an **update** and commuting non-update instructions is always consumption-safe thanks to existing guards used to preserve dominance in strict SSA form ($x_1 \notin FV(i_2)$ in TT-COMM-LET-LET and TT-COMM-LET-UPD). The proof is provided in Section A.8

7.4.5 Replacement

Replacement (T-REPLACE in Figure 7.12) substitutes different variables in instruction operands. Examples include value numbering and common subexpression elimination (CSE).

I find that arbitrary replacement does not preserve consumption-safety and requires tightening. Informally, replacing one variable with another may extend the set of free variables in the resulting expression. If the replacement variable is consumed in the outer binding context of the user, an outer update will become ill-formed. Take the GVN example from Listing 7.2, where an update operation is bypassed because it does not impact the state observed by a later read. To tighten this case, the transformation must check if the replacement variable is consumed in the outer binding context ($x_{new} \notin Consumed(B)$ in TT-REPLACE). The proof is provided in Section A.9.

7.4.6 Missed Opportunities Due to Tightening

Tightening is inherently conservative. It cannot anticipate future transformations that may restore consumption-safety. Consider Listing 7.1, where TT-REPLACE would block a replacement because t_0 is consumed in the outer binding context. If permitted, subsequent transformations may render t_1 dead, allowing dead code

elimination to remove it and restore consumption-safety. As a result, prematurely blocked transformations can lead to missed optimization opportunities. I evaluate such false positives in Section 7.6 (RQ7.6.1).

Blocked transformations can also cascade to later optimizations [113]. For example, a tightened guard for loop-invariant code motion may prevent hoisting an update out of a loop. As a result, the remaining loop-carried dependence can block automatic vectorization. I evaluate this in Section 7.6 (RQ7.6.3).

For performance-critical domains where these missed opportunities are unacceptable, I propose an optimistic compilation pipeline in Section 7.5 that permits violations and attempts to repair them after the fact.

7.5 Repairing Violations

I propose ERRATA as an alternative to tightening, where optimization passes run freely and repair passes restore consumption-safety after the fact. Since repair cannot be guaranteed for arbitrary transformations², I take a pragmatic approach guided by the following insight:

Standard compiler transformations do not randomly transform code, but rather operate conservatively using information available in the program. Consequently, when an optimization induces a consumption-safety violation, the information necessary to repair the def-use chain is almost always preserved in the resulting program.

In Sections 7.5.1 and 7.5.2, I present repair transformations adapted from prior work on functional languages [114], [116], [117] that leverage this information to eliminate consumption-safety violations. I then propose a speculative compila-

²This is equivalent to the functional aggregate update problem [114], which is undecidable by Rice’s theorem [115].

tion pipeline in Section 7.5.3 that guarantees a final consumption-safe program by rolling back transformations that could not be repaired. In our evaluation (Section 7.6.3), I find that ERRATA is effective in practice, but lacks the formal guarantees of ALMANAC.

7.5.1 Schedule

Many consumption-safety violations can be repaired by re-scheduling instructions, an approach adapted from Sastry [116]. Our scheduling repair uses two global transformations:

Sink moves an update to the latest program point that dominates all of its uses, delaying it until the consumed operand is no longer live in the continuation.

Hoist moves a read immediately before the update that consumes its operand, shortening the live range of the consumed variable. At each block boundary crossed, a fresh basic block argument threads the read through to any blocks that use it. The transformation only applies if the update dominates all uses of the read. To handle interactions between multiple reads and updates, instructions are processed in dominator-tree order. This avoids the need for a complicated solver for simultaneous hoisting [116]. I found that this pragmatic approach was sufficient for the evaluation.

7.5.2 Rethreading

Violations can also be repaired using information about how the structure is updated and observed. In the GVN example, t_0 is live after the update because a subsequent read observes index 1 (an index the update did not modify). Since the read and update are non-interfering, the read is *rethreaded* to observe t_1 instead,

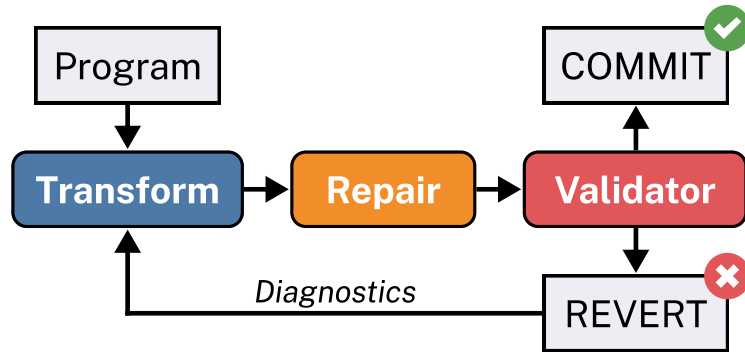


Figure 7.14: Speculative transformation conditioned on successful repair. Combining repair with rollback enables optimistic transformation with guaranteed consumption-safety.

eliminating the violation without changing semantics:

```

1  # After Rethreading: consumption-safe
2  let t1 = write v1 to t0[0] in # t0 consumed
3  let x1 = read t1[1] in # rethreaded to use t1
4  let t2 = write v2 to t1[1] in ...

```

Whether a rethread is valid depends on the semantics of the underlying collection type. For example, reads and writes to different indices of a tensor are always non-interfering. Rethreading is performed globally via a sparse data-flow analysis, replacing all non-interfering uses of the consumed operand that are dominated by the update.

7.5.3 Speculative Transformation

I propose a speculative compilation pipeline (Figure 7.14) that combines the optimization freedom of ERRATA with the formal guarantee of ALMANAC. In this pipeline, transformations that violate consumption-safety are followed by a repair attempt. If repair fails, the transformation is reverted. I evaluate the feasibility of this pipeline by measuring the compilation time overhead of its components in

RQ7.6.5.

7.6 Evaluation

In this section, I evaluate the proposed solutions to preserving consumption-safety in the compiler. I answer the following research questions:

RQ7.6.1 Does tightening block valid transformations?

RQ7.6.2 Can default pipelines reliably repair violations?

RQ7.6.3 What is the performance cost of missed optimizations due to tightening?

RQ7.6.4 Can dedicated repair passes safely recover these missed opportunities?

RQ7.6.5 Do ALMANAC and ERRATA introduce reasonable compilation time overheads?

7.6.1 Experimental Setup

I implement and evaluate our solutions in LLVM [2] (version 18.1.8). I evaluate on LLVM rather than MLIR or MEMOIR because LLVM has the most aggressive and diverse set of optimizing transformations, providing the most rigorous stress test of our approach.

I implement the language model using LLVM’s aggregate value types (array, struct and vector), where `insertvalue` and `extractvalue` directly map to **update** and **read** from our formalism, respectively. I implement a *validator* that checks consumption-safety using conventional live-variable analysis, mirroring the free-variable check in our formalism.

I evaluate on the LLVM test suite (318 C/C++ programs) and a custom random program generator. Our generator produces programs with structured control

flow³. Generated programs are consumption-safe by construction by threading live aggregates as block arguments throughout the control flow graph. I use a maximum nesting depth of eight and sequence length of three to create complex, yet reasonably sized, input programs.

7.6.2 Tightened Transformations with ALMANAC

I implement the ALMANAC pipeline by extending LLVM transformations with the tightened checks derived in Section 7.4. The following modifications were made for each class of transformation to explicitly bridge our formal rules with the LLVM C++ API:

Code Motion (e.g., `SimplifyCFG`, `SpeculativeExecution`, `JumpThreading`): When moving an update instruction to a new program point, the consumed operand is checked to ensure that it is not live at the destination. This is done by checking that it has no uses that are reachable from the destination without passing through the definition of the consumed operand. This directly enforces TT-COMM-LET-UPD and is implemented once in `isSafeToSpeculativelyExecute`, which is shared across all code motion passes.

Replacement (e.g., `Inst Combine` and `GVN`): When replacing an aggregate type at a use site, check that the new value is not consumed by any update instruction that dominates the use site. This implements the TT-REPLACE guard.

Injection (N/A): There are no cases where fresh updates are injected by transformations.

The experimental setup for ALMANAC is shown in Figure 7.15. For a given program, I compile using both LLVM and ALMANAC. The resulting programs are then

³Random CFG generators were not amenable since most LLVM passes expect reducible control flow.

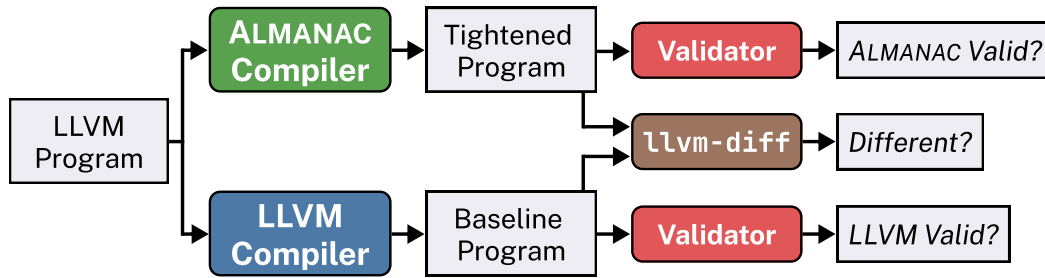


Figure 7.15: Experimental setup used to evaluate ALMANAC tightened transformations.

checked for consumption-safety using the validator, and checked for structural difference using `llvm-diff`. I use these measurements to determine the result of compilation, as shown in the table on the right hand side of Figure 7.15:

- *Success* if the output of LLVM is not consumption-safe, but the output of ALMANAC is.
- *Failure* if the output of ALMANAC is not consumption-safe. If our implementation is true to the results of Section 7.4, no failures are expected.
- *False Positive* if the output of both LLVM and ALMANAC is consumption-safe, but the programs differ. This implies that tightening blocked a transformation that would have been valid, as discussed in Section 7.4.6.

7.6.3 Repairing Violations with ERRATA

I implement the ERRATA pipeline by writing the repair transformations from Section 7.5 as LLVM passes.

Sink: Convert all `select` instructions that operate on aggregate types with branches, then apply off-the-shelf LLVM sink transformation to updates that cause linearity violations.

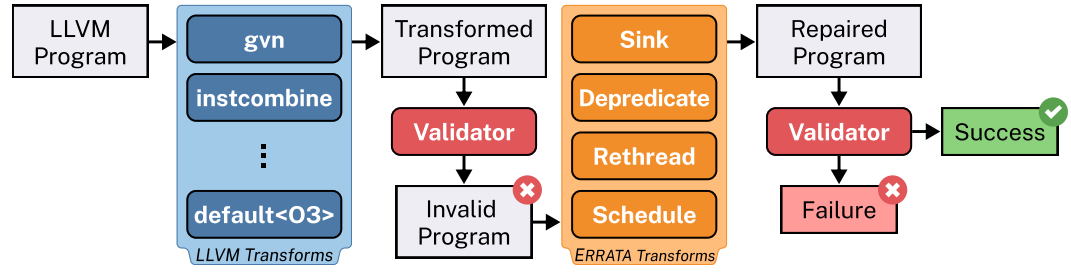


Figure 7.16: Experimental setup used to evaluate ERRATA repair transformations.

Hoist: Given a read that uses an object operand consumed in the outer context, move the read instruction before all updates that consume the object.

Rethread: Given an update whose operand is live after the instruction, replace all non-interfering uses of the operand with the updated value iff the use is dominated by the update.

Repair passes are run immediately following an LLVM transformation pass if it introduces a violation, as shown in Figure 7.16. RQ7.6.4 evaluates the granularity of repair, i.e., after individual transformation or pipeline of transformations. I evaluate the compilation time of repair passes and validation to estimate the feasibility of the speculative pipeline described in Section 7.5.3.

RQ7.6.1: Does tightening block valid transformations?

Result: the ALMANAC implementation preserves consumption safety in 100% of test cases, with a 5.72% false positive rate attributable to conservative guards in replacement and code motion passes.

I generate 1,000,000 random LLVM programs transformed by 24 passes and 5 default pipelines, for 29,000,000 total test cases. Baseline LLVM introduced violations in 6,368,144 cases (21.96%), all of which ALMANAC avoided. ALMANAC outputs 0 programs that violate consumption-safety, validating our implementation of the tightened guards derived in Section 7.4. However, ALMANAC yields false positives in 1,658,015 (5.72%) cases.

	<i>Transform</i>	<i>Successes</i>	<i>False Positives</i>	<i>Failures</i>
<i>Passes</i>	SimplifyCFG	946,263	6	0
	SpeculativeExecution	946,305	0	0
	InstCombine	793,799	155,088	0
	GVN	13,010	401,502	0
	JumpThreading	0	120,183	0
<i>Pipelines</i>	Default<O1>	734,771	195,297	0
	Default<O2>	734,265	195,825	0
	Default<O3>	734,265	195,825	0
	Default<Os>	734,265	195,825	0
	Default<Oz>	731,201	198,464	0
Total		6,368,144 (21.96%)	1,658,015 (5.72%)	0 (0%)

Table 7.2: Efficacy of tightened transformations evaluated with the cross product of 1,000,000 random LLVM programs and 29 transformations (24 passes and 5 default pipelines) for 29,000,000 total test cases. Successes, false positives and failures are computed as described in Figure 7.15.

The majority of false positives come from replacement transformations (InstCombine and GVN), which replace single uses of a value conservatively since other uses may prevent DCE from eliminating the update. Jump threading also has a high false positive rate, because it does not know the move destination at query time. As a result, the guard is overly conservative and could be safely weakened by re-engineering the pass.

RQ7.6.2: Can default pipelines reliably repair violations?

Result: In Section 7.4.6, I discussed the possibility for intermediate violations to be fixed by later transformations that normalize the code to eliminate violations. I find that default pipelines incidentally repair some violations, but not reliably enough to replace purpose-built tightening or repair.

Pipelines see fewer violations than individual passes (SimplifyCFG and SpeculativeExecution), dropping from $\sim 940,000$ to $\sim 730,000$, confirming the intuition. This reduction is explained by later elimination passes, particularly DCE, which happen to eliminate violations introduced by earlier passes, incidentally restoring

consumption-safety. For example, in the GVN example in Listing 7.2, subsequent optimizations may render `x1` dead. If that violating use is eliminated, then `t1` is consumption-safe. However, not all violations are repaired by happenstance, and relying on pipeline normalization alone is too fragile without ALMANAC or ERRATA.

RQ7.6.3: What is the performance cost of missed optimizations due to tightening?

Result: Tightening blocks some high-value optimizations, causing isolated performance regressions in ALMANAC, while ERRATA avoids these entirely.

To evaluate transformations blocked by ALMANAC, I evaluate fine-grained pass statistics (`clang -stats`) on the LLVM test suite, a collection of 318 C/C++ programs from well-established benchmark suites, shown in Table 7.3.

As expected, tightened guards immediately block code motion and replacement transformations. Blocking these enabling transformations cascades to SLP vectorization and instruction selection in the backend. For the majority of benchmarks, ALMANAC sees no significant regression from LLVM, but there are isolated cases on the right-hand side of Figure 7.17. Notably, `srad` saw a 9.57% regression from blocked LICM, while `halide_local_laplacian` and `pifft` had SLP vectorization regressions but paradoxically saw performance improvements (3.59% and 8.44%), likely due to poor vectorization cost models [118]. ERRATA did not block SLP vectorization on these benchmarks and introduces no statistically significant performance regression from LLVM.

RQ7.6.4: Can dedicated repair passes safely recover these missed opportunities?

Result: ERRATA repairs 100% of violations introduced by standard LLVM transformations, however it lacks the formal guarantees of ALMANAC.

ERRATA is evaluated using the same 1,000,000 LLVM programs and 29 trans-

<i>Class</i>	<i>Transform</i>	<i>Statistic</i>	Δ
Code Motion	SimplifyCFG	Simplifications made	-367
	InstCombine	Dead instructions removed	-96
	InstCombine	Instructions reassociated	-100
	Reassociate	Instructions transformed	-67
	LICM	Instructions hoisted	-1959
Replacement	GVN	Values replaced	-221
	PRE	Instructions eliminated	-29
	InstCombine	Instructions combined	-69
	InstCombine	Common sub-expressions eliminated	-4
Cascaded	SLP Vectorization	Vector instructions generated	-50
	Instruction Selection	SelectionDAG nodes combined	-24

Table 7.3: Pass statistics showing transformations blocked by tightened guards in ALMANAC on the LLVM test suite. As expected, transformations that may violate consumption-safety are immediately blocked. However, the cascading effects of these blocked transformations is only seen in a few cases for SLP vectorization and backend instruction selection.



Figure 7.17: Comparison of generated executable performance, shown as per-benchmark speedup relative to LLVM (log scale). All pipelines run the same optimization pipeline (-o3). The x axis of plot is sorted by ALMANAC speedup with ERRATA speedup for the same benchmark placed to its right. ALMANAC sees significant performance regressions on some benchmarks due to tightened guards leading to missed optimizations, while ERRATA does not see a statistically significant regression from LLVM.

formations from RQ7.6.1, with repair attempted on all programs that violated consumption-safety. An ablation study confirms that no single repair pass is sufficient, each is necessary. For example, Sink repairs all violations from SimplifyCFG but fails on other passes. Results are shown in Table 7.4.

This 100% success rate is not a coincidence, it spawns from the insight that guided repair to begin with (Section 7.5). Violations can only be introduced in our LLVM transforms by replacing a use of the consumed operand with a non-interfering equivalent (as in GVN), or by moving an instruction to a point where the consumed operand is live (as in code motion). In both cases, the structural information that justified the violation is precisely the information needed to repair it. Rethreading and GVN use the same non-interference analysis, and a moved instruction can be safely moved back.

However, this guarantee only holds if that structural information is preserved by the pipeline. With more complex non-interference checks such as range analysis [88], intervening passes may destroy repair-relevant information. ALMANAC

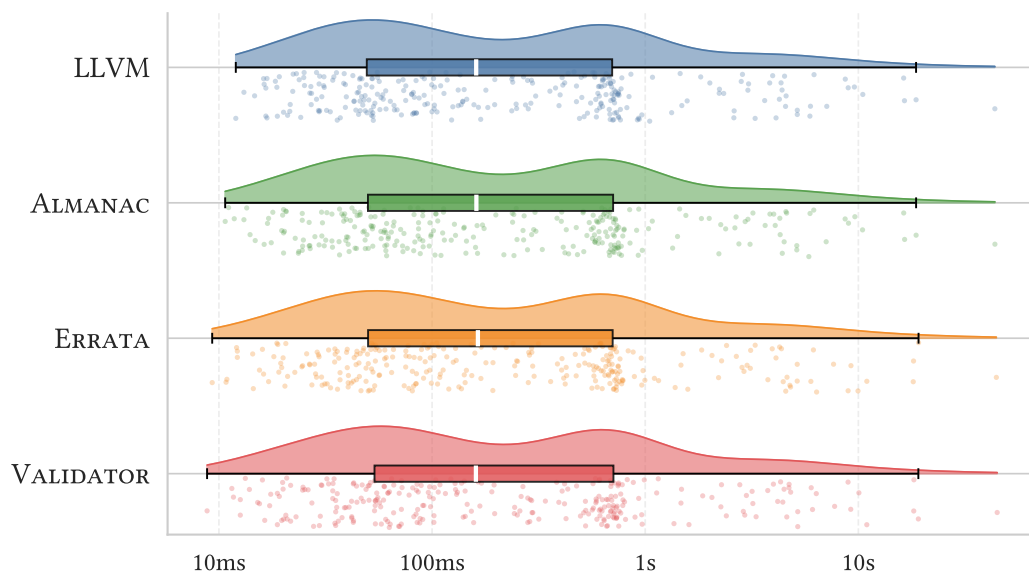


Figure 7.18: Raincloud diagrams comparing compilation time between the four compilation pipelines, evaluated on the LLVM test suite. All pipelines run the same optimization pipeline (-O3). Neither ALMANAC nor ERRATA have a significant impact on compilation time.

avoids this risk entirely, at the cost of occasionally blocking valid optimizations. Furthermore, a spurious copy likely imposes a far greater performance degradation than any missed optimization in Figure 7.17. The choice between ALMANAC and ERRATA therefore depends on whether repair-relevant information is robust to the pipeline.

RQ7.6.5: Do ALMANAC and ERRATA introduce reasonable compilation time overheads?

Result: ALMANAC and ERRATA both add work to the compilation pipeline. However, I find that neither introduce a statistically significant overhead in compilation time.

I evaluate baseline LLVM, ALMANAC, ERRATA, and the validator on the LLVM test suite under the same optimization pipeline (-O3). Results are presented in Figure 7.18.

<i>Transform</i>	<i>LLVM Violations</i>	<i>Individual</i>			<i>Ablation (pass disabled)</i>			<i>Full Errata</i>
		<i>Sink</i>	<i>Hoist</i>	<i>Rethread</i>	<i>Sink</i>	<i>Hoist</i>	<i>Rethread</i>	
Simplify CFG	946,263	100%	0%	0%	0%	100%	100%	100%
Spec. Exec.	946,305	100	0	100	100	100	100	100%
Inst Combine	793,799	2.27	100	100	100	100	100	100%
GVN	13,010	0	0	0.01	0.01	0.01	0	100%
Default<O1>	734,771	4.58	49.65	59.24	59.24	>99.99	87.60	100%
Default<O2>	734,265	4.60	49.64	59.21	59.21	>99.99	87.62	100%
Default<O3>	734,265	4.60	49.64	59.21	59.21	>99.99	87.62	100%
Default<Os>	734,265	4.60	49.64	59.21	59.21	>99.99	87.62	100%
Default<Oz>	731,201	4.70	50.50	59.04	59.04	>99.99	88.91	100%
Total	6,368,144	32.85	41.16	61.42	61.42	99.79	92.81	100%

Table 7.4: Breakdown of repair success by transformation, shown as percentage of violations successfully repaired (higher is better). The full ERRATA pipeline achieves a 100% success rate for all violations. I evaluate the efficacy of individual ERRATA transformations and ablate each repair transformation. No single repair transformation is sufficient to repair all violations. Furthermore, each repair transformation is necessary to achieve no violations in the full pipeline.

The tightening checks in ALMANAC are lightweight and add no measurable overhead. ERRATA passes operate on demand, performing small local transformations where needed, making their overhead negligible compared to global LLVM passes. This also suggests that a speculative pipeline built on ERRATA would incur overhead only from rollback. However, I expect rollback to be rare in practice, since ERRATA successfully repaired all violations in our evaluation.

7.7 Related Work

The contributions in this chapter build upon prior work from functional language compilers.

The aggregate update problem arises in functional languages, where compilers must recover destructive updates from an immutable, functional program. Instead, ALMANAC starts from a consumption-safe SSA program—guaranteed by construction from an imperative program—and must only preserve consumption-safety throughout compilation.

Linear and affine type systems guarantee consumption-safety by restricting programmers to an explicit, single-use type discipline. ALMANAC treats consumption safety as a property of the compiler IR itself, without restricting the input language.

Type-preserving compilation seeks to carry safety properties established in the source language through to generated code. ALMANAC follows this same goal, but for affine consumption-safety specifically, extending it through a broader class of compiler transformations than prior work has addressed.

7.7.1 The Aggregate Update Problem

The aggregate update problem was first identified by Hudak and Bloss [114], with subsequent work proposing static analyses to recover destructive updates [116], [117], [119] and dynamic approaches such as Perceus reference counting [120] in Koka [121], [122]. In this work, I focus on one-shot bufferization, since it is the most commonly used algorithm in modern compilers. My approach sidesteps both costs by starting from an imperative program, shifting the burden from recovering consumption-safety to preserving it.

7.7.2 Linearity, Uniqueness, and Lazy Linearity

Consumption-safety is closely related to affine typing, which is enforced by Linear Haskell [123] and Clean [124]. Affine types guarantee consumption-safety, since the variable may be used at most once. However, unlike affine typing, consumption safety is a property of the compiler IR rather than a programmer-facing type discipline. Linear types are more restrictive than affine types, requiring that variables be used exactly once. This enforces consumption-safety *and* ensures that all variables are explicitly dropped, avoiding memory leaks. I use affine types rather than linear types in this work because SSA optimization frequently creates dead paths requiring implicit drops. This restriction leads to the constrained results from Mesquita and Toninho [111], which prove that elimination and inlining are safe for linear typed programs. ALMANAC builds directly upon this result, extending them to the broader set of transformations in the context of consumption-safety.

7.7.3 Type-Preserving Compilation

Proof-carrying code [125] is an approach to guarantee the safety of generated code without placing the compiler in the trusted computing base. The idea behind this was to attach a machine-checkable safety proof to compiled code so a lightweight verifier, rather than the compiler, is the trusted component. Morrisett et al. [126]’s typed assembly language (TAL) provided a concrete instantiation of this idea, extending typing all the way to assembly so that a well-typed TAL program is provably type- and memory-safe. This inspired type-preserving compilation as a method for writing compilers that convert well-typed code into a typed assembly language [46], [127], [128]. While there is a bevy of work on type-preserving compilation, there is insufficient work on preserving linear types throughout aggressive compiler optimizations. Chen et al. [129] demonstrate type-preserving compilation of CIL extended with affine types, however this solely covers code generation, with no optimization. Mesquita and Toninho [111] explicitly set out to prove type preservation for a linear intermediate language, implemented in the Glasgow Haskell Compiler, but only cover a small subset of compiler transformations. This motivates the need for consumption-safety, as a simple framework for reasoning about affine type preservation. Consumption-safety takes a pragmatic approach, demonstrating how compiler transformations can be made to preserve affine types.

7.8 Discussion

In this chapter, I have shown that SSA construction, transformation, and destruction can be copy-free. This is not merely a theoretical result, but a practical guar-

antee that can be enforced in a production compiler with negligible overhead. For compiler engineers, the evaluation reveals clear design principles. If a system cannot tolerate a spurious copy under any circumstance, ALMANAC provides a strict static guarantee at the cost of occasional missed optimizations. If performance is paramount, ERRATA allows passes to run freely and repairs violations after the fact, achieving 100% effectiveness without the regressions seen in ALMANAC. As SSA compilers continue to expand into performance-critical data types and exclusive resources, this chapter provides the formal foundations and practical tools that compiler engineers need to build upon SSA form with confidence.

The formalization in this chapter targets the one-shot bufferization algorithm, which is intentionally simple compared to the sophisticated alternatives developed for functional language compilers [116], [117], [119]. Those approaches reason about arbitrary functional code to *recover* destructive updates, which is a fundamentally harder problem. By starting from an imperative program, consumption-safety is established by construction. Thanks to this, the approach solely need to reason about *preserving* consumption-safety to guarantee no spurious copies. The results of this chapter demonstrate how far this simple algorithm can be taken with the right formal foundations.

Our repair transformations rely on structural information preserved by the transformation pipeline. For data types with complex non-interference checks, such as range-based analyses [88], intervening passes may destroy this information before repair runs, reducing the effectiveness of ERRATA. In such cases, ALMANAC provides a stronger guarantee.

Finally, our building block decomposition covers the vast majority of common compiler transformations, but does not claim to be exhaustive. Novel transformations may require new building blocks or tightening guards beyond those derived

here, though the framework provides a clear methodology for doing so.

CHAPTER 8

Conclusion

The conventional wisdom in compiler design has long held that data collections are a source language abstraction, and that the compiler’s job is to lower them to memory as early as possible. The hash table becomes a flat array of buckets and the linked list becomes a chain of pointers. Once lowered, the compiler can proceed with analysis and optimization on the resulting memory representation.

The difficulty, as this thesis has argued, is that this approach discards the exact information needed for compiler analysis and optimization. A hash table is not merely an array of buckets, it is a key-value store with specific access patterns and update semantics. A linked list is not merely a chain of pointers, it is a sequence of values with $O(1)$ insertion and pointer-stable elements. By the time the compiler sees the lowered representation, all of that structure is gone, replaced by pointer arithmetic and aliasing. The compiler is left to recover high-level structure from low-level code, and it fails in even the most trivial cases.

MEMOIR takes a different approach. Rather than lowering collections immediately, it preserves them in the compiler IR throughout optimization. This approach is both feasible and productive. The optimizations described in this thesis—dead element elimination, field elision, automatic data enumeration, redundant induction elimination—are all consequences of having access to the logical structure of collections at compile time. In traditional compiler IRs, these same optimiza-

tions are impractical. Together, they demonstrate that data collections are not an obstacle to be lowered away, but an opportunity to be exploited.

The formal work in this thesis provides the foundation for doing that safely. Consumption-safety gives compiler developers a precise characterization of when SSA destruction is guaranteed to be copy-free, along with a framework for ensuring that transformations preserve it. This matters for MEMOIR, but it applies equally to any compiler that represents large or exclusive resources in SSA form, such as tensors in ML compilers, qubits in quantum compilers, and others being built on the same infrastructure. The result is a set of tools that compiler engineers can use to apply aggressive transformations with confidence, knowing that the lowering step will not silently introduce copies that cause catastrophic performance bugs.

Bookmarks address a tension between SSA value semantics and the references and iterators that appear throughout real programs. This tension turns out to not be fundamental, but rather an artifact of lacking the right abstraction. By representing references as immutable SSA values contextualized against collection versions, bookmarks give the compiler a way to reason about references statically, lower them to raw pointers with no runtime overhead, and detect invalidation bugs before they reach production. The same lineage functions that underpin bookmark semantics also enable the live element analysis and dead element elimination described earlier in the thesis, which suggests that the abstraction is well-founded and broadly applicable.

The tensor dialect in MLIR offers a concrete near-term opportunity for applying this work to a production compiler. It already represents tensor computations in SSA form, but lacks a principled way to express slices and views—references into sub-regions of a tensor that remain valid across updates. Bookmark seman-

tics are precisely the right abstraction for this, and the safety analyses developed here would give the tensor dialect a foundation for building both optimizations and verifiers.

Looking forward, the most promising direction is extensibility. The implementation specification language introduced for bookmarks is not merely a tool for describing standard library containers, it is also a mechanism for bringing custom collection semantics into the compiler. A data structure developer can write a specification for a skip list, a trie, or a custom graph representation, and immediately benefit from the analyses and optimizations built on top of MEMOIR. This is how compiler infrastructure should work: not as a closed system supporting a fixed set of types, but as an open platform that can be extended with new abstractions as demand evolves.

All of the work in this thesis is implemented in an open-source compiler built on LLVM. Proving that data collections can be preserved in a production-quality compiler, and evaluated against industry-standard benchmarks. The ideas presented here are grounded in that implementation, providing a firm foundation for a future of collection-centric compilation.

Bibliography

- [1] G. Ramalingam, “The undecidability of aliasing,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994, ISSN: 0164-0925. DOI: 10.1145/186025.186041. [Online]. Available: <https://doi.org/10.1145/186025.186041>.
- [2] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004. DOI: 10.1109/CGO.2004.1281665.
- [3] C. Lattner et al., “Mlir: Scaling compiler infrastructure for domain specific computation,” in *CGO*, 2021. DOI: 10.1109/CGO51591.2021.9370308.
- [4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991. DOI: 10.1145/115372.115320.
- [5] F. Rastello and F. B. Tichadou, Eds., *SSA-based Compiler Design*. Springer, 2022, ISBN: 978-3-030-80514-2.
- [6] D. Novillo, “Tree SSA: A new optimization infrastructure for GCC,” in *Proceedings of the 2003 GCC Developers Summit*, 2003. [Online]. Available: <https://www.airs.com/dnovillo/Papers/tree-ssa-gccs03.pdf>.
- [7] S. S. Ishtiaq and P. W. O’Hearn, “Bi as an assertion language for mutable data structures,” *SIGPLAN Notices*, vol. 36, no. 3, pp. 14–26, Jan. 2001, ISSN: 0362-1340. DOI: 10.1145/373243.375719. [Online]. Available: <https://doi.org/10.1145/373243.375719>.
- [8] F. E. Allen, “Control flow analysis,” *SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1970. DOI: 10.1145/390013.808479.

- [9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987, ISSN: 0164-0925. DOI: 10.1145/24039.24041. [Online]. Available: <https://doi.org/10.1145/24039.24041>.
- [10] G. A. Kildall, “A unified approach to global program optimization,” in *POPL*, 1973, pp. 194–206. DOI: 10.1145/512927.512945.
- [11] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks,” *Acta Inf.*, vol. 7, no. 3, pp. 305–317, Sep. 1977, ISSN: 0001-5903. DOI: 10.1007/BF00290339. [Online]. Available: <https://doi.org/10.1007/BF00290339>.
- [12] P. Cousot and R. Cousot, “Static determination of dynamic properties of generalized type unions,” *SIGSOFT Softw. Eng. Notes*, vol. 2, no. 2, pp. 77–94, Mar. 1977, ISSN: 0163-5948. DOI: 10.1145/390019.808314. [Online]. Available: <https://doi.org/10.1145/390019.808314>.
- [13] P. Cousot and R. Cousot, “Comparing the galois connection and widening/narrowing approaches to abstract interpretation,” in *Programming Language Implementation and Logic Programming*, M. Bruynooghe and M. Wirsing, Eds. Berlin/Heidelberg: Springer-Verlag, 1992, pp. 269–295. DOI: 10.1007/3-540-55844-6_142.
- [14] R. Leißa and J. M. Griebl, “Ssa without dominance for higher-order programs,” *Proc. ACM Program. Lang.*, vol. 10, no. PLDI, Jun. 2026. DOI: 10.1145/3808286.
- [15] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, New York, NY, USA: Association for Computing Machinery, 2007, pp. 89–100, ISBN: 9781595936332. DOI: 10.1145/1250734.1250746. [Online]. Available: <https://doi.org/10.1145/1250734.1250746>.

- [16] A. K. Sharp. “Memory layout transformations.”[Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html>.
- [17] L. Ye, M. Lis, and A. Fedorova, “A unifying abstraction for data structure splicing,” in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS ’19, 2019, pp. 173–183. DOI: 10.1145/3357526.3357548.
- [18] F. Chow, S. Chan, S. .- Liu, R. Lo, and M. Streich, “Effective representation of aliases and indirect memory operations in ssa form,” in *Compiler Construction*, T. Gyimóthy, Ed., 1996, pp. 253–267. DOI: 10.1007/3-540-61053-7_66.
- [19] D. Novillo et al., “Memory ssa-a unified approach for sparsely representing memory operations,” 2007.
- [20] Accessed: Jun. 30, 2026. [Online]. Available: <https://releases.llvm.org/3.9.0/docs/ReleaseNotes.html>.
- [21] Accessed: Jun. 30, 2026. [Online]. Available: <https://github.com/llvm/llvm-project/pull/152859>.
- [22] K. Knobe and V. Sarkar, “Array ssa form and its use in parallelization,” in *POPL*, 1998. [Online]. Available: <https://doi.org/10.1145/268946.268956>.
- [23] M. G. Burke et al., “The jalapeño dynamic optimizing compiler for java,” in *Proceedings of the ACM 1999 Conference on Java Grande*, ser. JAVA ’99, 1999, pp. 129–141. DOI: 10.1145/304065.304113.
- [24] R. Bodík, R. Gupta, and V. Sarkar, “Abcd: Eliminating array bounds checks on demand,” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI ’00, 2000. DOI: 10.1145/349299.349342.
- [25] S. J. Fink, K. Knobe, and V. Sarkar, “Unified analysis of array and object references in strongly typed languages,” in *Proceedings of the 7th International Symposium on Static Analysis*, Berlin, Heidelberg: Springer-Verlag, 2000, ISBN: 3540676686.

- [26] T. Ben-Nun, B. Ates, A. Calotoiu, and T. Hoefler, “Bridging control-centric and data-centric optimization,” in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO ’23, 2023. DOI: 10.1145/3579990.3580018.
- [27] A. J. Root, B. Yan, P. Liu, C. Gyurgyik, A. J. Bik, and F. Kjolstad, “Compilation of shape operators on sparse arrays,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. DOI: 10.1145/3689752. [Online]. Available: <https://doi.org/10.1145/3689752>.
- [28] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The tensor algebra compiler,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. DOI: 10.1145/3133901.
- [29] S. Smith and G. Karypis, “Tensor-matrix products with a compressed sparse tensor,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA³ ’15, 2015. DOI: 10.1145/2833179.2833183.
- [30] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, “Splatt: Efficient and parallel sparse tensor-matrix multiplication,” in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS ’15, 2015, pp. 61–70. DOI: 10.1109/IPDPS.2015.27.
- [31] W. Ahrens, D. Donenfeld, F. Kjolstad, and S. Amarasinghe, “Looplets: A language for structured coiteration,” in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO ’23, 2023, pp. 41–54. DOI: 10.1145/3579990.3580020.
- [32] W. Ahrens, T. F. Collin, R. Patel, K. Deeds, C. Hong, and S. Amarasinghe, “Finch: Sparse and structured tensor programming with control flow,” *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025. DOI: 10.1145/3720473.
- [33] J. Won, W. Ahrens, T. F. Collin, J. S. Emer, and S. Amarasinghe, “The continuous tensor abstraction: Where indices are real,” *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. DOI: 10.1145/3763146.

- [34] R. Patel, W. Ahrens, and S. Amarasinghe, “Systec: A symmetric sparse tensor compiler,” in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO ’25, 2025, pp. 47–62. DOI: 10.1145/3696443.3708919.
- [35] S. Kovach, P. Kolichala, T. Gu, and F. Kjolstad, “Indexed streams: A formal intermediate representation for fused contraction programs,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. DOI: 10.1145/3591268.
- [36] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, “Compiler support for sparse tensor computations in mlir,” *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, Sep. 2022, ISSN: 1544-3566. DOI: 10.1145/3544559.
- [37] K. Deeds, W. Ahrens, M. Balazinska, and D. Suci, “Galley: Modern query optimization for sparse tensor programs,” *Proc. ACM Manag. Data*, vol. 3, no. 3, Jun. 2025. DOI: 10.1145/3725301.
- [38] V. Gladshtein, Q. Zhao, W. Ahrens, S. Amarasinghe, and I. Sergey, “Mechanised hyper-safety proofs about structured data,” *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. DOI: 10.1145/3656403.
- [39] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee, “Til: A type-directed, optimizing compiler for ml,” *SIGPLAN Notices*, vol. 39, no. 4, pp. 554–567, Apr. 2004. DOI: 10.1145/989393.989449.
- [40] G. Morrisett, “Compiling with types,” Ph.D. dissertation, Carnegie Mellon University, 1995.
- [41] J. Chen, D. Wu, A. W. Appel, and H. Fang, “A provably sound tal for back-end optimization,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03, 2003, pp. 208–219. DOI: 10.1145/781131.781155.
- [42] V. S. Menon et al., “A verifiable ssa program representation for aggressive compiler optimization,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’06, 2006, pp. 397–408. DOI: 10.1145/1111037.1111072.

- [43] J. Chen and D. Tarditi, “A simple typed intermediate language for object-oriented languages,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’05, 2005, pp. 38–49. DOI: 10.1145/1040305.1040309.
- [44] F. Perry, C. Hawblitzel, and J. Chen, “Simple and flexible stack types,” in *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO 07)*, 2007.
- [45] Z. Shao, C. League, and S. Monnier, “Implementing typed intermediate languages,” in *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP 98)*, New York, NY, USA, 1998. DOI: 10.1145/289423.289460.
- [46] J. Chen et al., “Type-preserving compilation for large-scale optimizing object-oriented compilers,” *SIGPLAN Notices*, vol. 43, no. 6, pp. 183–192, Jun. 2008. DOI: 10.1145/1379022.1375604.
- [47] R. Leißa, M. Ullrich, J. Meyer, and S. Hack, “Mimir: An extensible and type-safe intermediate representation for the dsl age,” *Proc. ACM Program. Lang.*, vol. 9, no. POPL, Jan. 2025. DOI: 10.1145/3704840.
- [48] T. Gawlitza, J. Leroux, J. Reineke, H. Seidl, G. Sutre, and R. Wilhelm, “Polynomial precise interval analysis revisited,” in *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, S. Albers, H. Alt, and S. Näher, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 422–437, ISBN: 978-3-642-03456-5. DOI: 10.1007/978-3-642-03456-5_28.
- [49] W. Blume and R. Eigenmann, “Symbolic range propagation,” in *Proceedings of the 9th International Symposium on Parallel Processing*, ser. IPPS ’95, IEEE Computer Society, 1995, pp. 357–363, ISBN: 0818670746.
- [50] D. do Couto Teixeira and F. M. Q. Pereira, “The design and implementation of a non-iterative range analysis algorithm on a production compiler,” Federal University of Minas Gerais, Tech. Rep., 2011.

- [51] V. H. S. Campos, R. E. Rodrigues, I. R. de Assis Costa, and F. M. Q. Pereira, “Speed and precision in range analysis,” in *Proceedings of the 16th Brazilian Conference on Programming Languages*, ser. SBLP’12, Natal, Brazil, 2012, pp. 42–56. DOI: 10.1007/978-3-642-33182-4_5.
- [52] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Global value numbers and redundant computations,” in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’88, 1988, pp. 12–27. DOI: 10.1145/73560.73562.
- [53] T. M. Chilimbi, M. D. Hill, and J. R. Larus, “Cache-conscious structure layout,” in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI ’99, 1999, pp. 1–12. DOI: 10.1145/301618.301633.
- [54] A. Matni et al., “Noelle offers empowering llvm extensions,” in *CGO*, 2022. [Online]. Available: <https://doi.org/10.1109/CGO53902.2022.9741276>.
- [55] J. Bucek, K.-D. Lange, and J. v. Kistowski, “Spec cpu2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’18, 2018, pp. 41–42. DOI: 10.1145/3185768.3185771.
- [56] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, “Forma: A framework for safe automatic array reshaping,” *ACM Transactions on Programming Languages and Systems*, 2007. [Online]. Available: <https://doi.org/10.1145/1290520.1290522>.
- [57] C. Lattner and V. Adve, “Automatic pool allocation: Improving performance by controlling data structure layout in the heap,” in *PLDI*, 2005. [Online]. Available: <https://doi.org/10.1145/1065010.1065027>.
- [58] S. Curial et al., “Mpads: Memory-pooling-assisted data splitting,” in *Proceedings of the 7th international symposium on Memory management*, 2008. DOI: 10.1145/1375634.1375649.
- [59] J. Dolby and A. Chien, “An automatic object inlining optimization and its evaluation,” in *ACM SIGPLAN Notices*, no. 5, pp. 345–357, May 2000. DOI: 10.1145/358438.349344.

- [60] R. Veldema, C. J. H. Jacobs, R. F. H. Hofman, and H. E. Bal, “Object combining: A new aggressive optimization for object intensive programs,” en, *Concurrency and Computation: Practice and Experience*, vol. 17, no. 5–6, pp. 439–464, Apr. 2005. DOI: 10.1002/cpe.836.
- [61] R. Bruno, V. Jovanovic, C. Wimmer, and G. Alonso, “Compiler-assisted object inlining with value fields,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Association for Computing Machinery, 2021, pp. 128–141. DOI: 10.1145/3453483.3454034.
- [62] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, “Energy profiles of java collections classes,” 2016. [Online]. Available: <https://doi.org/10.1145/2884781.2884869>.
- [63] L. Liu and S. Rus, “Perflint: A context sensitive performance advisor for c++ programs,” in *CGO*, 2009. [Online]. Available: <https://doi.org/10.1109/CGO.2009.36>.
- [64] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, “Brainy: Effective selection of data structures,” in *PLDI*, 2011. [Online]. Available: <https://doi.org/10.1145/1993498.1993509>.
- [65] D. Costa and A. Andrzejak, “Collectionswitch: A framework for efficient and dynamic collection selection,” in *CGO*, 2018. [Online]. Available: <https://doi.org/10.1145/3168825>.
- [66] B. Franke, Z. Li, M. Morton, and M. Steuwer, “Collection skeletons: Declarative abstractions for data collections,” in *SLE*, 2022. [Online]. Available: <https://doi.org/10.1145/3567512.3567528>.
- [67] O. Shacham, M. Vechev, and E. Yahav, “Chameleon: Adaptive selection of collections,” *ACM SIGPLAN Notices*, 2009. [Online]. Available: <https://doi.org/10.1145/1543135.1542522>.
- [68] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, “Darwinian data structure selection,” in *ESEC/FSE*, 2018. [Online]. Available: <https://doi.org/10.1145/3236024.3236043>.

- [69] N. Couderc, E. Söderberg, and C. Reichenbach, “Jbrainy: Micro-benchmarking java collections with interference,” in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE, 2020. [Online]. Available: <https://doi.org/10.1145/3375555.3383760>.
- [70] C. Loncaric, E. Torlak, and M. D. Ernst, “Fast synthesis of fast collections,” in *PLDI*, 2016. [Online]. Available: <https://dl.acm.org/doi/10.1145/2908080.2908122>.
- [71] G. Xu, “Coco: Sound and adaptive replacement of java collections,” in *ECOOP 2013*, 2013.
- [72] C. Wang, P. Yao, W. Tang, Q. Shi, and C. Zhang, “Complexity-guided container replacement synthesis,” 2022. [Online]. Available: <https://doi.org/10.1145/3527312>.
- [73] G. L. Steele, *Common LISP: The Language* (HP Technologies), eng, 2nd ed. Saint Louis: Elsevier Science, 1984, ISBN: 9780080502267.
- [74] C. Hanson and M. S. Team, *MIT/GNU Scheme Reference Manual*. London, GBR: Samurai Media Limited, 2015, ISBN: 9789888381579.
- [75] “Java development kit version 19 api specification.” Accessed: May 14, 2025. [Online]. Available: <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/String.html>.
- [76] “Lua-users wiki: Immutable objects.” Accessed: May 14, 2025. [Online]. Available: <http://lua-users.org/wiki/ImmutableObjects>.
- [77] “Clojure - special forms.” Accessed: May 14, 2025. [Online]. Available: https://clojure.org/reference/special_forms.
- [78] “Chapter 5. conversions and promotions.” Accessed: May 14, 2025. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html>.
- [79] “New unique package - the go programming language.” Accessed: May 14, 2025. [Online]. Available: <https://go.dev/blog/unique>.

- [80] P. R. Calder and M. A. Linton, “Glyphs: Flyweight objects for user interfaces,” in *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1990. [Online]. Available: <https://doi.org/10.1145/97924.97935>.
- [81] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, “Consistently faster and smaller compressed bitmaps with roaring,” *Software: Practice and Experience*, vol. 46, no. 11, pp. 1547–1569, Apr. 2016, ISSN: 1097-024X. [Online]. Available: <https://doi.org/10.1002/spe.2402>.
- [82] [Online]. Available: <https://lucene.apache.org/index.html>.
- [83] B. Chattopadhyay et al., “Procella: Unifying serving and analytical data at youtube,” *Proceedings of the VLDB Endowment*, 2019. [Online]. Available: <https://doi.org/10.14778/3352063.3352121>.
- [84] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, “An experimental study of bitmap compression vs. inverted list compression,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago Illinois USA, 2017. [Online]. Available: <https://doi.org/10.1145/3035918.3064007>.
- [85] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *PLDI*, 2013. [Online]. Available: <https://doi.org/10.1145/2491956.2462176>.
- [86] T. Chen et al., “Tvm: An automated end-to-end optimizing compiler for deep learning,” in *OSDI*, 2018. [Online]. Available: <https://doi.org/10.5555/3291168.3291211>.
- [87] M. P. Lücke, O. Zinenko, W. S. Moses, M. Steuwer, and A. Cohen, “The mlir transform dialect: Your compiler is more powerful than you think,” in *CGO*, 2025. [Online]. Available: <https://doi.org/10.1145/3696443.3708922>.
- [88] T. McMichen, N. Greiner, P. Zhong, F. Sossai, A. Patel, and S. Campanoni, “Representing data collections in an ssa form,” in *CGO*, 2024. DOI: 10.1109/CGO57630.2024.10444817.

- [89] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, “Lonestar: A suite of parallel irregular programs,” in *ISPASS*, 2009. [Online]. Available: <http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf>.
- [90] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *PACT*, 2008. [Online]. Available: <https://doi.org/10.1145/1454115.1454128>.
- [91] J. Leskovec and R. Sosič, “Snap: A general-purpose network analysis and graph-mining library,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, Jul. 2016. DOI: 10.1145/2898361.
- [92] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, <http://snap.stanford.edu/data>, Jun. 2014.
- [93] Abseil. “An open-source collection of core c++ library code.” [Online]. Available: <https://abseil.io/>.
- [94] Y. Smaragdakis and D. Batory, “DiSTiL: A transformation library for data structures,” in *Conference on Domain-Specific Languages (DSL 97)*, 1997. [Online]. Available: <https://www.usenix.org/conference/dsl-97/distil-transformation-library-data-structures>.
- [95] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv, “Data representation synthesis,” 2011. [Online]. Available: <https://doi.org/10.1145/1993316.1993504>.
- [96] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv, “Concurrent data representation synthesis,” in *PLDI*, 2012. [Online]. Available: <https://doi.org/10.1145/2254064.2254114>.
- [97] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, and G. Tagliavini, “Tiny pointers,” in *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms*, 2024. DOI: 10.1137/1.9781611977554.ch21.

- [98] N. Wanninger, T. McMichen, S. Campanoni, and P. Dinda, “Getting a handle on unmanaged memory,” in *ASPLOS*, 2024. DOI: 10.1145/3620666.3651326. [Online]. Available: <https://doi.org/10.1145/3620666.3651326>.
- [99] C. Lattner and V. S. Adve, “Transparent pointer compression for linked data structures,” in *Proceedings of the 2005 Workshop on Memory System Performance*, 2005. DOI: 10.1145/1111583.1111587.
- [100] J. Shirako and V. Sarkar, “An affine scheduling framework for integrating data layout and loop transformations,” in *LCPC*, ser. Lecture Notes in Computer Science, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-95953-1_1.
- [101] N. Vasilache et al., “Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction,” 2022. DOI: 10.48550/arXiv.2202.03293.
- [102] M. Fehr, J. Niu, R. Riddle, M. Amini, Z. Su, and T. Grosser, “Irdl: An ir definition language for ssa compilers,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022. DOI: 10.1145/3519939.3523700.
- [103] M. Fehr et al., “Xdsl: Sidekick compilation for ssa-based compilers,” in *CGO*, 2025. DOI: 10.1145/3696443.3708945.
- [104] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL ’77, 1977, pp. 238–252. DOI: 10.1145/512950.512973.
- [105] P. Cousot and R. Cousot, “Static determination of dynamic properties of generalized type unions,” in *Proceedings of an ACM Conference on Language Design for Reliable Software*, 1977, pp. 77–94. DOI: 10.1145/800022.808314.

- [106] D. Grove, G. DeFouw, J. Dean, and C. Chambers, “Call graph construction in object-oriented languages,” in *OOPSLA*, ser. OOPSLA ’97, 1997, pp. 108–124. DOI: 10.1145/263698.264352.
- [107] T. Johnson, M. Amini, and X. D. Li, “Thinlto: Scalable and incremental lto,” in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO ’17, 2017, pp. 111–121. DOI: 10.5555/3049832.3049845.
- [108] A. McCaskey and T. Nguyen, *A mlir dialect for quantum assembly languages*, 2021. DOI: 10.48550/arXiv.2101.11365. arXiv: 2101.11365 [quant-ph].
- [109] A. Peduri, S. Bhat, and T. Grosser, “Qssa: An ssa-based ir for quantum computing,” in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022, 2022, pp. 2–14. DOI: 10.1145/3497776.3517772.
- [110] F. E. Allen and J. Cocke, “A catalogue of optimizing transformations,” 1971.
- [111] R. Mesquita and B. Toninho, “Lazy linearity for a core functional language,” in *POPL*, 2026. DOI: 10.1145/3776711.
- [112] R. Mesquita and B. Toninho, *Lazy Linearity for a Core Functional Language*, 2025. DOI: 10.48550/ARXIV.2511.10361. Accessed: Mar. 10, 2026.
- [113] D. L. Whitfield and M. L. Soffa, “An approach for exploring code improving transformations,” *ACM TOPLAS*, vol. 19, no. 6, pp. 1053–1084, Nov. 1997, ISSN: 0164-0925. DOI: 10.1145/267959.267960.
- [114] P. Hudak and A. Bloss, “The aggregate update problem in functional programming systems,” in *POPL*, 1985. DOI: 10.1145/318593.318660.
- [115] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, pp. 358–366, 1953. DOI: 10.1090/S0002-9947-1953-0053041-6.

- [116] A. V. S. Sastry, “Efficient array update analysis of strict functional languages,” Ph.D. dissertation, University of Oregon, USA, 1994. DOI: 10.5555/221818.
- [117] N. Shankar, “Static analysis for safe destructive updates in a functional language,” in *Logic Based Program Synthesis and Transformation*, 2002. DOI: 10.1007/3-540-45607-4_1.
- [118] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, “Neurovectorizer: End-to-end vectorization with deep reinforcement learning,” in *CGO*, 2020, pp. 242–255. DOI: 10.1145/3368826.3377928.
- [119] C. Dimoulas and M. Wand, “The higher-order aggregate update problem,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2008. DOI: 10.1007/978-3-540-93900-9_8.
- [120] A. Reinking, N. Xie, L. De Moura, and D. Leijen, “Perceus: Garbage free reference counting with reuse,” in *PLDI*, 2021. DOI: 10.1145/3453483.3454032.
- [121] D. Leijen, “Koka: Programming with row polymorphic effect types,” *Electronic Proceedings in Theoretical Computer Science*, vol. 153, pp. 100–126, Jun. 2014, ISSN: 2075-2180. DOI: 10.4204/eptcs.153.8. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.153.8>.
- [122] D. Leijen, “Type directed compilation of row-typed algebraic effects,” in *POPL*, Jan. 2017, pp. 486–499. DOI: 10.1145/3009837.3009872.
- [123] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, “Linear haskell: Practical linearity in a higher-order polymorphic language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158093.
- [124] M. Plasmeijer, “Clean: A programming environment based on term graph rewriting,” *Electronic Notes in Theoretical Computer Science*, vol. 2, pp. 215–221, 1995, SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80199-9.

- [125] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 97)*, Paris, France, 1997. doi: 10.1145/263699.263712.
- [126] G. Morrisett, D. Walker, K. Crary, and N. Glew, “From system f to typed assembly language,” *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, pp. 527–568, May 1999. doi: 10.1145/319301.319345.
- [127] C. League, Z. Shao, and V. Trifonov, “Type-preserving compilation of featherweight java,” *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 2, pp. 112–152, Mar. 2002. doi: 10.1145/514952.514954.
- [128] A. Chlipala, “A certified type-preserving compiler from lambda calculus to assembly language,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, 2007, pp. 54–65. doi: 10.1145/1250734.1250742.
- [129] J. Chen, R. Chugh, and N. Swamy, “Type-preserving compilation of end-to-end verification of security enforcement,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10, 2010, pp. 412–423. doi: 10.1145/1806596.1806643.

APPENDIX A

Supplemental Proofs for SSA Destruction with No Spurious Copies

A.1 Operational Semantics

A.1.1 Semantic Domain

Our semantics are defined on the semantic domain in Figure A.1. I abstract the underlying data type by representing values as a history of updates.

$l \in Loc$	(memory location)
$n \in \mathbb{N}$	(abstract update)
$H \in History ::= \epsilon \mid H \cdot n$	(sequence of updates)
$v \in Val ::= l \mid n$	(value)
$E \in Env = Var \rightarrow Loc$	(environment)
$S \in Store = Loc \rightarrow History$	(store)

Figure A.1: Semantic domain of CORE, SSA and IMP.

A.1.2 Instruction Evaluation

The rules for instruction evaluation of CORE are defined in Figure A.2.

$$\begin{array}{c}
\frac{l \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{alloc}} \rangle \rightarrow \langle S[l \mapsto \epsilon]; l \rangle} \text{E-ALLOC} \\
\\
\frac{E(x) = l \quad S(l) = H \quad l' \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{copy } x} \rangle \rightarrow \langle S[l' \mapsto H]; l' \rangle} \text{E-COPY} \\
\\
\frac{E(x) = l \quad S(l) = H}{\langle E; S; \boxed{\text{read } x} \rangle \rightarrow \langle S; \text{observe}(H) \rangle} \text{E-READ}
\end{array}$$

Figure A.2: Instruction Evaluation of CORE.

The semantics for destructive updates in IMP are provided in Figure A.3. Notice that the resulting location l is the same location as the operand, meaning the update is performed in-place. The store S is destructively updated to append the new operation n to the history H .

$$\frac{E(x) = l \quad S(l) = H}{\langle E; S; \boxed{\text{update! } x \ n} \rangle \rightarrow \langle S[l \mapsto H \cdot n]; l \rangle} \text{E-UPDATE-IMP}$$

Figure A.3: Instruction Evaluation of IMP.

The semantics for functional updates in SSA are provided in Figure A.4. Notice that the resulting location l' is fresh, meaning that the update yields a new value. The store S is solely extended, with the new operation n appended to a new history.

$$\frac{E(x) = l \quad S(l) = H \quad l' \notin \text{Domain}(S)}{\langle E; S; \boxed{\text{update } x \ n} \rangle \rightarrow \langle S[l' \mapsto H \cdot n]; l' \rangle} \text{E-UPDATE-SSA}$$

Figure A.4: Instruction Evaluation of SSA.

A.1.3 Small-Step Operational Semantics

The operational semantics of expressions in CORE are defined as small-step semantics in Figure A.5. The E-LET uses the instruction evaluation rules define in

Section A.1.2. We write $E[\overline{x}_{new}/\overline{x}_{old}]$ for variable substitution in the environment domain.

$$\begin{array}{c}
\frac{\langle E; S; i \rangle \rightarrow \langle S'; v \rangle}{\langle E; S; \boxed{\text{let } x = i \text{ in } e} \rangle \longrightarrow \langle E[x \mapsto v]; S'; e \rangle} \text{E-LET} \\
\\
\frac{p(\ell) = \boxed{\text{block } \ell (\overline{x}_p) e}}{\langle E; S; \boxed{\text{goto } \ell (\overline{x}_a)} \rangle \longrightarrow \langle E[\overline{x}_p/\overline{x}_a]; S; e \rangle} \text{E-GOTO} \\
\\
\frac{E(x) = 0 \quad p(\ell_0) = \boxed{\text{block } \ell_0 (\overline{x}_p) e}}{\langle E; S; \boxed{\text{if0 } x \ell_0 (\overline{x}_0 \ell_1 (\overline{x}_1))} \rangle \longrightarrow \langle E[\overline{x}_p/\overline{x}_a]; S; e \rangle} \text{E-IF0-THEN} \\
\\
\frac{E(x) \neq 0 \quad p(\ell_0) = \boxed{\text{block } \ell_1 (\overline{x}_p) e}}{\langle E; S; \boxed{\text{if0 } x \ell_0 (\overline{x}_0 \ell_1 (\overline{x}_1))} \rangle \longrightarrow \langle E[\overline{x}_p/\overline{x}_a]; S; e \rangle} \text{E-IF0-ELSE}
\end{array}$$

Figure A.5: Small-step operation semantics of CORE.

A.2 Formal Definition of SSA Destruction

$$\begin{array}{c}
 \boxed{\text{Expr}_{\text{SSA}} \xrightarrow{\text{IMP}} \text{Expr}_{\text{IMP}}} \\
 \\
 \frac{x \notin FV(e) \quad e \xrightarrow{\text{IMP}} e'}{\boxed{\text{let } y = \text{update } x \text{ n in } e} \xrightarrow{\text{IMP}} \boxed{\text{let } y = \text{update! } x \text{ n in } e'}} \quad \text{D-INPLACE} \\
 \\
 \frac{x \in FV(e) \quad e \xrightarrow{\text{IMP}} e' \quad z \text{ is fresh}}{\boxed{\text{let } y = \text{update } x \text{ n in } e} \xrightarrow{\text{IMP}} \boxed{\begin{array}{l} \text{let } z = \text{copy } x \text{ in} \\ \text{let } y = \text{update! } z \text{ n in } e' \end{array}}} \quad \text{D-COPY} \\
 \\
 \frac{Op(i) \neq \text{update} \quad e \xrightarrow{\text{IMP}} e'}{\boxed{\text{let } y = i \text{ in } e} \xrightarrow{\text{IMP}} \boxed{\text{let } y = i \text{ in } e'}} \quad \text{D-LET} \quad \frac{}{\boxed{t} \xrightarrow{\text{IMP}} \boxed{t}} \quad \text{D-TERM}
 \end{array}$$

Figure A.6: SSA Destruction relation.

A.3 Formal Definition of SSA Construction

$$\begin{array}{c}
 \boxed{\text{Expr}_{\text{IMP}} \xrightarrow{\text{SSA}} \text{Expr}_{\text{SSA}}} \\
 \\
 \frac{e \xrightarrow{\text{SSA}} e'}{\boxed{\text{let } y = \text{update! } x \text{ v in } e} \xrightarrow{\text{SSA}} \boxed{\text{let } y = \text{update } x \text{ v in } e'[y/x]}} \quad \text{C-Update} \\
 \\
 \frac{Op(i) \neq \text{update!} \quad e \xrightarrow{\text{SSA}} e'}{\boxed{\text{let } x = i \text{ in } e} \xrightarrow{\text{SSA}} \boxed{\text{let } x = i \text{ in } e'}} \quad \text{C-Let}
 \end{array}$$

Figure A.7: SSA Construction Relation ($\xrightarrow{\text{SSA}}$).

A.4 Elimination Preserves Consumption-Safety

Lemma A.4.1 (Consumption-safety of Continuation). *If $\boxed{\text{let } x = i \text{ in } e_\kappa}$ ok, then e_κ ok.*

Proof. By inversion of S-UPDATE and S-LET, both have e_κ ok as a premise. $\square$

Lemma A.4.2 (T-ELIM Preserves Consumption-safety). *Given a consumption-safe SSA expression e the transformation $e \Rightarrow_{\text{T-ELIM}} e'$ preserves consumption-safety.*

Proof. Let $e = \boxed{\text{let } x = i \text{ in } e_\kappa}$. By Lemma A.4.1, e_κ ok. Therefore, e' ok. By T-ELIM, $e' = e_\kappa$. $\square$

A.5 Inlining Preserves Consumption-Safety

Lemma A.5.1 (T-INLINE Preserves Consumption-safety). *Given a consumption-safe SSA expression e the transformation $e \Rightarrow_{\text{T-INLINE}} e'$ preserves consumption-safety.*

Proof. Let $e = \boxed{\text{jump } \ell (\bar{x}_j)}$, where $p(\ell) = \boxed{\text{block } \ell (\bar{x}_\ell) e_\ell}$. Since blocks are closed, only the block parameters can appear free in the subexpression e_ℓ , i.e., $FV(e_\ell) \subseteq \{\bar{x}_\ell\}$. Since $e' = e_\ell[\bar{x}_j/\bar{x}_\ell]$, we know that $FV(e') \subseteq \{\bar{x}_j\}$. The only free variables in e are those passed as arguments to the jump expression, i.e., $FV(e) = \{\bar{x}_j\}$. Therefore, $FV(e') \subseteq FV(e)$. Since the outer binding context $B[e] \in p$ is consumption safe with $FV(e)$, $B[e']$ is consumption safe with a subset of $FV(e)$. $\square$

A.6 Block Injection Preserves Consumption-Safety

Lemma A.6.1 (T-INJ-BLOCK Preserves Consumption-safety). *Given a consumption-safe SSA program p , the transformation $p \Rightarrow_{\text{T-INJ-BLOCK}} p'$ preserves consumption-safety.*

Proof Sketch. Since blocks are closed and all variable names are globally unique, the injected block's free variables are disjoint from $Consumed(B)$ for any binding context B in the program. Therefore injection cannot introduce a variable that is consumed in an outer context. $\square$

A.7 Let Injection Requires Tightening

Lemma A.7.1 (TT-INJ-* Preserve Consumption-safety). *Given a consumption-safe SSA expression e , the transformations $e \Rightarrow_{\text{TT-INJ-LET}} e'$ and $e \Rightarrow_{\text{TT-INJ-UPD}} e'$ both preserve consumption-safety.*

Proof. We will prove this for both TT-INJ-LET and TT-INJ-UPD simultaneously, handling differences between the rules in a final case at the end.

Let $e = B[e_\kappa]$, $e'_\kappa = \boxed{\text{let } x = i \text{ in } e_\kappa}$ and $e' = B[e'_\kappa]$. By Lemma A.4.1, e_κ ok.

For $B[e'_\kappa]$ to be consumption-safe, we must show that none of the consumed variables in B are used in e'_κ , i.e., $Consumed(B) \cap FV(e'_\kappa) = \emptyset$ where $FV(e'_\kappa) = FV(i) \cup FV(e_\kappa)$. By inversion of S-UPDATE, $Consumed(B) \cap FV(e_\kappa) = \emptyset$. Therefore, it suffices to show that $Consumed(B) \cap FV(i) = \emptyset$, which is satisfied by inversion of TT-INJ-LET.

It remains to show that e'_κ is well-formed. We proceed by cases on i :

- **Case $i = \text{update } x_0 \text{ n}$:** It remains to show that $\boxed{\text{let } x = \text{update } x_0 \text{ n in } e_\kappa}$ is consumption safe. By S-UPDATE, it suffices to show $x_0 \notin FV(e_\kappa)$. By inversion of TT-INJ-UPD, $x_0 \notin FV(e_\kappa)$. Hence e'_κ ok by S-UPDATE.
- **Case otherwise:** Since $Op(i) \neq \text{update}$ and e_κ ok, therefore e'_κ ok by S-LET.

Since $e' = B[e'_\kappa]$ and B is unchanged, the consumption-safety of B follows from the consumption-safety of e by inversion. Therefore, in both cases e' ok. $\square$

A.8 Code Motion Requires Tightening

Lemma A.8.1 (TT-COMM-LET-* Preserves Consumption-safety). *Given a consumption safe SSA expression $B[e]$, the transformations*

$$B[e] \Rightarrow_{\text{TT-COMM-LET-LET}} B[e'] \text{ and } B[e] \Rightarrow_{\text{TT-COMM-LET-UPD}} B[e']$$

both preserve consumption-safety.

Proof. First, we prove that e' , the expression in the hole, is consumption-safe. We will start our proof at the innermost continuation and move out. Let $e = \boxed{\text{let } x_1 = i_1 \text{ in } e_2}$, where $e_2 = \boxed{\text{let } x_2 = i_2 \text{ in } e_\kappa}$. Let $e' = \boxed{\text{let } x_2 = i_2 \text{ in } e'_1}$, where $e'_1 = \boxed{\text{let } x_1 = i_1 \text{ in } e_\kappa}$.

We prove this for e'_1 by contradiction. Assume that e'_1 is not consumption safe. In all cases of consumption-safety, this implies that the continuation is not safe, i.e., $\neg e_\kappa \text{ ok}$. However, by Lemma A.4.1 on e_2 , we know that $e_\kappa \text{ ok}$. Therefore, we must prove that the other premises of either S-UPDATE or S-LET do not hold. We proceed by cases on i_1 :

- **Case $i_1 = \text{update } x_0 \text{ n}$:** In this case, S-LET does not apply, so S-UPDATE must not hold. This implies that the consumed variable is used in the continuation, i.e., $x_0 \in FV(e_\kappa)$. By inversion of S-UPDATE on e , we know that $x_0 \notin FV(e_2)$, where $FV(e_2) = FV(i_2) \cup FV(e_\kappa)$, which implies that $x_0 \notin FV(e_\kappa)$. This is a contradiction.
- **Case otherwise:** In this case, S-UPDATE does not apply, so S-LET must not hold. This implies that $Op(i_1) = \text{update}$. However, in this case $Op(i_1) \neq \text{update}$. This is a contradiction.

The assumption that e'_1 is not consumption safe contradicts the consumption-safety of the input expression e , therefore e'_1 ok.

We then prove this for e' . We proceed by cases on i_2 :

- **Case $i_2 = \mathbf{update} \ x_0 \ n$:** To do so, we must show that the variable consumed by i_2 is not used in its continuation, i.e., $x_0 \notin FV(e'_1)$ where $FV(e'_1) = FV(i_1) \cup FV(e_\kappa)$. By inversion of S-UPDATE on e_2 , we know that $x_0 \notin FV(e_\kappa)$. Therefore, it suffices to show that $x_0 \notin FV(i_1)$, which is satisfied by inversion of TT-COMM-LET-UPD.
- **Case otherwise:** The continuation must be consumption-safe, which we already proved earlier.

Therefore, e' ok.

Finally, we need to ensure that the consumption-safety of updates in the outer binding context B are maintained by the transformation. By inversion of S-UPDATE on e , $Consumed(B) \cap FV(e) = \emptyset$. Since we only reorder let bindings, the free variables in the hole do not change, i.e., $FV(e) = FV(e')$. Therefore, $Consumed(B) \cap FV(e') = \emptyset$. Since $B[e]$ ok and e' ok, then $B[e']$ ok. $\square$

A.9 Replacement Requires Tightening

Lemma A.9.1 (TT-REPLACE Preserves Consumption-safety). *Given a consumption-safe SSA expression e , the transformation $e \Rightarrow_{\text{TT-REPLACE}} e'$ is consumption-safety preserving.*

Proof. Let $e = B[\mathbf{let} \ x = i \ \mathbf{in} \ e_\kappa]$, $e' = B[\mathbf{let} \ x = i' \ \mathbf{in} \ e_\kappa]$, and $i' = i[x_{\text{new}}/x_{\text{old}}]$.

If $x_{\text{new}} = x_{\text{old}}$, then $i' = i$ and $e' = e$. e ok holds by assumption, hence e' ok.

By inversion of S-LET, e_κ ok.

For $B[\mathbf{let\ x = i' \ in\ } e_k]$ to be well-formed, we must show that none of the consumed variables in B are used in $\mathbf{let\ x = i' \ in\ } e_k$. That is:

$$Consumed(B) \cap FV(\mathbf{let\ x = i' \ in\ } e_k) = \emptyset$$

$FV(\mathbf{let\ x = i' \ in\ } e_k) = FV(i') \cup FV(e_k)$. By inversion of S-UPDATE, $Consumed(B) \cap FV(e_k) = \emptyset$. Therefore, it suffices to show that $Consumed(B) \cap FV(i') = \emptyset$. Since $i' = i[x_{new}/x_{old}]$, $FV(i') \subseteq FV(i) \cup \{x_{new}\}$. By inversion of S-UPDATE, $Consumed(B) \cap FV(i) = \emptyset$. Therefore, it suffices to show that $x_{new} \notin Consumed(B)$, which is given by inversion of TT-REPLACE. $\square$

A.10 SSA Destruction of a Consumption-Safe Program

Guarantees No Spurious Copies

Lemma A.10.1 (Consumption-safety Guarantees Destruction with No Spurious Copies). *Given a consumption-safe SSA expression e , the destruction $e \xrightarrow{\text{IMP}} e'$ will not evaluate the D-COPY rule, resulting in an IMP expression e' containing zero spurious copies.*

Proof. The proof proceeds by structural induction on the destruction relation. For any consumption-safe subexpression e' of e , destruction of e' introduces no spurious copies. The base cases are the terminators, which are reconstructed directly by D-TERM.

Because non-update let-bindings strictly recurse on their subexpressions using D-LET, by the inductive hypothesis the subexpression introduces no spurious copies.

The only critical case is the evaluation of the **update** instruction.

Let $e = \boxed{\text{let } x_1 = \text{update } x_0 \text{ n in } e_\kappa}$.

The destruction algorithm must choose between D-INPLACE and D-COPY for this expression.

The premise for D-COPY strictly requires that the object operand is live in the continuation, i.e., $x_0 \in FV(e_\kappa)$. However, by inversion of S-UPDATE on e , we know that $x_0 \notin FV(e_\kappa)$. Therefore, D-COPY is unreachable.

Conversely, the premise for D-INPLACE requires $x_0 \in FV(e_\kappa)$, which is satisfied. Therefore, the destruction relation is forced to evaluate D-INPLACE for all **update** instructions in e , safely emitting the in-place, destructive update. Copies in the original SSA expression are propagated by D-LET and are not spurious by definition, so the only way spurious copies could be introduced is by D-COPY, which is unreachable. Thus, no spurious copy instructions are introduced into the output IMP expression by destruction of a functional **update**. $\square$

A.11 SSA Construction yields a Consumption-Safe Program

Lemma A.11.1 (SSA Constructed Expressions are Consumption-Safe). *Given an IMP expression e , the constructed SSA expression e' is consumption safe.*

Proof. The proof proceeds by structural induction on the construction relation $(\xrightarrow{\text{SSA}})$. For any IMP subexpression e_κ , if e_κ ok then e'_κ ok. The base cases are the terminators, which contain no update instructions and are trivially consumption-safe.

Given a non-update let-binding $e = \boxed{\text{let } x = i \text{ in } e_\kappa}$, where $Op(i) \neq \text{update!}$, construction by C-LET preserves the instruction i and recurses on the continuation

$e_k \xrightarrow{\text{SSA}} e'_k$. Since no **update** is introduced into e' by C-LET and by the inductive hypothesis e'_k ok, then e' ok holds.

The only critical case is the evaluation of the **update!** instruction.

Let $e = \boxed{\text{let } x_1 = \text{update! } x_0 \text{ n in } e_k}$.

By C-UPDATE, $e' = \boxed{\text{let } x_1 = \text{update } x_0 \text{ n in } e'_k}$, where $e'_k = e_k[x_1/x_0]$.

By the standard definition of capture-avoiding substitution, the free variables of the substituted continuation are $FV(e'_k) = FV(e_k[x_1/x_0]) = (FV(e_k) \setminus \{x_0\}) \cup \{x_1\}$. Therefore, $x_0 \notin FV(e'_k)$.

By the induction hypothesis on e_k , we know that e'_k ok.

Since, $x_0 \notin FV(e'_k)$ and e'_k ok, e' ok by S-UPDATE. $\square$

A.12 SSA Construction and Destruction are Semantics Preserving

A.12.1 State Equivalence

While Lemma 7.2.1 guarantees that our destruction algorithm achieves the structural goal of introducing zero spurious copies, we must also prove that translating a functional update into an in-place mutation preserves the runtime semantics of the program. We demonstrate this semantic soundness via a forward simulation. To do so, we use state equivalence to show that an IMP state and an SSA state are equivalent.

Definition A.12.1 (State Equivalence). *Let ρ be a mapping between IMP variables and SSA variables. An IMP state $\langle E_{\text{IMP}}; S_{\text{IMP}} \rangle$ and an SSA state $\langle E_{\text{SSA}}; S_{\text{SSA}} \rangle$ are equivalent*

under ρ , written $\langle E_{IMP}; S_{IMP} \rangle \approx \langle E_{SSA}; S_{SSA} \rangle$, if and only if for all $x \in \text{Domain}(E_{IMP})$:

$$S_{IMP}(E_{IMP}(x)) = S_{SSA}(E_{SSA}(\rho(x)))$$

A.12.2 Copy-Free SSA Destruction Preserves Semantics

While the full destruction relation ($\xrightarrow{IMP}$) naturally preserves state equivalence for any well-formed SSA expression by falling back on the D-COPY rule, this introduces spurious copies. We prove that for programs satisfying our syntactic check, this fallback is mathematically unreachable, allowing optimal in-place mutation while preserving semantics.

Theorem A.12.1 (Simulation of Copy-Free SSA Destruction). *Given a consumption safe SSA expression e (e ok), where $e \xrightarrow{IMP} e'$ under a well-formed injective mapping ρ , and equivalent initial states $\langle E_{SSA}; S_{SSA} \rangle \approx_{\rho} \langle E_{IMP}; S_{IMP} \rangle$.*

If the SSA program takes a step:

$$\langle E_{SSA}; S_{SSA}; e_{SSA} \rangle \longrightarrow \langle E'_{SSA}; S'_{SSA}; e'_{SSA} \rangle$$

Then the destructed IMP program will take a corresponding step:

$$\langle E_{IMP}; S_{IMP}; e_{IMP} \rangle \longrightarrow \langle E'_{IMP}; S'_{IMP}; e'_{IMP} \rangle$$

where $\langle E'_{SSA}; S'_{SSA} \rangle \approx_{\rho'} \langle E'_{IMP}; S'_{IMP} \rangle$.

Proof Sketch. The proof proceeds by structural induction on the small-step evaluation relation.

For jump, branch and return neither program modifies the store, so $S'_{SSA} = S_{SSA}$ and $S'_{IMP} = S_{IMP}$. Both programs symmetrically patch their environments by

substituting block arguments for block parameters. Since ρ maps corresponding variables in both programs, the environment patching preserves $\approx_\rho$, and we set $\rho' = \rho$.

Assume the SSA program evaluates $\boxed{\text{let } x_1 = \text{update } x_0 \text{ n in } e_\kappa}$. Because the SSA expression is consumption-safe (i.e., e_{SSA} ok) rule S-UPDATE guarantees $x_0 \notin FV(e_\kappa)$. Therefore, the destruction relation strictly applies D-INPLACE by Lemma 7.2.1, translating this to an IMP expression $\boxed{\text{let } x_1 = \text{update } x'_0 \text{ n in } e'_\kappa}$, where $\rho(x'_0) = x_0$.

In the SSA operational semantics, E-UPDATE-SSA allocates a fresh location l' such that $S'_{\text{SSA}}(l') = H \cdot n$, where $E_{\text{SSA}}(x_0) = l$ and $S_{\text{SSA}}(l) = H$. In the destructed IMP semantics, E-UPDATE-IMP destructively updates the original location l such that $S'_{\text{IMP}}(l) = H \cdot n$, where $E_{\text{IMP}}(x'_0) = l$.

We define the updated mapping $\rho' = \rho[x_1 \mapsto x_1] \setminus \{x'_0\}$, extending ρ with the freshly bound result variable x_1 and removing x'_0 , which is consumed and no longer live.

It remains to show $\langle E'_{\text{SSA}}; S'_{\text{SSA}} \rangle \approx_{\rho'} \langle E'_{\text{IMP}}; S'_{\text{IMP}} \rangle$.

That is, that for all $x \in \text{Domain}(E'_{\text{IMP}})$:

$$S'_{\text{IMP}}(E'_{\text{IMP}}(x)) = S'_{\text{SSA}}(E'_{\text{SSA}}(\rho'(x)))$$

We proceed by cases on x :

- **Case** $x = x_1$: $E'_{\text{IMP}}(x_1)$ maps to l , where $S'_{\text{IMP}}(l) = H \cdot n$. Under ρ' , x_1 maps to x_1 in SSA, where $E'_{\text{SSA}}(x_1) = l'$ and $S'_{\text{SSA}}(l') = H \cdot n$. Therefore the equivalence holds for x_1 .
- **Case** $x = x'_0$: $x'_0 \notin \text{Domain}(\rho')$ since it was removed. Because $x_0 \notin FV(e_\kappa)$, x'_0 will never be looked up in any future evaluation step, so this case is vacuously satisfied.

- **Case otherwise:** For all other variables x , neither store was modified at $E_{\text{IMP}}(x)$. Therefore, $S'_{\text{IMP}}(E'_{\text{IMP}}(x)) = S_{\text{IMP}}(E_{\text{IMP}}(x))$ and $S'_{\text{SSA}}(E'_{\text{SSA}}(\rho'(x))) = S_{\text{SSA}}(E_{\text{SSA}}(\rho(x)))$. By the initial state equivalence $\approx_\rho$, these are equivalent.

Thus semantic equivalence is preserved, and D-COPY is never evaluated. $\square$

that

A.12.3 SSA Construction Preserves Semantics

Theorem A.12.2 (Simulation of SSA Construction). *Given an IMP expression e_{IMP} where $e_{\text{IMP}} \xrightarrow{\text{SSA}} e_{\text{SSA}}$ under a well-formed injective mapping ρ , and given equivalent initial states $\langle E_{\text{IMP}}; S_{\text{IMP}} \rangle \approx_\rho \langle E_{\text{SSA}}; S_{\text{SSA}} \rangle$. If the IMP program takes a step:*

$$\langle E_{\text{IMP}}; S_{\text{IMP}}; e_{\text{IMP}} \rangle \longrightarrow \langle E'_{\text{IMP}}; S'_{\text{IMP}}; e'_{\text{IMP}} \rangle$$

Then there exists an updated mapping ρ' such that the SSA program takes a corresponding step:

$$\langle E_{\text{SSA}}; S_{\text{SSA}}; e_{\text{SSA}} \rangle \longrightarrow \langle E'_{\text{SSA}}; S'_{\text{SSA}}; e'_{\text{SSA}} \rangle$$

where $\langle E'_{\text{IMP}}; S'_{\text{IMP}} \rangle \approx_{\rho'} \langle E'_{\text{SSA}}; S'_{\text{SSA}} \rangle$.

Proof. By structural induction on the evaluation of e_{IMP} .

For jump, branch, and return, neither program modifies the store, so $S'_{\text{IMP}} = S_{\text{IMP}}$ and $S'_{\text{SSA}} = S_{\text{SSA}}$. Both programs patch their environments symmetrically by substituting block arguments for block parameters. Since ρ maps corresponding variables in both programs, the environment patching preserves $\approx_\rho$, and we set $\rho' = \rho$.

Assume the IMP program evaluates **let** $x_1 = \text{update! } x_0$ **n in** e_K .

By E-UPDATE-IMP, the store is mutated in-place, i.e., $S'_{\text{IMP}} = S_{\text{IMP}}[l \mapsto H \cdot n]$, where $l = E_{\text{IMP}}(x_0)$ and $S_{\text{IMP}}(l) = H$. By the construction relation C-UPDATE, the SSA program evaluates $\boxed{\text{update } \rho(x_0) \text{ } n}$, yielding a fresh variable x'_1 bound to a fresh location l' . By E-UPDATE-SSA, the SSA store is extended, i.e., $S'_{\text{SSA}} = S_{\text{SSA}}[l' \mapsto H \cdot n]$.

We define the updated mapping $\rho' = \rho[x_0 \mapsto x'_1, x_1 \mapsto x'_1]$. The mapping of $x_0 \mapsto x'_1$ reflects that C-UPDATE substitutes x'_1 for all occurrences of x_0 in e_k , so x_0 transfers its identity to x'_1 in the SSA program. The mapping of $x_1 \mapsto x'_1$ reflects that x_1 is the result variable in IMP and x'_1 is the corresponding result variable in SSA.

It remains to show that $\langle E'_{\text{IMP}}; S'_{\text{IMP}} \rangle \approx_{\rho'} \langle E'_{\text{SSA}}; S'_{\text{SSA}} \rangle$. That is:

$$\forall x \in \text{Domain}(E'_{\text{IMP}}) . S'_{\text{IMP}}(E'_{\text{IMP}}(x)) = S'_{\text{SSA}}(E'_{\text{SSA}}(\rho'(x)))$$

We proceed by cases on x :

- **Case $x = x_1$:** $E'_{\text{IMP}}(x_1)$ maps to l , where $S'_{\text{IMP}}(l) = H \cdot n$. Under ρ' , x_1 maps to x'_1 in SSA, where $E'_{\text{SSA}}(x'_1) = l'$ and $S'_{\text{SSA}}(l') = H \cdot n$. Therefore the equivalence holds for x_1 .
- **Case $x = x_0$:** $x_0 \notin \text{Domain}(\rho')$ since it was removed. By C-UPDATE, all occurrences of x_0 in e_k have been substituted by x'_1 , so x_0 will never be looked up in any future SSA evaluation step. This case is therefore vacuously satisfied.
- **Case otherwise:** For all other variables x , neither store was modified at $E_{\text{IMP}}(x)$. Therefore, $S'_{\text{IMP}}(E'_{\text{IMP}}(x)) = S_{\text{IMP}}(E_{\text{IMP}}(x))$ and $S'_{\text{SSA}}(E'_{\text{SSA}}(\rho'(x))) = S_{\text{SSA}}(E_{\text{SSA}}(\rho(x)))$. By the initial state equivalence $\approx_{\rho}$, these are equal.

Thus semantic equivalence is preserved under ρ' . $\square$

A.13 Mapping Optimizing Transformations to Building Blocks

We provide a mapping from transformations cataloged by Allen and Cocke [110] to the transformation building blocks defined in Section 7.4. This provides readers with a resource for them to map their own transformation to the necessary tightening checks. While the catalog does not include all possible transformations, it summarizes the vast majority.

Transformation	Building Block(s)	Preserves?
<i>Procedure Transformations</i>		
Procedure Integration	Inlining	✓
<i>Loop Transformations</i>		
Loop Unrolling	Inlining	✓
Loop Fusion/Jamming	Inlining	✓
Loop Unswitching	Code Motion + Injection	✗
<i>Redundancy Elimination</i>		
Common Subexpression Elimination	Replacement	✗
Global Value Numbering	Replacement	✗
Partial Redundancy Elimination	Code Motion + Replacement	✗
<i>Code Motion</i>		
Loop-Invariant Code Motion	Code Motion	✗
Hoisting	Code Motion	✗
Sinking	Code Motion	✓
<i>Constant Optimization</i>		
Constant Folding	Replacement	✗
Constant Propagation	Replacement	✗
<i>Elimination</i>		
Dead Code Elimination	Elimination	✓
Dead Argument Elimination	Elimination	✓
Unreachable Code Elimination	Elimination	✓
<i>Strength Reduction</i>		
Strength Reduction	Replacement + Injection	✗
Linear Function Test Replacement	Replacement + Elimination	✗
Carry	Replacement + Code Motion	✗
<i>Injection</i>		
Bounds Check Insertion	Injection	✗
Critical Edge Splitting	Injection	✓
<i>Machine Dependent</i>		
Instruction Scheduling	Code Motion	✗
Peephole Optimization	Replacement + Elimination	✗
Special Case Code Generation	Replacement	✗
Anchor Pointing	Elimination	✓
Parsing Methods	Code Motion	✗
Register Allocation	N/A	N/A
Storage Mapping	N/A	N/A
Shadow Variables	N/A	N/A

Table A.1: Mapping of Allen & Cocke’s catalogue of optimizing transformations [110] to our transformation building blocks, and whether they naturally preserve consumption-safety.